Abusing WCF Endpoints for Fun and Profit

muffsec.com/blog/abusing-wcf-endpoints-for-fun-and-profit

Posted by dugisec

TL;DR

A previous version of Check Point's ZoneAlarm antivirus and firewall product exposes a WCF interface which could be abused by low privilege users to trigger the execution of an update binary as SYSTEM. The issue has been disclosed by Check Point here. The exploitable WCF method takes the full path to the update binary as an argument which can be specified by the caller. The service attempts to prevent unauthorized processes from interacting with it by checking that any WCF clients are signed by Check Point. This can bypassed via DLL injection into a signed process or by simply signing the client (exploit code) with self-signed cert, which low priv users can trust on Windows. The service also only allows the execution of signed update binaries, but this can also be bypassed by either DLL hijacking a legitimately signed binary or again, with a self-signed certificate.

My friend Fabius Watson (@FabiusArtrel) recently gave what I consider to be a groundbreaking talk on abusing WCF endpoints. In 2018 he got a number of CVEs for privilege escalation and remote code execution in various commercial products which employed .NET based WCF services. Here are a few of them:

CVE-2018-13101 – KioskSimpleService Local Privilege Escalation

CVE-2018-10169 – Proton VPN Local Privilege Escalation

CVE-2018-10170 – NordVPN Local Privilege Escalation

CVE-2018-10190 – Private Internet Access Local Privilege Escalation

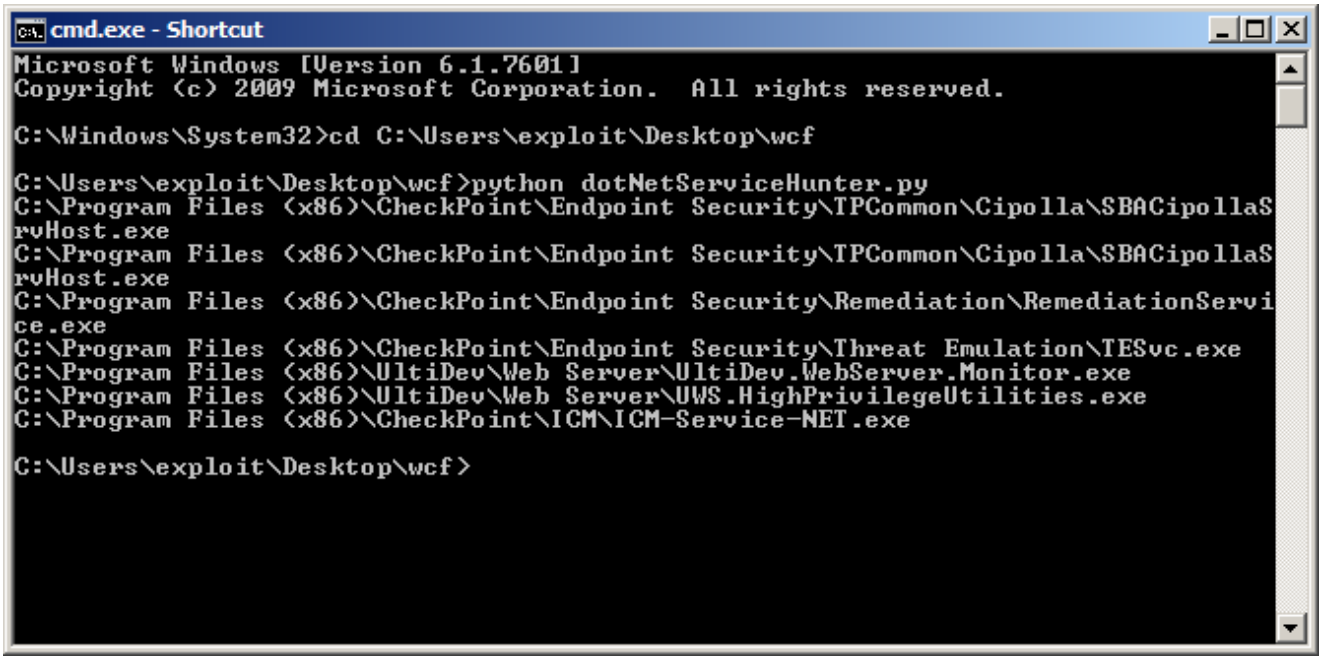
After reviewing the slides from his awesome talk at ekoparty 2018 I decided to go bug hunting. My first foray, looking into a ZoneAlarm by Check Point (a commercial antivirus product), was a success and a lot fun. So, with this post, I'd like to share my experience learning this bug class and writing a working exploit.

The first order of business was to install the software which is freely available at https://www.zonealarm.com/software/free-antivirus/. I have also made a vulnerable version of the software available here. After the ZoneAlarm tray pops up and seems to be running the installer is actually still going and there are services that will take some time to appear (maybe 30 minutes or more, sometimes less).

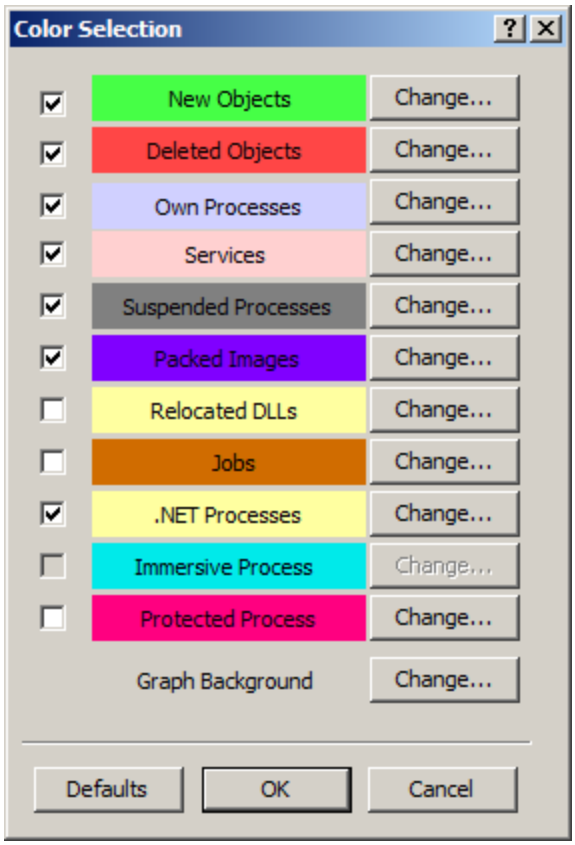
Once the install is truly complete, a python script created by @FabiusArtrel can be used to help quickly identify any services which may be vulnerable. The tool enumerates all services which meet the following criteria:

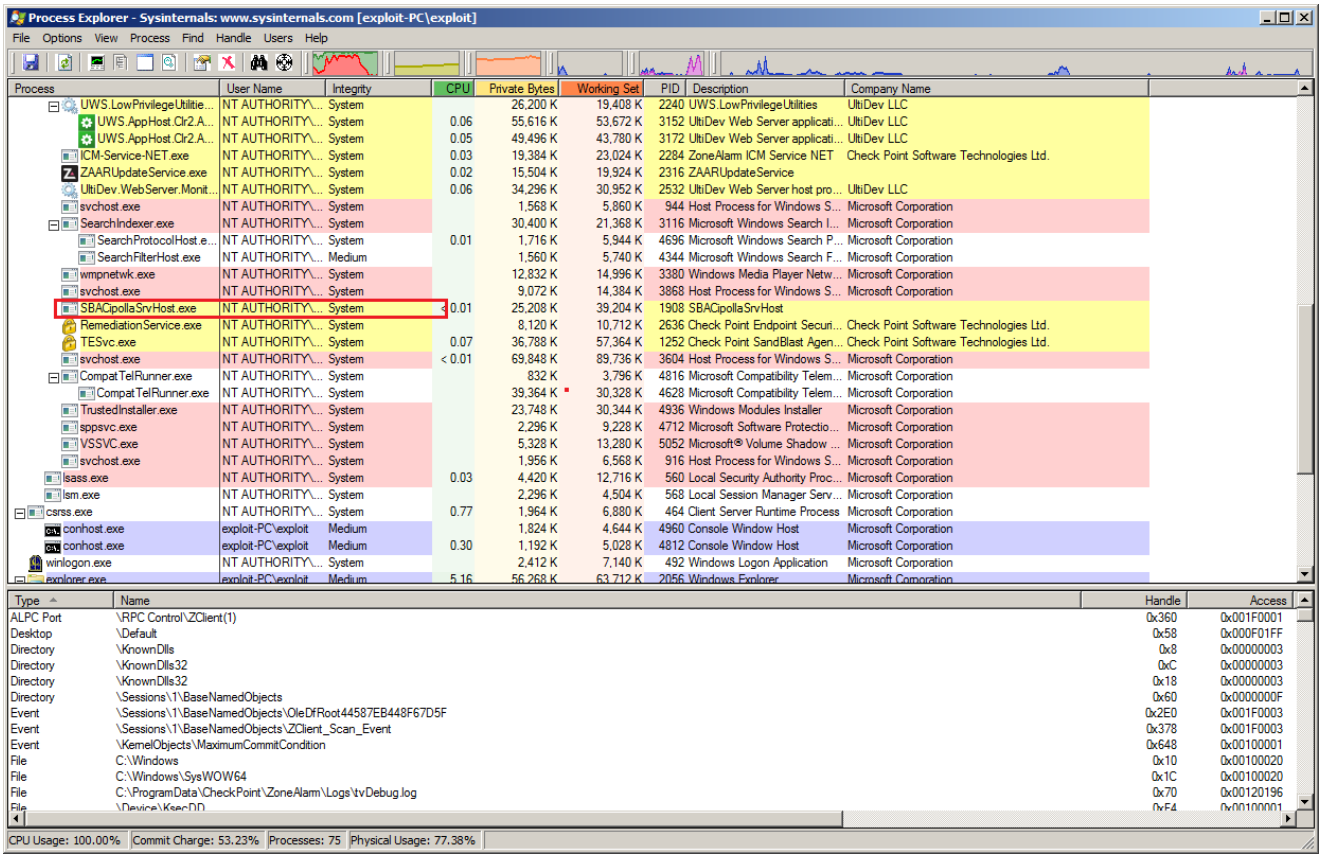
- Running as LocalSystem (NT AUTHORITY\SYSTEM)
- Service binary is a .NET application

Within the script a WMIC query is used to identify all services running as SYSTEM. Then the python module “pefile” is used to check if the service binary has mscoree.dll in the import table. All .NET applications depend on this library. Here’s what it looks like when run on a vulnerable system:

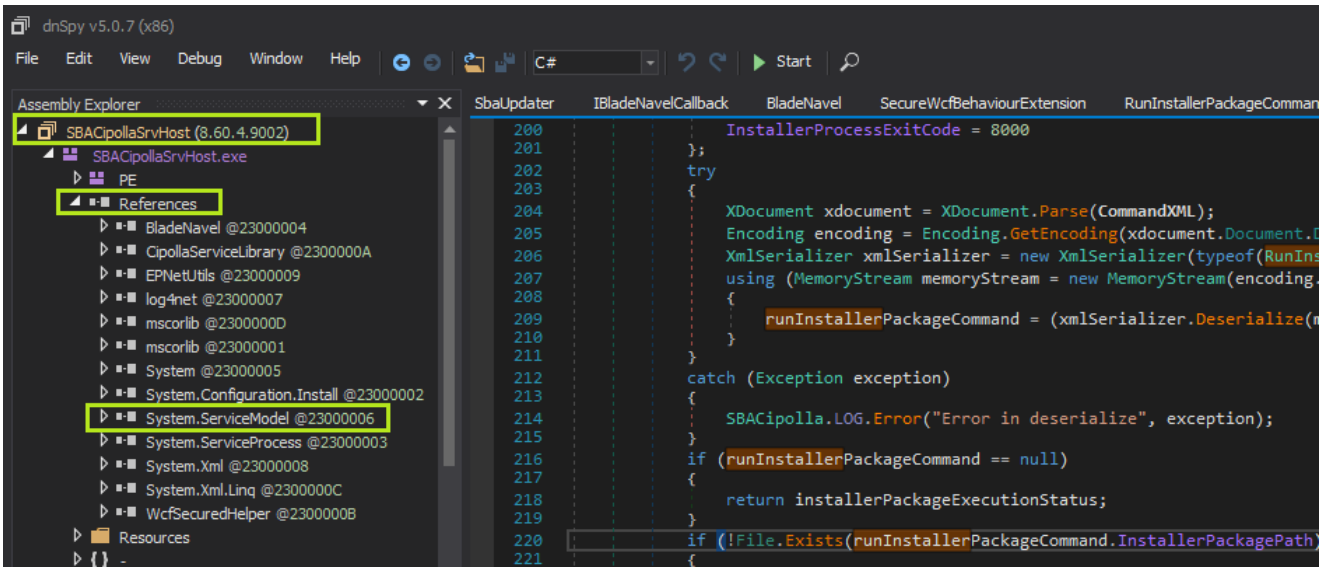


Process Explorer can also be used to help identify these type of services by going to Options > Configure Colors > .NET Process


Color Selection
New Objects
Change...
Deleted Objects
Change...
Own Processes
Change...
Services
Change...
Suspended Processes
Change...
Packed Images
Change...
Relocated DLLs
Change...
Jobs
Change...
.NET Processes
Change...
Immersive Process
Change...
Protected Process
Change...
Graph Background
Change...
Defaults
OK
Cancel

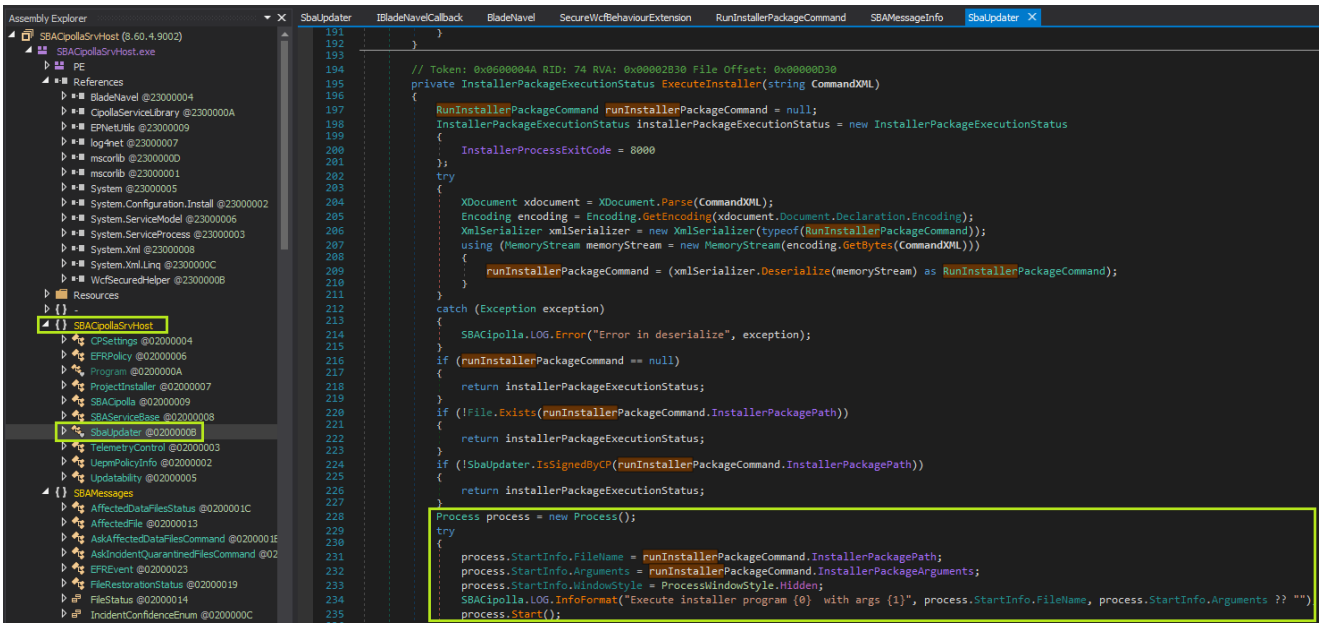

Process Explorer - Sysinternals: www.sysinternals.com [exploit-PC\exploit]
File Options View Process Find Handle Users Help
Process
User Name
Integrity
CPU
Private Bytes
Working Set
PID
Description
Company Name
UWS.LowPrivilegeUtilitie... NT AUTHORITY\... System 26,200 K 19,408 K 2240 UWS.LowPrivilegeUtilities UltiDev LLC
UWS.AppHost.Clr2.A... NT AUTHORITY\... System 0.06 55,616 K 53,672 K 3152 UltiDev Web Server applicati... UltiDev LLC
UWS.AppHost.Clr2.A... NT AUTHORITY\... System 0.05 49,496 K 43,780 K 3172 UltiDev Web Server applicati... UltiDev LLC
ICM-Service-NET.exe NT AUTHORITY\... System 0.03 19,384 K 23,024 K 2284 ZoneAlarm ICM Service NET Check Point Software Technologies Ltd.
ZAARUpdateService.exe NT AUTHORITY\... System 0.02 15,504 K 19,924 K 2316 ZAARUpdateService
UltiDev.WebServer.Monit... NT AUTHORITY\... System 0.06 34,296 K 30,952 K 2532 UltiDev Web Server host pro... UltiDev LLC
svchost.exe NT AUTHORITY\... System 1,568 K 5,860 K 944 Host Process for Windows S... Microsoft Corporation
SearchIndexer.exe NT AUTHORITY\... System 30,400 K 21,368 K 3116 Microsoft Windows Search I... Microsoft Corporation
SearchProtocolHost.e... NT AUTHORITY\... System 0.01 1,716 K 5,944 K 4696 Microsoft Windows Search P... Microsoft Corporation
SearchFilterHost.exe NT AUTHORITY\... Medium 1,560 K 5,740 K 4344 Microsoft Windows Search F... Microsoft Corporation
wmpnetwk.exe NT AUTHORITY\... System 12,832 K 14,996 K 3380 Windows Media Player Netw... Microsoft Corporation
svchost.exe NT AUTHORITY\... System 9,072 K 14,384 K 3868 Host Process for Windows S... Microsoft Corporation
SBACipollaSrvHost.exe NT AUTHORITY\... System < 0.01 25,208 K 39,204 K 1908 SBACipollaSrvHost
RemediationService.exe NT AUTHORITY\... System 8,120 K 10,712 K 2636 Check Point Endpoint Securi... Check Point Software Technologies Ltd.
TESvc.exe NT AUTHORITY\... System 0.07 36,788 K 57,364 K 1252 Check Point SandBlast Agen... Check Point Software Technologies Ltd.
svchost.exe NT AUTHORITY\... System < 0.01 69,848 K 89,736 K 3604 Host Process for Windows S... Microsoft Corporation
CompatTelRunner.exe NT AUTHORITY\... System 832 K 3,796 K 4816 Microsoft Compatibility Telem... Microsoft Corporation
CompatTelRunner.exe NT AUTHORITY\... System 39,364 K 30,328 K 4628 Microsoft Compatibility Telem... Microsoft Corporation
TrustedInstaller.exe NT AUTHORITY\... System 23,748 K 30,344 K 4936 Windows Modules Installer Microsoft Corporation
sppsvc.exe NT AUTHORITY\... System 2,296 K 9,228 K 4712 Microsoft Software Protectio... Microsoft Corporation
VSSVC.exe NT AUTHORITY\... System 5,328 K 13,280 K 5052 Microsoft® Volume Shadow ... Microsoft Corporation
svchost.exe NT AUTHORITY\... System 1,956 K 6,568 K 916 Host Process for Windows S... Microsoft Corporation
lsass.exe NT AUTHORITY\... System 0.03 4,420 K 12,716 K 560 Local Security Authority Proc... Microsoft Corporation
lsm.exe NT AUTHORITY\... System 2,296 K 4,504 K 568 Local Session Manager Serv... Microsoft Corporation
csrss.exe NT AUTHORITY\... System 0.77 1,964 K 6,880 K 464 Client Server Runtime Process Microsoft Corporation
conhost.exe exploit-PC\exploit Medium 1,824 K 4,644 K 4960 Console Window Host Microsoft Corporation
conhost.exe exploit-PC\exploit Medium 0.30 1,192 K 5,028 K 4812 Console Window Host Microsoft Corporation
winlogon.exe NT AUTHORITY\... System 2,412 K 7,140 K 492 Windows Logon Application Microsoft Corporation
explorer.exe exploit-PC\exploit Medium 5.16 56,268 K 63,712 K 2056 Windows Explorer Microsoft Corporation
Type
Name
Handle
Access
ALPC Port \RPC Control\ZClient(1) 0x360 0x001F0001
Desktop \Default 0x58 0x000F01FF
Directory \KnownDlls 0x8 0x00000003
Directory \KnownDlls32 0xC 0x00000003
Directory \KnownDlls32 0x18 0x00000003
Directory \Sessions\1\BaseNamedObjects 0x60 0x0000000F
Event \Sessions\1\BaseNamedObjects\OleDfRoot44587EB448F67D5F 0x2E0 0x001F0003
Event \Sessions\1\BaseNamedObjects\ZClient_Scan_Event 0x378 0x001F0003
Event \KernelObjects\MaximumCommitCondition 0x648 0x00100001
File C:\Windows 0x10 0x00100020
File C:\Windows\SysWOW64 0x1C 0x00100020
File C:\ProgramData\CheckPoint\ZoneAlarm\Logs\tvDebug.log 0x70 0x00120196
CPU Usage: 100.00%
Commit Charge: 53.23%
Processes: 75
Physical Usage: 77.38%

So, with some candidate services to look at the next thing to do is open them up in dnSpy, an awesome .NET decompiler. Although there are many .NET services running they may not all be using WCF. All WCF services depend on System.ServiceModel, so right away we can check for a reference to this assembly. Only one of the ZoneAlarm services (SBACipollaSrvHost.exe) references this:

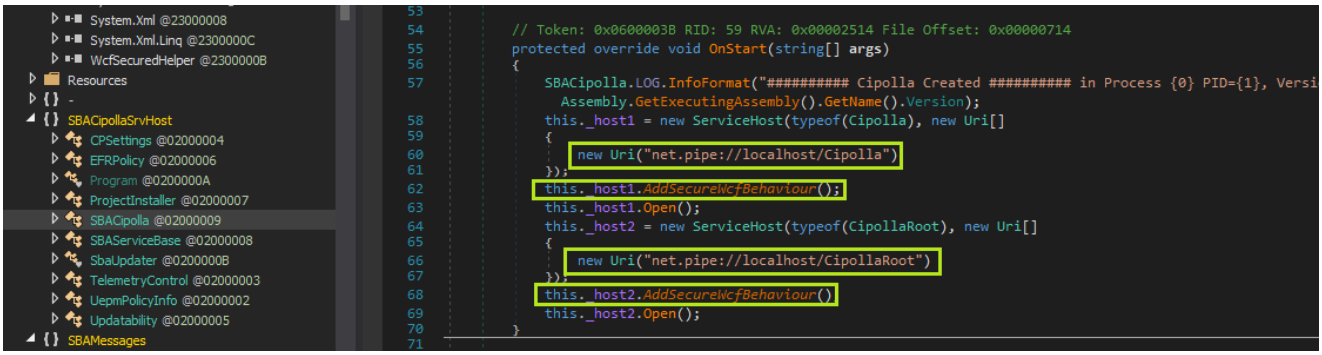


Great, so now that we know there is a WCF service running as SYSTEM we can check to see if it exposes any methods which might be exploitable. In some cases there will be methods which literally take a command to run as input, this results in a really easy win. In other cases, it may not be as direct. It's also possible that the service exposes no methods which can be abused for code execution by any means.

After trolling the source code one method caught my eye called **OnCommandReceived**. After tracing the series of calls that this method makes I determined that it was used to execute an installer binary in a method called **ExecuteInstaller** which looks like this:



After seeing the name of the method, and that it was used to spawn new processes, I actually didn't take the time to notice that it only launches checkpoint-signed binaries (see line 224). Instead I moved straight to figuring out how to talk to the WCF service so that I could try triggering this functionality. So next on my list of things to do was to learn more about the service. In the SBACipolla class we can see that two named-pipe service endpoints are created, **Cipolla** and **CipollaRoot**. WCF services can operate over a variety of transport protocols. If HTTP or TCP protocols are used it may be possible to exploit the service remotely. In this case it's using named-pipes, so local privilege escalation will be the only angle available:

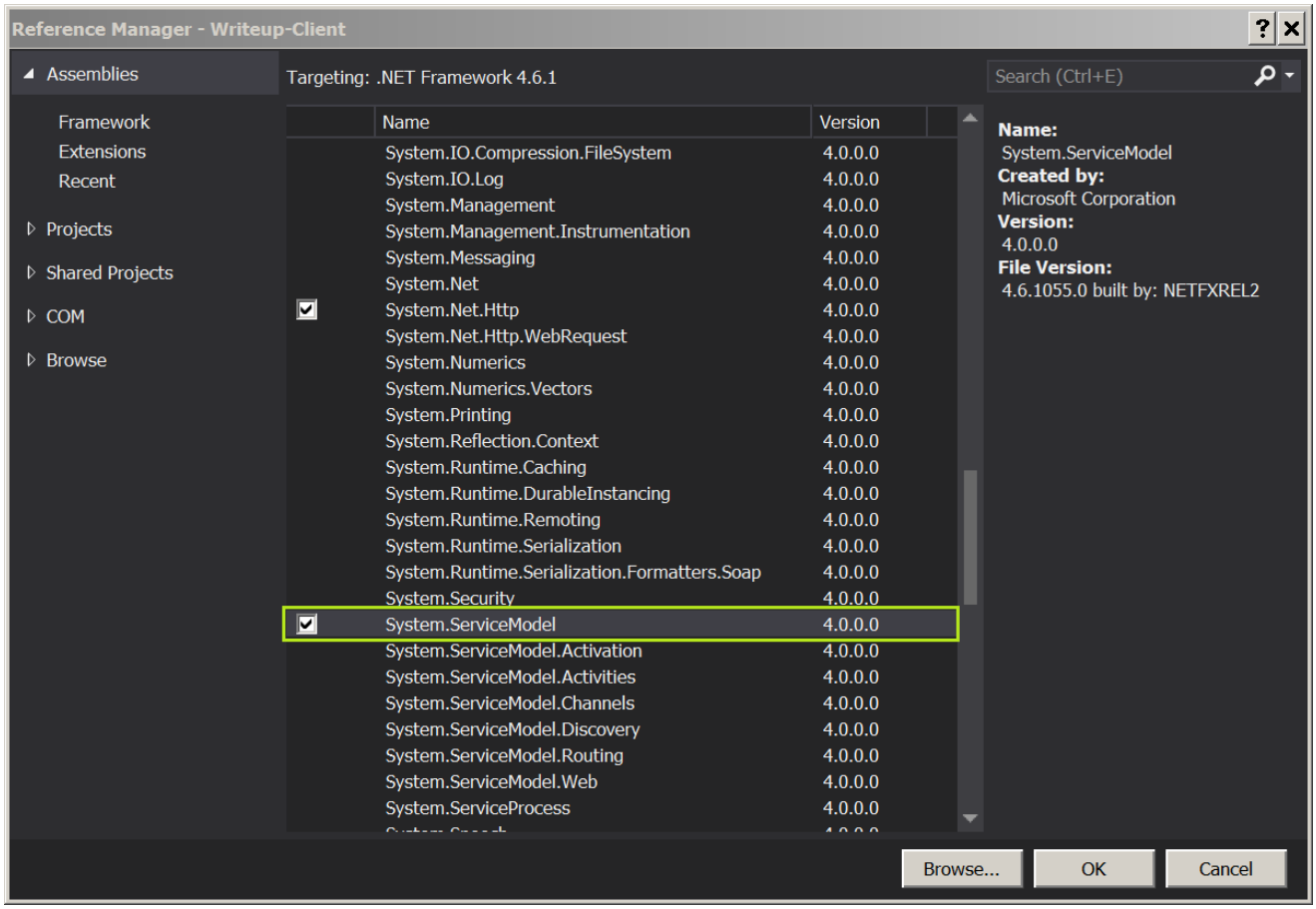


The service endpoints also have a custom **AddSecureWcfBehavior** method called on them, a harbinger that there may be some attempt by the developers to lock down these services.

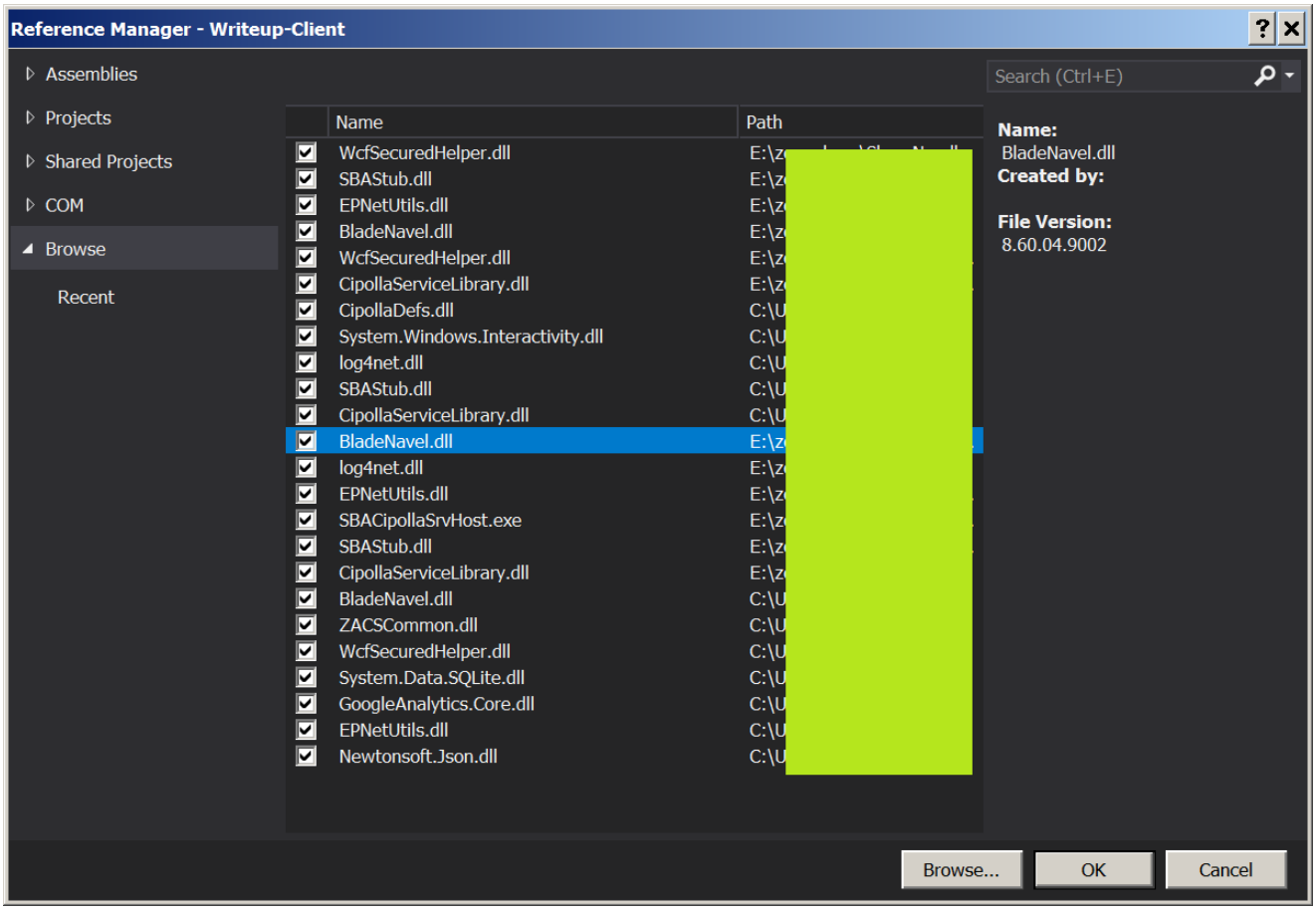
After seeing this, I used a tool called IO Ninja to sniff on the named-pipes. I turned it on then attempted to update ZoneAlarm multiple times, hoping to see some action on the pipe which might help me better understand what was going on, but nothing ever came across. Since there was no luck to be had with that angle I turned to trying to find a legitimate WCF client to connect to this service with. Eventually I stumbled on SBAStub.dll (found in the same folder as the service binary: C:\Program Files (x86)\CheckPoint\Endpoint

Security\TPCommon\Cipolla) which has a method called **SetUpWCFConnection** that connects to the **CipollaRoot** named-pipe, and another method called **SendCommand** which sounded really nice

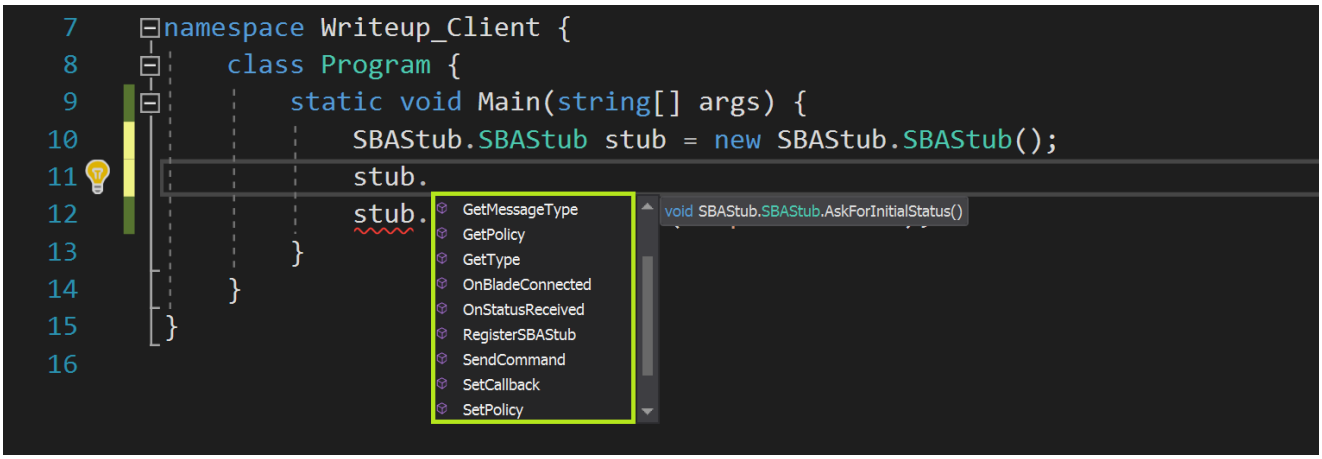
To test this out I created a new C# Console App project in visual studio and added a reference to System.ServiceModel to the project (necessary for WCF):



A reference to SBAStub.dll was also needed. Because I wasn't sure if there would be a dependency chain within this library, I added references to pretty much all the DLLs in the same folder as a shotgun approach to ensure everything would work:



To test this out I started by creating a new SBAStub object and then let intellisense within Visual Studio let me know which methods were available on that object:

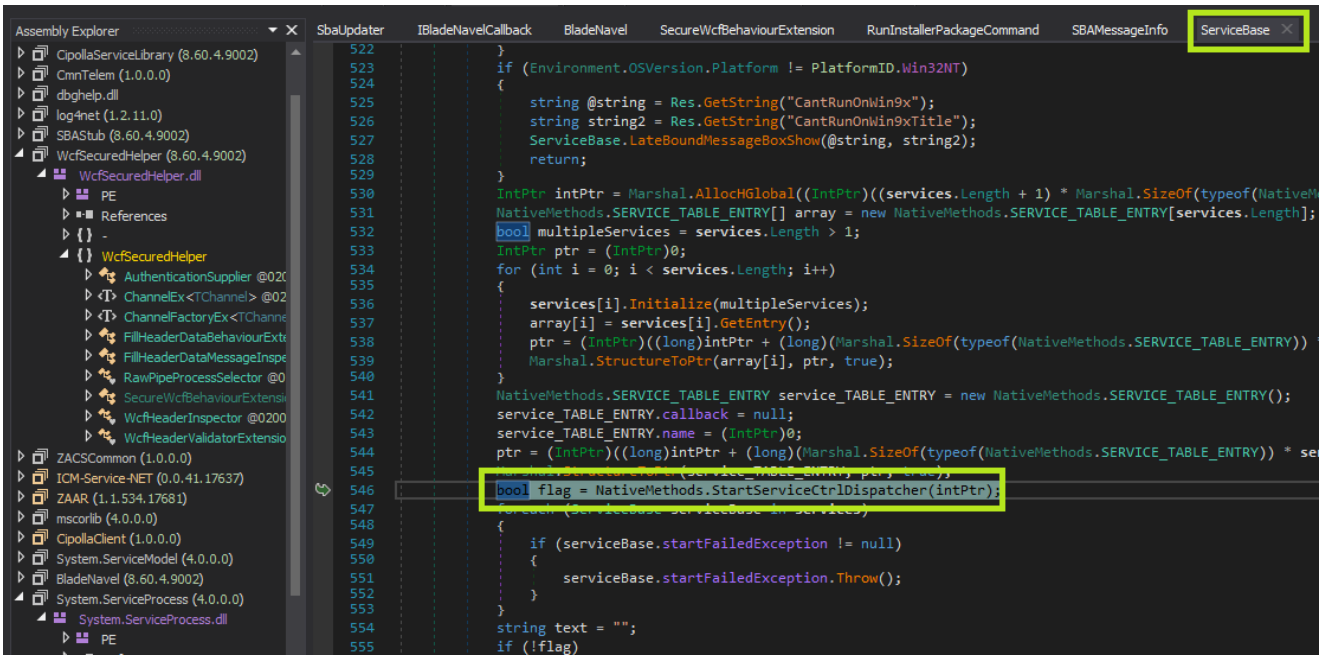


I tried calling **RegisterSBAStub** because it took a simple string as an argument and because when it works the registration is logged in C:\ProgramData\CheckPoint\Logs\Cipolla.log. Seeing a log entry as a result of my code running would let me know that I was successfully interacting with the service. Of course, after running this code nothing showed up in the logs. My attempt at troubleshooting looked like this:

- Attach to the SBACipollaSrvHost.exe process with dnSpy (running as admin)

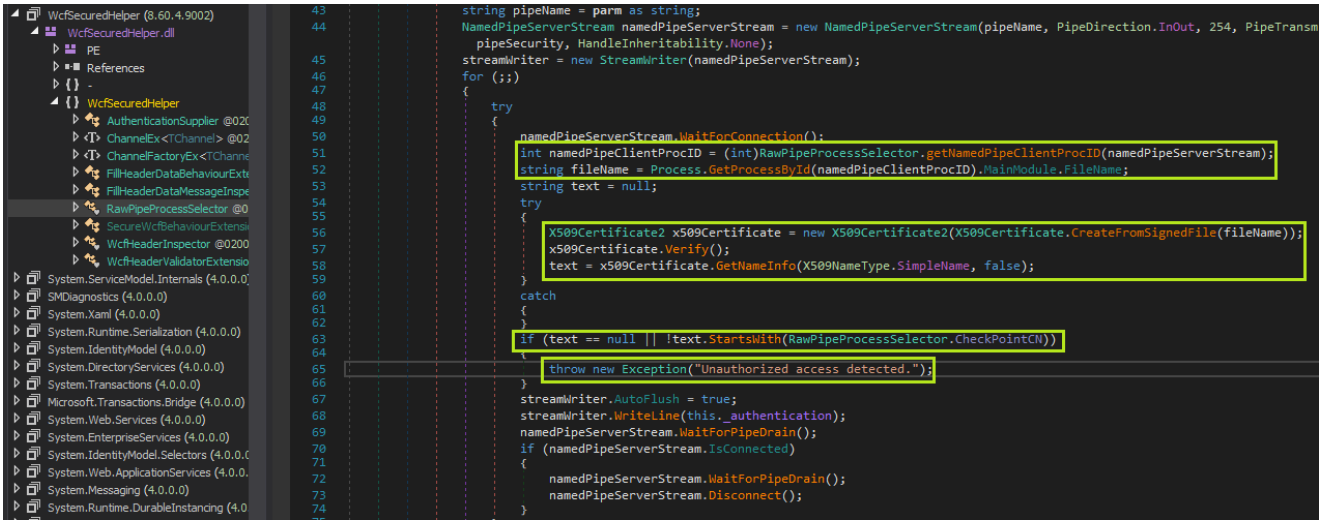
- Hit “Break All” (the pause button)
- Run the client code
- Single Step

This was a failing strategy. Every time I would step (whether it was over, into, or out of) my client code would just finish running and I wouldn’t see any action in the debugger before landing back here:



This was the same line of code I was on when I initially paused execution -_-

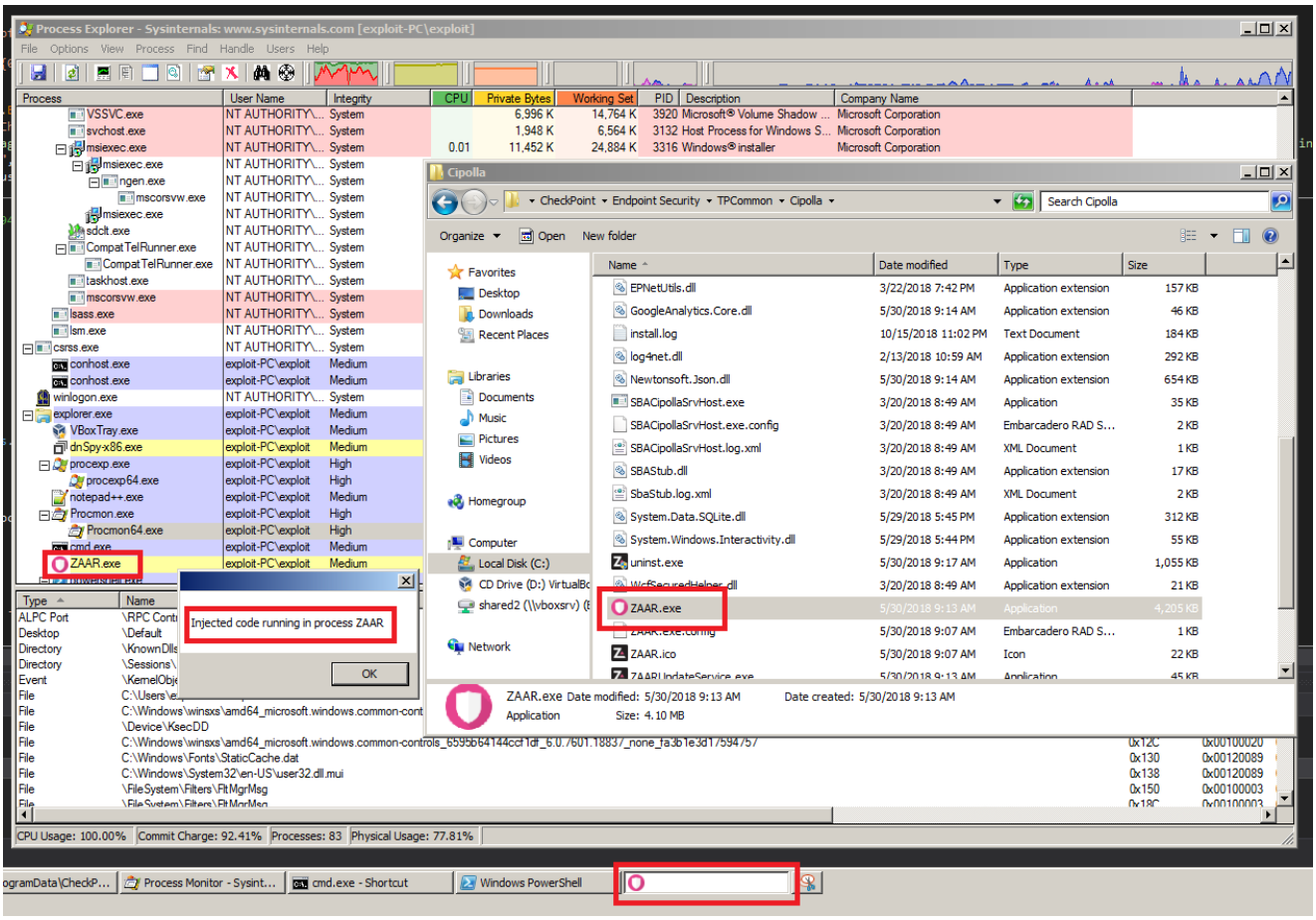
After spending a lot of time browsing the source I ended up finding a location which seemed like a good break point. It was inside of WcfSecuredHelper.dll around the point where the named-pipe server starts listening for connections:



I tried adding a break point on the if statement on line 63, attaching, then running my code. Sure enough, the service was throwing an exception “Unauthorized access detected”. On lines 50 and 51 the filename of the process attempting to connect to the name pipe is stored in the **fileName** variable. On lines 56-58 it checks to see if the program is signed with a valid certificate, and stores the “Common Name” (CN) portion of the certificate in the **text** variable. The if statement on line 63 checks to see if the CN starts with “Check Point Software Technologies”. Since the client code that we have written is not signed it is going to fail this check which is why we aren’t seeing the SBA Stub get registered in the logs.

From here my thought was to inject this client code into a legitimate checkpoint-signed binary. My first approach to achieving this was by getting a meterpreter shell on the system, migrating the session into a CheckPoint process, then using execute -m (execute from memory) to run my client code. Unfortunately, I never had success getting execution from memory to work in metasploit, even when trying to run standard binaries (rather than .NET binaries). After some googling I found a project on github called SharpNeedle that facilitates the injection of .NET code into any x86 process. Within the C:\Program Files (x86)\CheckPoint\Endpoint Security\TPCommon\Cipolla” directory I found a legitimately signed program called ZAAR.exe which I could start up and then inject code into. The following is just a PoC of the code injection:

```
78    public class Example {
79
80        public static int EntryPoint(string pwzArgument) {
81            string processName = Process.GetCurrentProcess().ProcessName;
82            MessageBox.Show("Injected code running in process " + processName);
83
84            /*
```



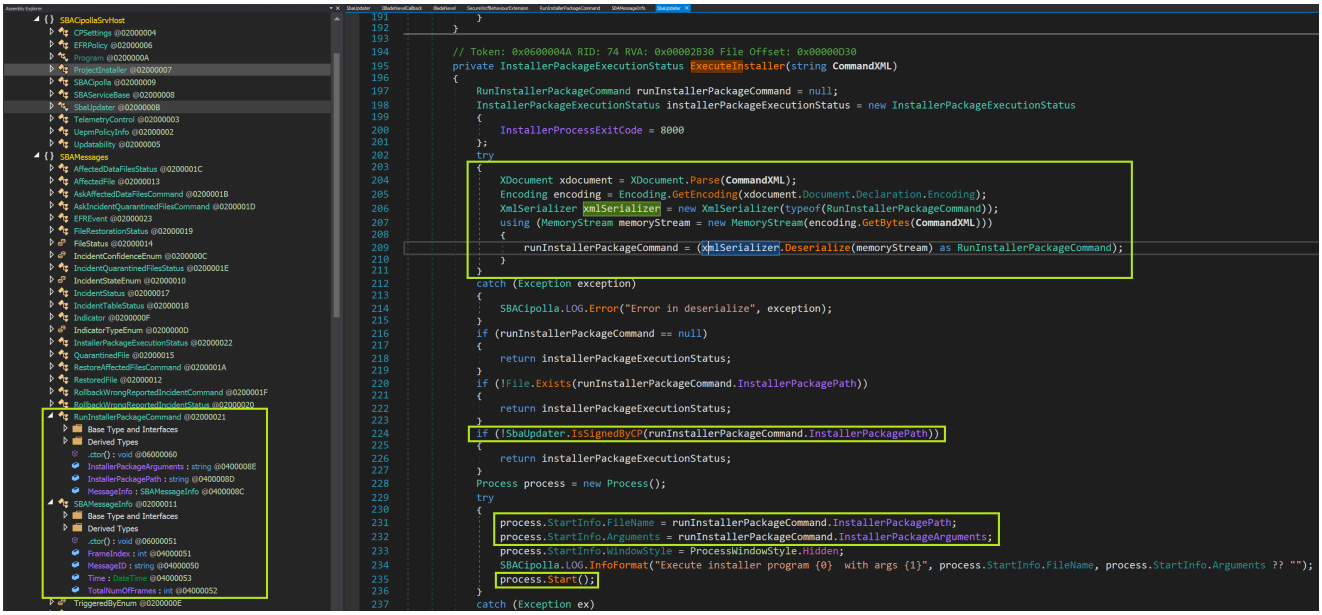
Great so with that working we now have a way to connect to the named pipe and can try registering a stub again. Here's the code:

```
77      [SecuritySafeCritical]
78      public class Example {
79
80          public static int EntryPoint(string pwzArgument) {
81              string processName = Process.GetCurrentProcess().ProcessName;
82              //MessageBox.Show("Injected code running in process " + processName);
83
84              try {
85                  SBAStub.SBAStub stub = new SBAStub.SBAStub();
86                  stub.RegisterSBAStub("exploit-stub");
87
88              }
89              catch (Exception e) {
90                  MessageBox.Show("Exception: " + e);
91              }
92
```

And here we see the stub registration was reflected in the log file this time (C:\ProgramData\CheckPoint\Logs\Cipolla.log):

```
180526
180527     </antiExploit>
180528 </TEPolicy>
180529 20181118 13:18:50.458 DEBUG 3 CipollaServiceLibrary.CipollaRoot.RegisterSBAStub:Enter  ComponentUniqName=exploit-stub
180530 20181118 13:18:50.459 DEBUG 3 CipollaServiceLibrary.CipollaRoot.RegisterSBAStub:Exit, ComponentUniqName=exploit-stub
180531 20181118 13:18:57.053 DEBUG 5 BladeNavel.BladeNavel.SetPolicy:Exit, PolicyID=100
180532 20181118 13:18:57.053 DEBUG 5 BladeNavel.BladeNavel.SetPolicy:Enter, PolicyID=400
```

Very sick! The next thing to do was to start playing with the **SendCommand** method of the **SBAStub** object. So, when calling **SendCommand** (which takes a string of XML called **CommandXML**), the arguments are eventually passed to a function called **ExecuteInstaller** which I'll show again here:



```
191         }
192     }
193
194     // Token: 0x0600004A RID: 74 RVA: 0x00002B30 File Offset: 0x00000D30
195     private InstallerPackageExecutionStatus ExecuteInstaller(string CommandXML)
196     {
197         RunInstallerPackageCommand runInstallerPackageCommand = null;
198         InstallerPackageExecutionStatus installerPackageExecutionStatus = new InstallerPackageExecutionStatus
199         {
200             InstallerProcessExitCode = 8000
201         };
202         try
203         {
204             XDocument xdocument = XDocument.Parse(CommandXML);
205             Encoding encoding = Encoding.GetEncoding(xdocument.Document.Declaration.Encoding);
206             XmlSerializer xmlSerializer = new XmlSerializer(typeof(RunInstallerPackageCommand));
207             using (MemoryStream memoryStream = new MemoryStream(encoding.GetBytes(CommandXML)))
208             {
209                 runInstallerPackageCommand = (xmlSerializer.Deserialize(memoryStream) as RunInstallerPackageCommand);
210             }
211         }
212         catch (Exception exception)
213         {
214             SBACipolla.LOG.Error("Error in deserialize", exception);
215         }
216         if (runInstallerPackageCommand == null)
217         {
218             return installerPackageExecutionStatus;
219         }
220         if (!File.Exists(runInstallerPackageCommand.InstallerPackagePath))
221         {
222             return installerPackageExecutionStatus;
223         }
224         if (!SbaUpdater.IsSignedByCP(runInstallerPackageCommand.InstallerPackagePath))
225         {
226             return installerPackageExecutionStatus;
227         }
228         Process process = new Process();
229         try
230         {
231             process.StartInfo.FileName = runInstallerPackageCommand.InstallerPackagePath;
232             process.StartInfo.Arguments = runInstallerPackageCommand.InstallerPackageArguments;
233             process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
234             SBACipolla.LOG.InfoFormat("Execute installer program {0}  with args {1}", process.StartInfo.FileName, process.StartInfo.Arguments ?? "");
235             process.Start();
236         }
237         catch (Exception ex)
```

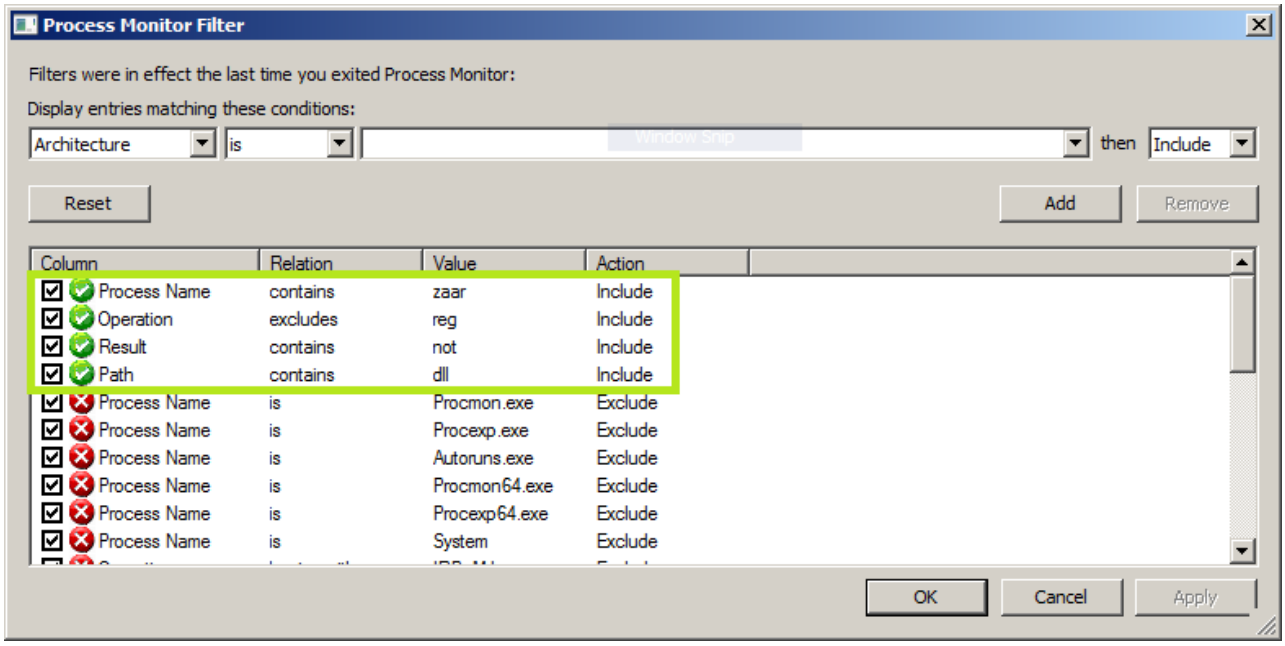
On lines 204-211 the **CommandXML** is deserialized into a **RunInstallerPackageCommand** object which is a custom class defined in the service binary. The class has three fields, (string) **InstallerPackagePath,** (string) **InstallerPackageArguments,** and another custom class (SBAMessageInfo) **MessageInfo**. The most interesting field is the **InstallerPackagePath** because that is used to start a process in the context of the service which is running as SYSTEM.

On line 224 we can see there is a check to verify that the program pointed to by **InstallerPackagePath** is signed by Check Point.

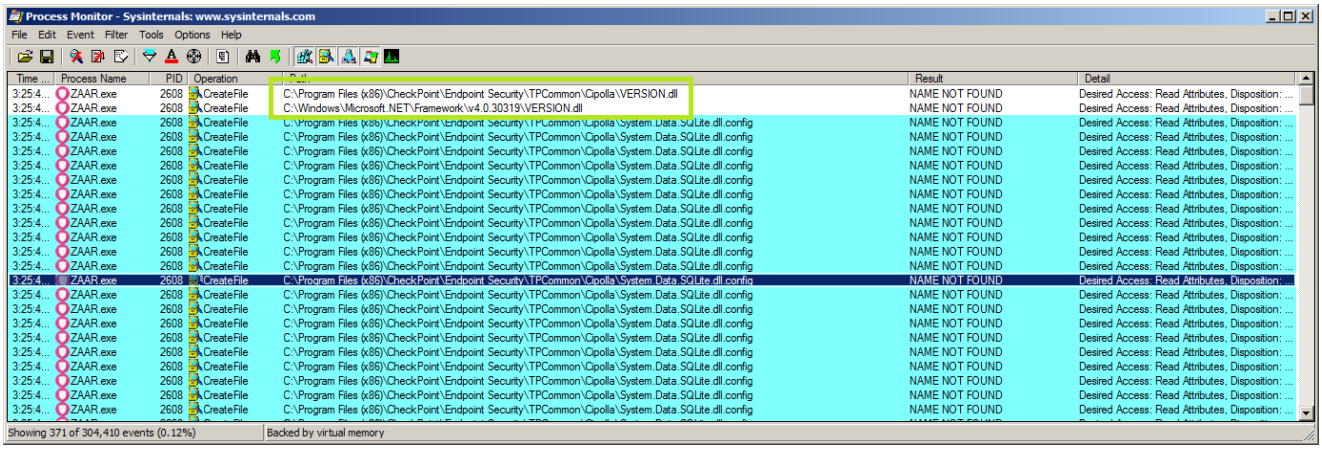
Lines 231-232 load the arguments into a Process object which is then then started on line 235.

Great! We can start any Check Point signed binary as SYSTEM.

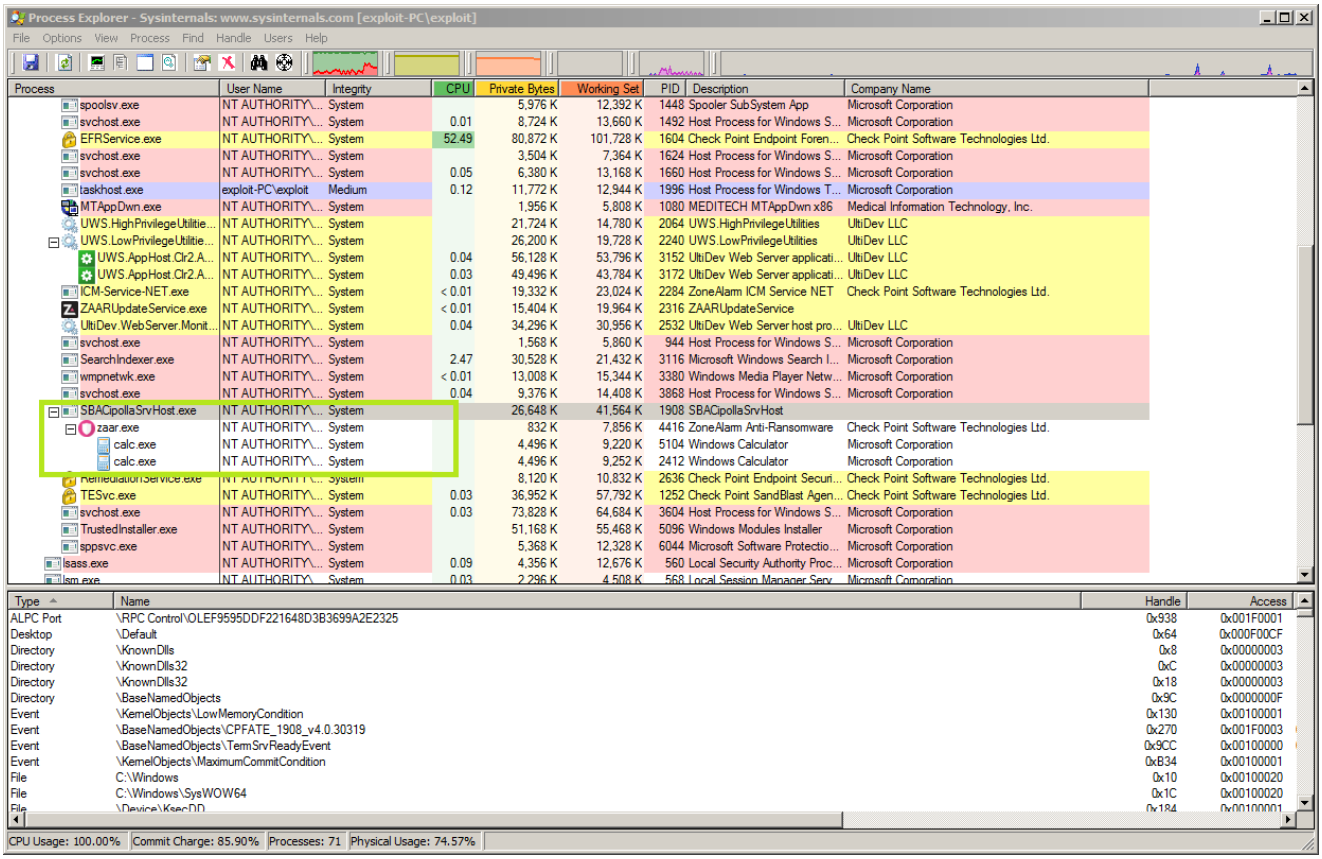
You may be wondering at this point how this can be exploited for arbitrary privileged code execution. One way is with a simple DLL hijack. I again turned to zaar.exe as a dummy signed binary that would help facilitate exploitation. I loaded up Process Monitor with the following filters:



Then launched C:\Program Files (x86)\CheckPoint\Endpoint Security\TPCommon\Cipolla\zaar.exe:



As shown outlined above ZAAR.exe attempts to load a file called **version.dll** in the current directory but it's not found. This means if a malicious version.dll was placed in the same directory as ZAAR.exe it would be executed. Since we're operating a low privilege user we can place a file in the C:\Program Files (x86)\CheckPoint\Endpoint Security\TPCommon\Cipolla\ directory, but since we control the full path to the update binary that gets executed by the service we can simply copy this to an arbitrary folder like temp and then place a DLL alongside it called version.dll with any payload we like. Here's the end result:



You can see that zaar.exe was launched as a SYSTEM process as a child of the SBACipollaSrvHost process, and it has two children, calc.exe, also running as SYSTEM.

Once I got to this point I contacted Check Point to disclose the issue. They came back and said that the PoC didn't work when the antivirus is enabled…whoops! All this time I forgot that in order to make testing easier I had disabled the AV. There are a few features about ZoneAlarm that can be configured an admin on the system, one in particular is "Application Control" which, when enabled, will block dll injection into the zaar.exe process that was needed to talk to the service. It also kept removing the **version.dll** from disk that was being used to launch calc. Damn!

To deal with this, I spent quite a while trying to find alternative means of DLL injection which would not be blocked by the AV, but all attempts were failed. Instead I ended up taking a totally different approach. This great article by Matt Graeber of SpecterOps describes a powershell cmdlet which makes it easy for low privilege users to sign code with a self-signed certificate and have the OS trust the certificate. Using this technique we sign the exploit code so that it's possible to talk to the WCF service without injecting into another process. Additionally, we'll be able to sign our payload which will be launched by the service and since this will be an ordinary executable it won't be removed by the AV. The process looks like this:

```
$cert = New-SelfSignedCertificate -certstorelocation cert:\CurrentUser\my -
dnsname checkpoint.com -Subject "CN=Check Point Software Technologies
Ltd." -Type CodeSigningCert
Export-Certificate -Type CERT -FilePath c:\tmp\MSKernel32Root_Cloned.cer -
Cert $cert
Import-Certificate -FilePath c:\tmp\MSKernel32Root_Cloned.cer -
CertStoreLocation Cert:\CurrentUser\Root\
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\tmp\exploit.exe
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\tmp\payload.exe
```

After signing the both these files and running the exploit arbitrary privileged code execution will take place with all AV features enabled