WINTAP

v6 Developer's Guide

Abstract

An extensible host-based agent for Windows providing real-time event collection, analytics, and response.

Frye, David J.

frye3@llnl.gov

Contents

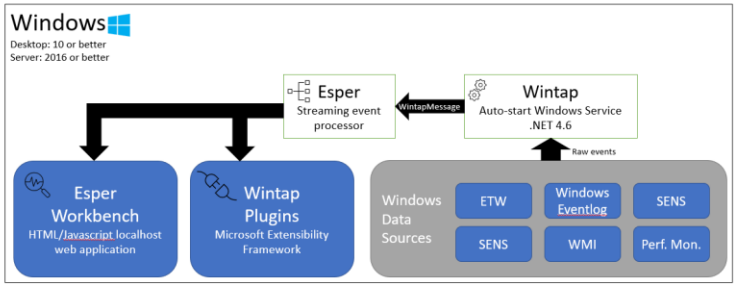
Background

In any large-scale Windows environment, there is likely a re-occurring need to collect, analyze, and react to host-based data whether it be for IT operations, cyber security response, or cyber research. Typically, these needs are met through the development of one-off scripts, utilities, single-purpose agents. One problem with this approach is scalability. As more data collection needs arise, more and more one-off scripts (and the like) are deployed to endpoints slowly increasing the combined resource load and associated "agent bloat".

In addition, because Windows is a complex and modular operating system, a myriad of APIs exist with little consistency in how they describe the data they provide. For example, a process might be identified by name from one API, by PID from another, and by a collection of thread IDs from a third. A user may be identified by SID, by Active Directory GUID, or by a username. A file might be identified by a logical path, by a physical path or by one of the other 17 observed OS path variants. In such an ecosystem, the one-off approach to host-based data collection creates datasets that, while useful for a specific need, likely lack any chance for reusability or discoverability.

Wintap

High-level design



Wintap aspires to solve these problems by providing the following capabilities:

- A singular and extensible service-based runtime environment. Wintap provides an easy-to-use extensibility API so that new functionality can be merged through the development of independent library modules while maintaining a single agent footprint on the endpoint.
- A unified data model. Wintap provides a single, strongly typed data model for describing data sourced from the myriad of underlying APIs it consumes from. This unified model forms a foundation upon which data can be consistently discovered, consumed, combined, and correlated.
- API abstraction. Wintap integrates deeply into many low-level Windows event streams such as Event Tracing for Windows (ETW), COM+ System Notifications and others. Plugin authors can easily take advantage of these rich event sources without needing to implement any of the low-level API details.
- Data discovery. Wintap provides an integrated, locally hosted web-based analytic "workbench" from where real-time event streams can be queried and explored.

Minimum System Requirements

Desktop OS: Windows 10, 64-bit

Server OS: Windows Server 2008 R2, 64-bit

.NET Framework: 4.8

Memory: 4GB

Frameworks

Wintap makes heavy use of the following 3rd party development frameworks:

TraceEvent. A Microsoft developed .NET wrapper around the native C++ ETW API. More Info: https://github.com/microsoft/perfview/blob/main/documentation/TraceEvent/TraceEventProgrammersGuide.md

Esper. Esper is a fast and scalable complex event processing and streaming analytics engine. More Info: https://www.espertech.com/

Managed Extensibility Framework. A Microsoft developed framework for developing highly decoupled applications (plugin architecture). More info: https://docs.microsoft.com/en-us/dotnet/framework/mef/

OWIN. A light-weight web hosting framework. More info: http://owin.org/

Installation

Installation of Wintap is achieved by running the installer under administrative privileges (setup.msi). The installer is a Windows Installer package (MSI). The installer can be run interactively, by simply opening the file from the Windows shell (e.g. double-click the file). As an MSI compliant installer, it also supports several command-line options which can be invoked from an administrative command shell as follows:

Install with UI: setup.msi

Install without UI: msiexec.exe /I setup.msi /qn

Uninstall: via Add/Remove Programs in the Windows control panel space.

All of the files and subdirectories for Wintap are installed into the following root directory:

%PROGRAMFILES%\Wintap

Wintap will create the following root registry key for its own usage:

HKEY_LOCAL_MACHINE\SOFTWARE\Wintap

After the MSI installation completes, run the following command from an administrative command window to create the service and start it:

```
Sc.exe create Wintap binPath= "C:\Program Files\Wintap\Wintap.exe" start= Auto

Sc.exe start Wintap
```

Configuration

Some Wintap behavior can be controlled via the Wintap configuration file. Within the Wintap directory, there is a text file named Wintap.exe.config. This file is protected by Windows NTFS permissions such that only Windows administrators should be able to modify this file. Any modifications to this file will require a restart of the Wintap service to become effective. This file contains a user settings section that looks like this:

LoggingLevel. Wintap writes a diagnostics log file at %WINDIR%\Temp\Wintap.log. Wintap supports two modes of logging Normal (default), and Debug. Changing this to Debug will result in increased logging detail and may be useful in some troubleshooting scenarios. The log is overwritten on startup.

EnableWorkbench. If true Wintap will host the analytic workbench web application (localhost accessible only). Value is False by default.

ApiPort. The port used to host the REST API required for the Workbench.

Profile. Can be either Production or Developer. If set to Production (default), Wintap will attempt to enforce CPU and Memory utilization constraints upon itself to limit host impact. The threshold is 20% CPU (all cores) and/or 700MB RAM. Developer mode removes these constraints.

Collector enablement. The remainder of the settings in the configuration enable/disable data collectors.

Developing Wintap

The following section provides a brief tutorial on common Wintap development activities.

Creating a new ETW Collector

A Wintap collector is responsible for interfacing with a Windows API and transforming the data it emits into a Wintap .NET data object and placing that object into the global Esper event stream. This can be accomplished as follows:

1. Extend the data model to support your new collector. Locate the WintapMessage.cs file under Collect\Shared\Models and create your new data class embedded within WintapMessage.
2. Create a Collector class in the gov.llnl.wintap.collectors namespace inheriting from EtwCollector. This is where you process your raw events. The name of your class should be the same as the ETW provider name (remove any dots or dashes) and be suffixed with the word Collector.

```
internal class MicrosoftWindowsCpuTriggerCollector : EtwCollector
{
```

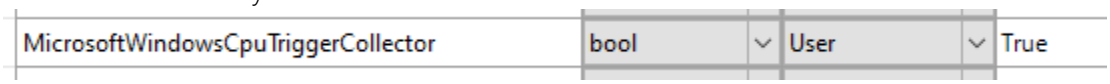
3. Set up your constructor. Set the CollectorName to the display name of your ETW provider, and set the EtwProviderId to the GUID of the provider.

```
public MicrosoftWindowsCpuTriggerCollector() : base()
{
    // For ETW events set source name here to be the Event Provider name
    this.CollectorName = "Microsoft.Windows.CpuTrigger";
    // this is the ETW Provider GUID, this what gets wired up with ETW
    this.EtwProviderId = "635d9d84-4106-4f3a-a5c2-7fda784ae6fc";
}
```

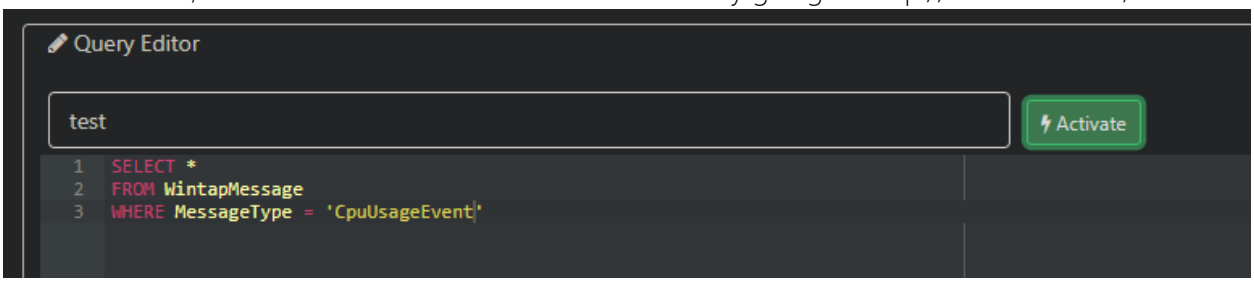
4. Implement the Process_Event method to transform the raw event into your data model object and place it on the global event stream.

```
public override void Process_Event(TraceEvent obj)
{
    // 1.)  Call base to setup rate monitoring
    base.Process_Event(obj);
    try
    {
        if(obj.EventName == "CpuUsageEvent")
        {
            // 2.)  Process the event
            Dictionary<string, string> parsedSessionEvent = parseEvent(obj.ToString());
            AppPerformanceMetricObject metric = new AppPerformanceMetricObject();
            this.CollectorName = obj.EventName; // override so that we can distinguish multiple events from the same provider.
            metric.OnBatteryPower = StateManager.OnBatteryPower;
            metric.UserBusy = StateManager.UserBusy;

            //  ...  do more stuff ...

            // 3.) Create a WintapMessage and attach your event to it
            WintapMessage wintapMsg = new WintapMessage(obj.TimeStamp, metric.PID, this.CollectorName);
            wintapMsg.AppPerformanceMetric = metric;

            // 4.) Send your event into the Wintap event pipeline
            wintapMsg.Send();
        }
    }
    catch (Exception ex)
    {
        WintapLogger.Log.Append("Error processing " + this.CollectorName + " event: " + ex.Message, LogLevel.Debug);

    }
```

5. Create a configuration entry for your new Collector. Right-click the project and select Properties to view the Settings screen. Add a new entry for your collector making sure to use the EXACT name as your class.



6. Test it. Compile and deploy your new Wintap.exe and Wintap.exe.config. Start Wintap and then use the Wintap Workbench to verify your new collector is working. By default, the Wintap Workbench is disabled. Modify your Wintap.exe.config (described above) to enable the Workbench, and then connect to it via a web browser by going to: http://localhost:8099/



Creating a standard Collector

The process for creating standard Collector (non-ETW) is pretty much the same as an ETW collector with these differences:

- Inherit from BaseCollector instead of EtwCollector
- Set only the CollectorName property in your Constructor
- Override the Start method and spin off a worker thread to gather/send event data from the source API (there is no Process_Event method in a standard Collector)

Notes on Developing the Workbench

- Workbench must be enabled in config before starting wintap
- The workbench exists as a compressed, embedded zip file within the Wintap binary.
- After editing any of the web source files, you must perform these steps to run and test your code:
    1. From the file system, delete or rename the existing Workbench.zip within your source code folder at \wintap\web
    2. Make a new Workbench.zip at \wintap\web by zipping the contents of \wintap\web\workbench
    3. Compile Wintap within Visual Studio
    4. Deploy your new Wintap.exe file and start wintap.
- Web dependencies can be installed/re-installed using NPM as follows:
    1. From an elevated command window, navigate to \wintap\web\workbench
    2. Run the following commands
        - npm install @coreui/icons –save
        - npm install flag-icon-css –save
        - npm install font-awesome –save
        - npm install simple-line-icons –save
- As of May 2021, all of the custom code for the workbench can be found in 3 files. Index.html at the root, and 2 javascript files at \wintap\web\workbench\scripts\workbench
- Swagger page at: http://localhost:8099/swagger

Extensibility

Wintap provides a set of programmatic interfaces which can be used to create plugins. The plugin infrastructure is provided by the Microsoft Extensibility Framework (MEF) found in `System.ComponentModel.Composition`. At minimum, a Wintap plugin is a compiled DLL file that implements one of the Wintap extensibility interfaces and is placed in %PROGRAMFILES%\Wintap\Plugins. Wintap will attempt to load all plugins in the Plugins directory upon service startup.

Here are the set of interfaces supported in Wintap version 6:

IRun

The `IRun` interface runs your code on a specified interval, much like a scheduled task or chron job. Getting small bits of code to run on a regular basis is a common requirement. Instrumenting recurring actions via Windows scheduled tasks can be brittle and error prone, and writing a dedicated Windows service is usually overkill. IRun provides a simple alternative for these scenarios. The run interval is specified by way of a TimeSpan object in the `RunManifest` object that is returned by the `RunStartup()` method (this example would tell Wintap to call the Run method once per day):

```
public RunManifest RunStartup()
{
    return new RunManifest { Interval = new TimeSpan(1, 0, 0, 0) };
}
```

Wintap will queue the plugin for execution at the next interval expiry. Upon expiry, Wintap will call your plugin's Run method:

```
public void Run()
{
    Logger.Get().Append("Run method called!  time to do stuff", LogVerboseLevel.Normal);
}
```

Wintap will impose a 2 minute time limit for the Run method to complete. If the plugin runs beyond 2 minutes, an exception will be raised (WATCH_DOG_TIMEOUT_EXCEEDED), and Wintap will be restarted.

ISubscribe

The `ISubscribe` interface delivers curated, modelled events to your Wintap plugin. Event data is delivered in the form of a WintapMessage C# data object. The Wintap message has at its root a set of attributes common to all event types (e.g. ReceiveTime, PID), and then a set of nested type specific objects. WintapMessage is a public object and becomes visible during development to plugin authors after referencing the Wintap assembly.

The events that a plugin will receive are specified in one of two ways; via the Wintap.exe.config file described earlier, or in code. To specify event providers in code, the plugin returns an EventFlags enum structure from its Startup method:

```
public EventFlags Startup()
{
    return EventFlags.FocusChange;
}
```

Multiple event providers can be specified as follows:

```
public EventFlags Startup()
{
    return EventFlags.FocusChange | EventFlags.Process | EventFlags.FileActivity;
}
```

The Subscribe method receives the stream of events at runtime. The plugin author can process events at their discretion from within the body of the Subscribe method, for example:

```
public void Subscribe(WintapMessage eventMsg)
{
    if(eventMsg.MessageType == "FocusChange")
    {
        Logger.Get().Append(eventMsg.FocusChange.OldProcessID + "->" + eventMsg.PID, LogVerboseLevel.Normal);
    }
}
```

ISubscribeEtw

The `ISubscribeEtw` interface delivers raw, unmodeled ETW event payloads to your Wintap plugin. This plugin interface allows authors to consume any ETW provider (of which there are 1000s on a typical machine) without having to implement a complete Collector (described earlier) which can be useful during early exploration and prototyping. To use this feature, a plugin author can simply specify the provider list they are interested in, and Wintap handles all of the ETW interactions on your behalf and returns the stream of events for the all of the providers you specify. To specify the ETW event providers of interest, you return a List<string> object from your Startup method as follows:

```
List<string> ISubscribeEtw.Startup()
{
    return new List<string>() { "Microsoft-Windows-Win32k" };
}
```

Any ETW provider name or ProviderGuid (in string form) can be used in your list. Like ISubscribe, ISubscribeEtw event data is delivered in the form of a WintapMessage C# data object, but the nested event data not strongly typed. Instead, the GenericMessage nested type is used for all event detail data. The raw ETW event is stringified and placed into the GenericMessage.Payload property as follows:

```
public void Subscribe(WintapMessage eventMsg)
{
    Logger.Get().Append(eventMsg.GenericMessage.Payload, LogVerboseLevel.Normal);
}
```

Example Plugin 1: Implementing all interfaces

All plugins require some ceremony to be loaded properly by Wintap. All plugins must export one of the above-described interface types, and they must expose two required metadata fields; Name and Description. Interface types are exported with the following class decoration:

```
[Export(typeof(%INTERFACE_NAME%))]
```

where %INTERFACE_NAME% is one of the supported Wintap interfaces.

Required metadata consists of two decorator fields: Name and Description:

```
[ExportMetadata("Name", "TestSensor")]
[ExportMetadata("Description", "A super simple example plugin.")]
```

The Name metadata field should match the assembly name of your plugin. The Description can be whatever you want. Here is a complete example demonstrating both interface exporting and metadata specification (next page). This first example demonstrates the implementation of all Wintap plugin interfaces in a single plug-in. In the next example, we will demonstrate the more typical pattern of implementing a single interface.

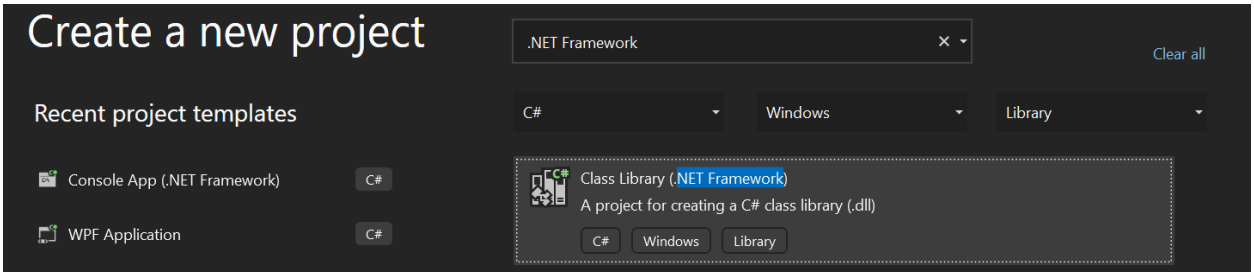
```
using gov.llnl.wintap;
using System.ComponentModel.Composition;
using static gov.llnl.wintap.Interfaces;
using gov.llnl.wintap.Models;

namespace TestSensor
{
    [Export(typeof(IRun))]
    [Export(typeof(ISubscribe))]
    [Export(typeof(ISubscribeEtw))]
    [ExportMetadata("Name", "TestSensor")]
    [ExportMetadata("Description", "A super simple example plugin.")]
    0 references | 0 changes | 0 authors, 0 changes
    public class TestSensor : ISubscribe, ISubscribeEtw, IRun
    {
        0 references | 0 changes | 0 authors, 0 changes
        public RunManifest RunStartup()
        {
            return new RunManifest { Interval = new TimeSpan(1, 0, 0, 0) };
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void Run()
        {
            Logger.Get().Append("Run method called!  time to do stuff", LogVerboseLevel.Normal);
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void RunShutdown()
        {
            // any special shutdown logic required by the IRun plugin
        }

        0 references | 0 changes | 0 authors, 0 changes
        public EventFlags Startup()
        {
            return EventFlags.FocusChange | EventFlags.Process | EventFlags.FileActivity;
        }

        0 references | 0 changes | 0 authors, 0 changes
        List<string> ISubscribeEtw.Startup()
        {
            return new List<string>() { "Microsoft-Windows-Win32k" };
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void Subscribe(WintapMessage eventMsg)
        {
            Logger.Get().Append(eventMsg.GenericMessage.Payload, LogVerboseLevel.Normal);
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void Shutdown()
        {
            // any required shutdown logic
        }
    }
}
```

Example Plugin 2: Implementing a response plugin

In this example we will demonstrate how to build a response plugin that reacts to specific events in the Wintap event stream. For this example, we will do something anytime we detect a new Notepad.exe process. To accomplish this, we will write a plugin implementing the ISubsribe interface and subscribe to Process events from Wintap. This example demonstrates creating the plugin using Visual Studio 2022, slight variations may be required if you are using a different version of Visual Studio.

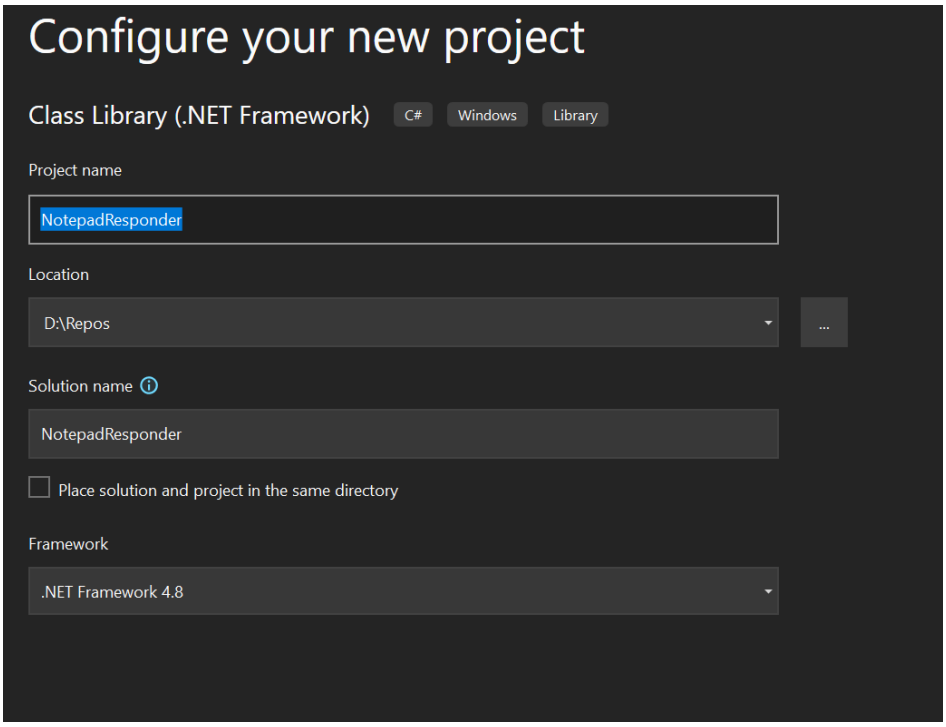
1. **Create your project in Visual Studio**

   Create a new project based upon the .NET Framework Class Library project template. From the Visual Studio "Create a new project" launch screen, filter the list of templates to show only C#, Windows, and Library types and then type "Class Library .NET Framework" into the search box. From the list of available templates, select "Class Library (.NET Framework)" as shown here:

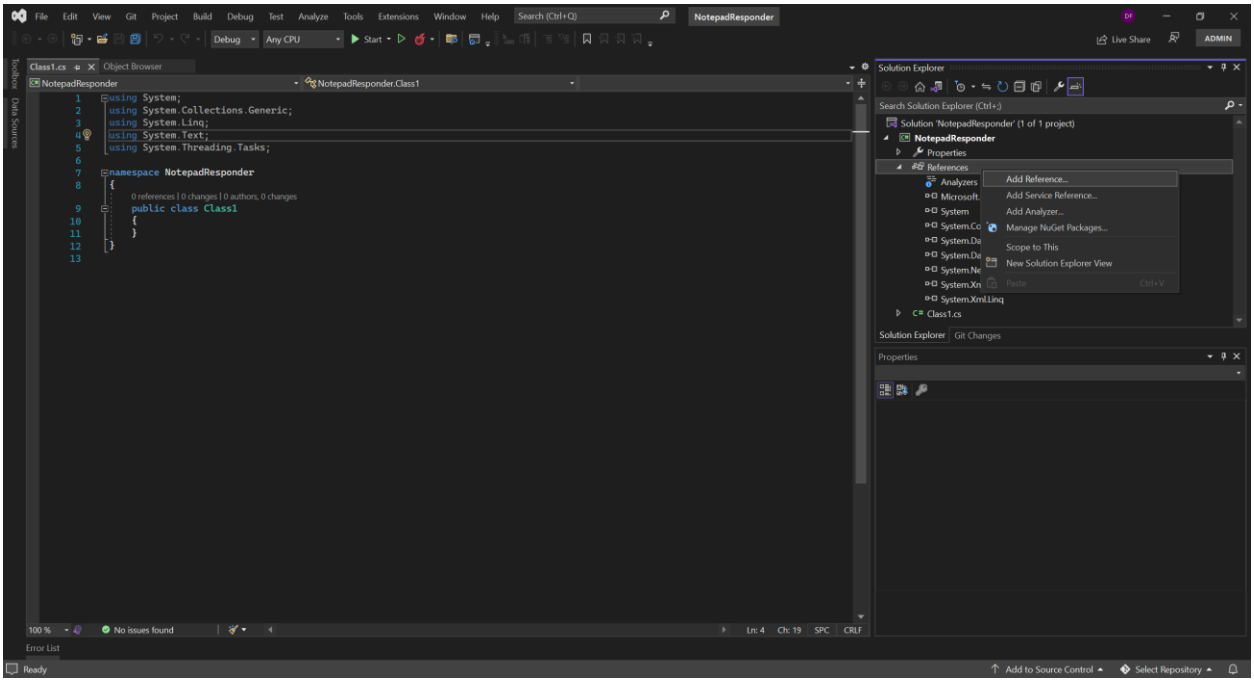


2. **Configure your new project**

   Fill in the fields to provide a name and file location as appropriate. Select the .NET Framework 4.8 in the Framework selector.

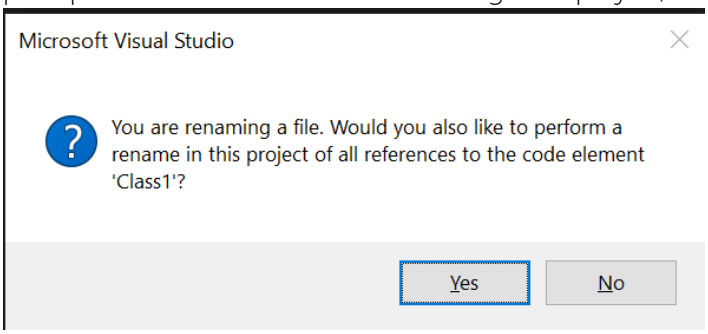


3. **Add a reference to the Wintap assembly.**
   From the Solution Explorer window, right click "References" and select "Add Reference" as shown here:



   Hit the browse button and locate the Wintap.exe file located on your computer.

4. **Name your class.** In this example, we will rename our class to match our project name. To do this, right-click the Class1.cs file in your Solution Explorer view (right window, just under References) and select Rename. Type "NotepadResponder" and hit Enter. You should be prompted to cascade the rename through the project, select YES:



5. **Add the required Wintap namespace reference.** Append the following line to the list of "Using" statements at the top of your code file:

```
using static gov.llnl.wintap.Interfaces;
```

6. **Implement the interface.** First, modify your class declaration as follows:

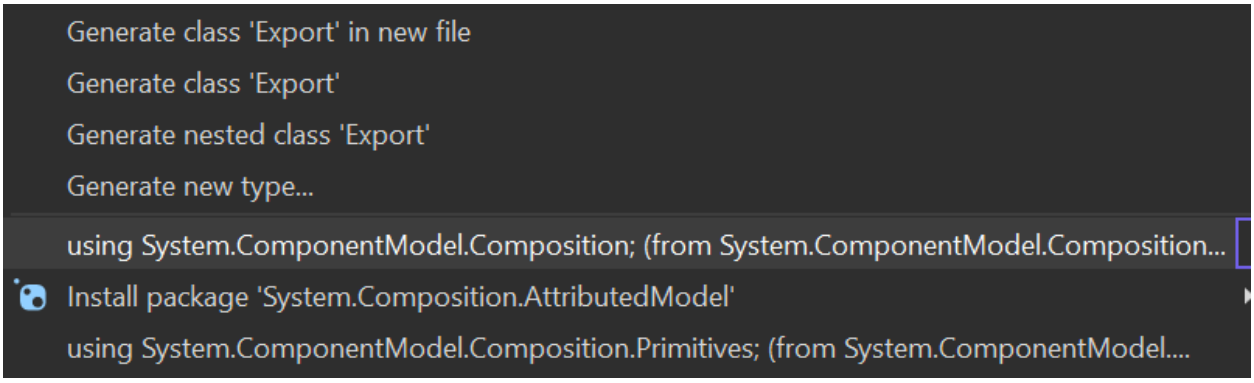
```
public class NotepadResponder : ISubscribe
{
}
```

7. Add class decorators. Add the following lines just above your class definition:

```
[Export(typeof(ISubscribe))]
```

```
[ExportMetadata("Name", "NotepadResponder")]
[ExportMetadata("Description", "Example responder plugin")]
```

8. Add extensibility framework support. Here, we need to bring in references for the Microsoft Extensibility Framework. Easiest way to do this is to right-click the word Export in the first line we just added and select "Quick Actions and Refactorings", then in the subsequent context menu that appears, select the fifth line down: "Using System.ComponentModel.Composition (from …) as shown here:



9. Refactor. Next, right-click the ISubscribe text (should have a red line underneath it) and select "Quick Actions and Refactorings", then select "Implement Interface". The following code changes should automatically populate:

```
namespace NotepadResponder
{
    0 references | 0 changes | 0 authors, 0 changes
    public class NotepadResponder : ISubscribe
    {
        0 references | 0 changes | 0 authors, 0 changes
        public void Shutdown()
        {
            throw new NotImplementedException();
        }

        0 references | 0 changes | 0 authors, 0 changes
        public EventFlags Startup()
        {
            throw new NotImplementedException();
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void Subscribe(WintapMessage eventMsg)
        {
            throw new NotImplementedException();
        }
    }
}
```

10. **Subscribe to Process events.** Modify your Startup method as follows:

```
public EventFlags Startup()
{
    return EventFlags.Process;
}
```

11. **Remove the throw statement** in your Shutdown method and leave the method empty. In a real plugin you may want to implement some behavior here, but for this example we do not need any.
12. **Implement Subscribe.** This is where the real action lives. The code written here will be called every time a subscribed event occurs on the local machine. In this example, every Process START or TERMINATE event.

```
public void Subscribe(WintapMessage eventMsg)
{
    if(eventMsg.MessageType == "Process" && eventMsg.ActivityType == "Start")
    {
        if(eventMsg.Process.Name == "notepad.exe")
        {
            // do something...
        }
    }
}
```

13. **Compile and test.** Right-Click your project in Solution Explorer and select Build. The code should compile without error. Once built, find NotepadResponder.dll from your bin\debug folder of your project's file system directory and copy it to your local machine's plugin directory, for example: C:\Program Files\Wintap\Plugins. Then restart the Wintap service and your plugin should load and execute.

For sake of completeness, here is the full class implementation of our new response plugin:

```
using gov.llnl.wintap.collect.models;
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static gov.llnl.wintap.Interfaces;

namespace NotepadResponder
{
    [Export(typeof(ISubscribe))]
    [ExportMetadata("Name", "NotepadResponder")]
    [ExportMetadata("Description", "Example responder plugin")]
    0 references | 0 changes | 0 authors, 0 changes
    public class NotepadResponder : ISubscribe
    {
        0 references | 0 changes | 0 authors, 0 changes
        public void Shutdown()
        {

        }

        0 references | 0 changes | 0 authors, 0 changes
        public EventFlags Startup()
        {
            return EventFlags.Process;
        }

        0 references | 0 changes | 0 authors, 0 changes
        public void Subscribe(WintapMessage eventMsg)
        {
            if(eventMsg.MessageType == "Process" && eventMsg.ActivityType == "Start")
            {
                if(eventMsg.Process.Name == "notepad.exe")
                {
                    // do something...
                }
            }
        }
    }
}
```