**DotNetVault Quick Start Guide – Visual Studio 2019**

1. Open visual studio and Create a new .NET Core 3.0+ or .NET Framework 4.8 Console Application.
2. **If you chose .NET Framework 4.8**, open the .csproj file and add the following Line <LangVersion>8.0</LangVersion> under each “Platform/Config” PropertyGroup as shown in highlight below. This is unnecessary if you chose a .NET Core 3.0+ based Console Application.

<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">

<PlatformTarget>AnyCPU</PlatformTarget>

<DebugSymbols>true</DebugSymbols>

<DebugType>full</DebugType>

<Optimize>false</Optimize>

<OutputPath>bin\Debug\</OutputPath>

<DefineConstants>DEBUG;TRACE</DefineConstants>

<ErrorReport>prompt</ErrorReport>

<LangVersion>8.0</LangVersion>

<WarningLevel>4</WarningLevel>

</PropertyGroup>

<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">

<PlatformTarget>AnyCPU</PlatformTarget>

<DebugType>pdbonly</DebugType>

<Optimize>true</Optimize>

<OutputPath>bin\Release\</OutputPath>

<DefineConstants>TRACE</DefineConstants>

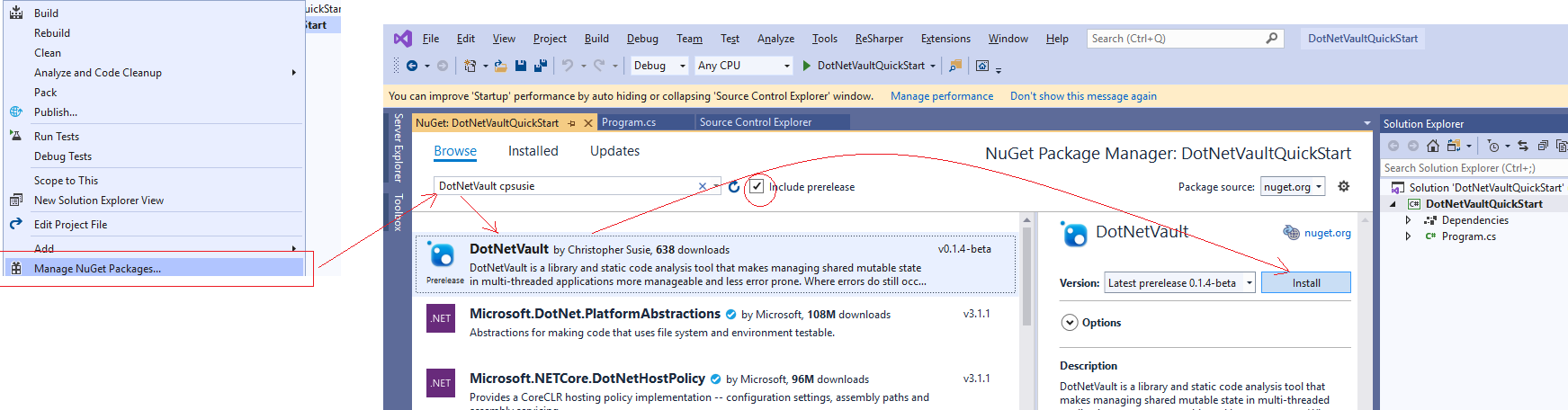
<ErrorReport>prompt</ErrorReport>

<LangVersion>8.0</LangVersion>

<WarningLevel>4</WarningLevel>

</PropertyGroup>

1. Right click on your project and chose “Manage NuGet packages”. Click on the Browse option, select “Include Prerelease”, enter “cpsusie dotnetvault” into the search block, select DotNetVault, then click “Install”



1. You are now set up to use DotNetVault in your project.
   1. To ensure that the static analyzer installed correctly, delete the “Hello World” program entered in by default and replace it with the following code:

using System;

using System.Threading;

using DotNetVault.Vaults;

namespace LinuxDotNetVaultSetup

{

class Program

{

static void Main(string[] args)

{

var strVault = new BasicVault<string>(string.Empty);

Thread t1 = new Thread(() =>

{

Thread.SpinWait(50000);

var lck = strVault.SpinLock();

lck.Value += "Hello from thread 1, DotNetVault! ";

});

Thread t2 = new Thread(() =>

{

using var lck = strVault.SpinLock();

lck.Value += "Hello from thread 2, DotNetVault! ";

});

t1.Start();

t2.Start();

t2.Join();

t1.Join();

string finalResult = strVault.CopyCurrentValue(TimeSpan.FromMilliseconds(100));

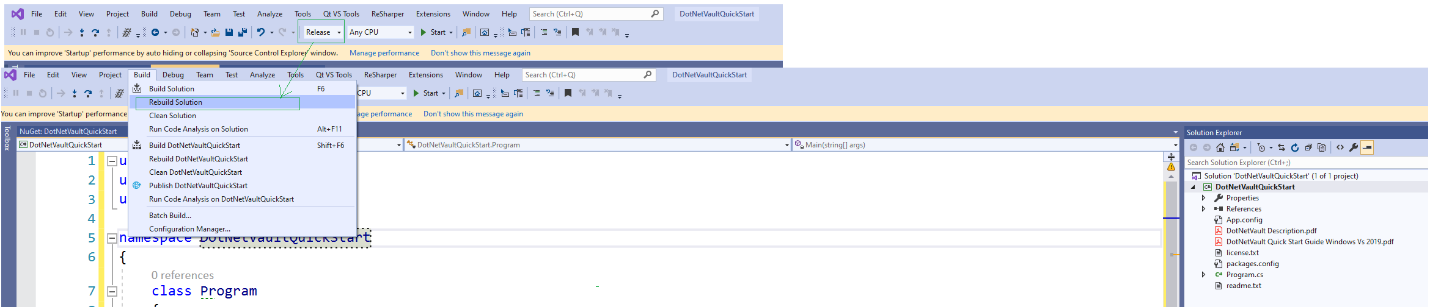
Console.WriteLine(finalResult);

}

}

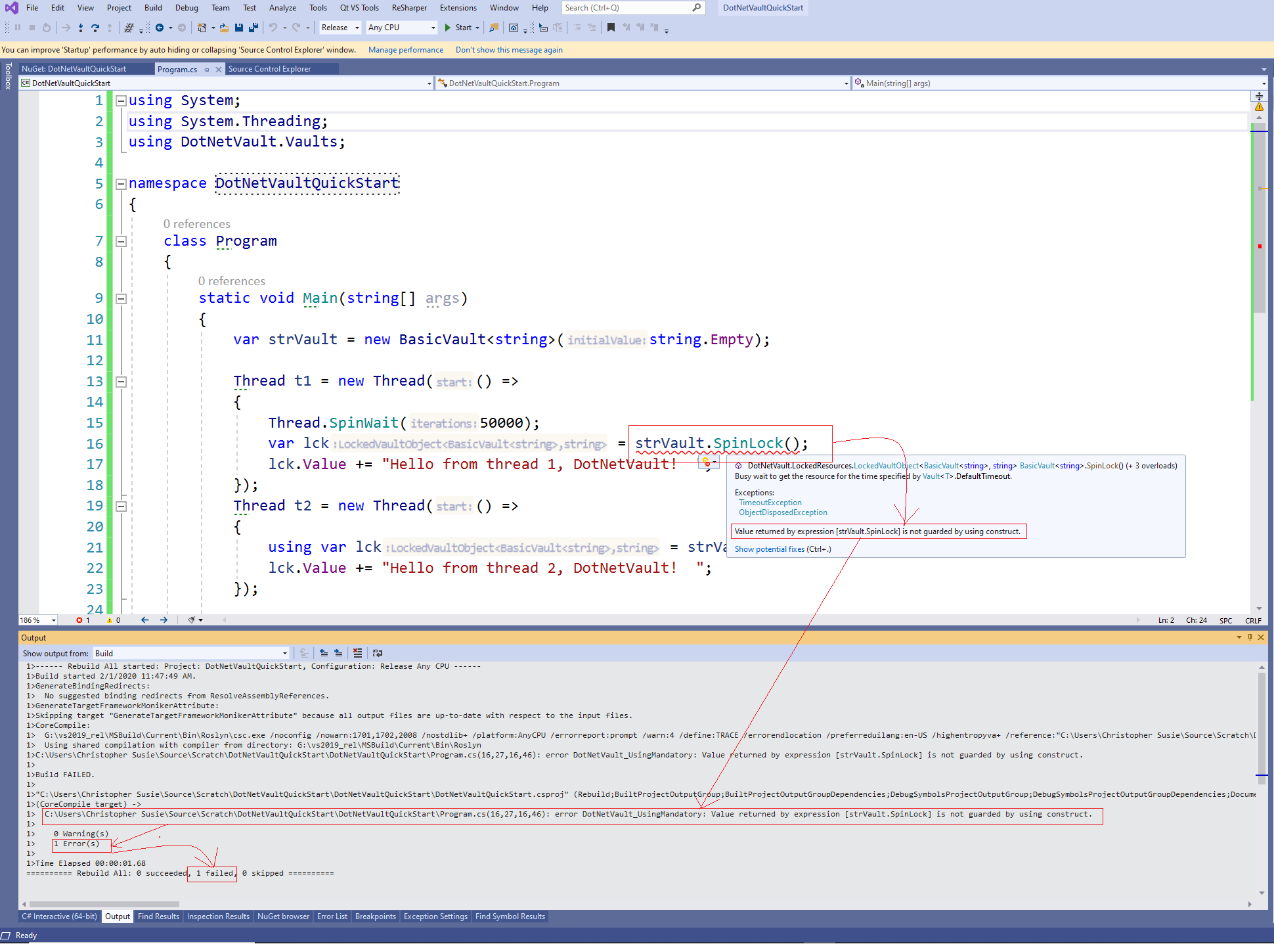
}

* 1. Change the Configuration to “Release” and Build the project:

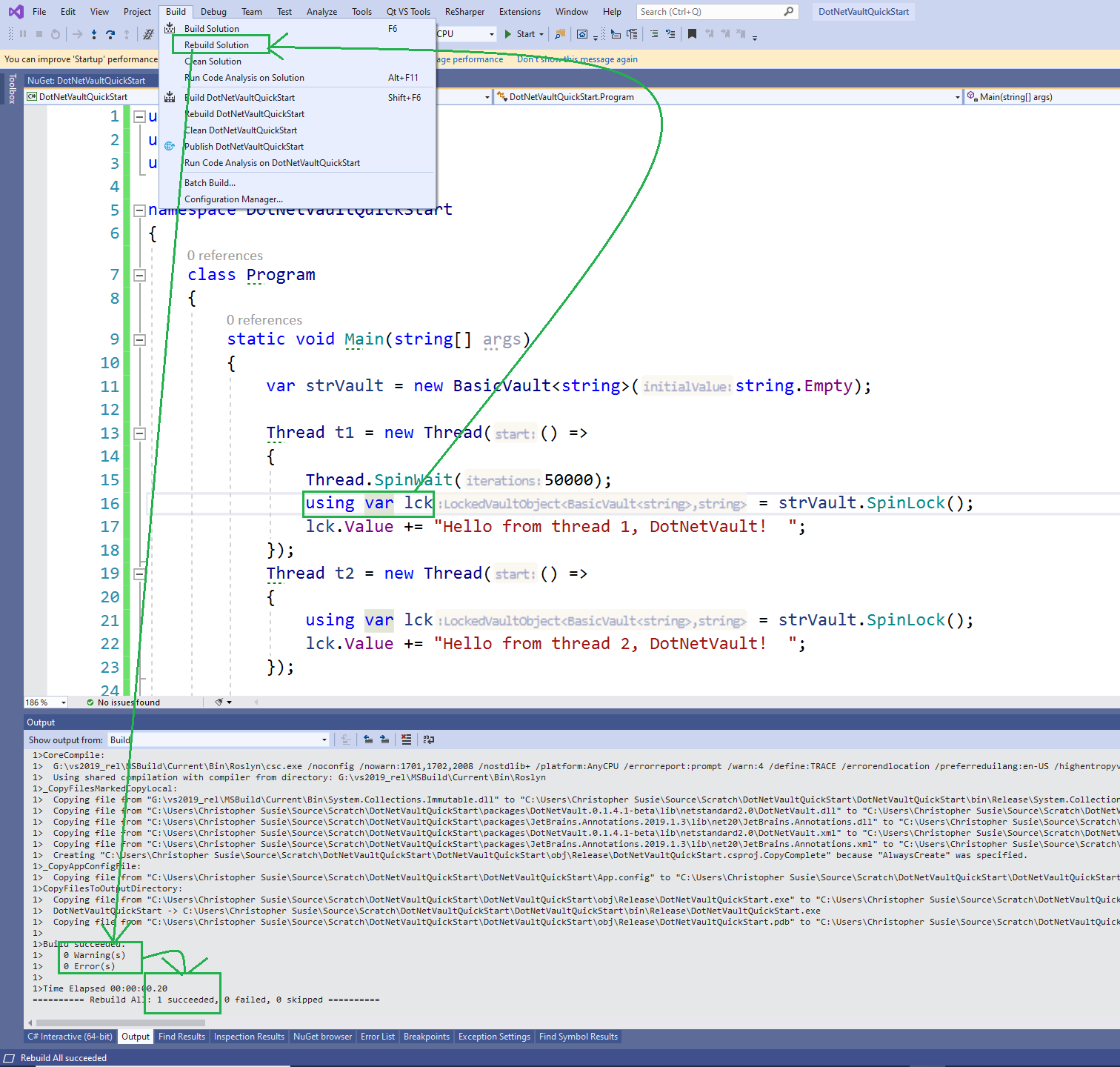


* 1. If you have installed everything correctly, **the build should fail**.

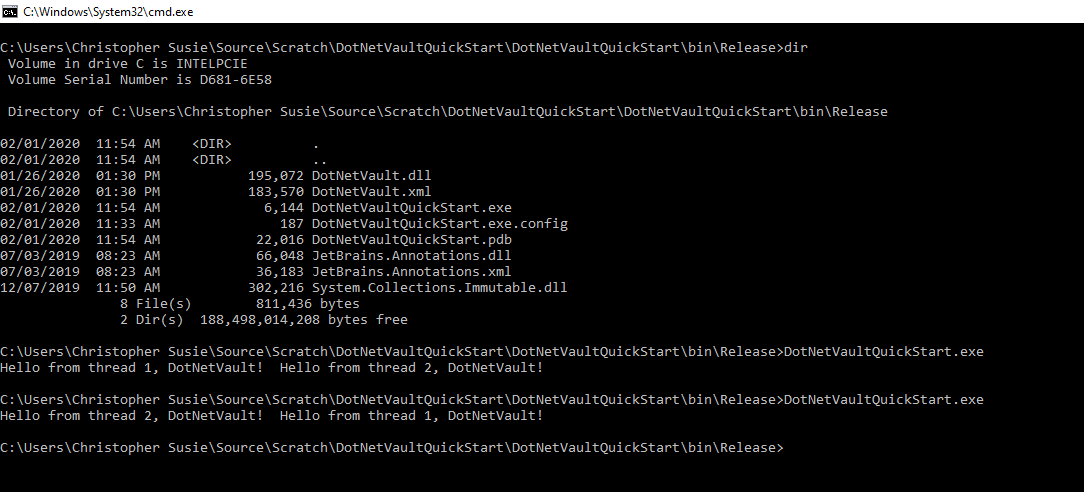
If it builds, consult VisualStudio documentation for how to enable Roslyn Analyzers. Do not attempt to use this library without static analysis enabled. Assuming it does not build, you should see the following as a result of your build attempt:



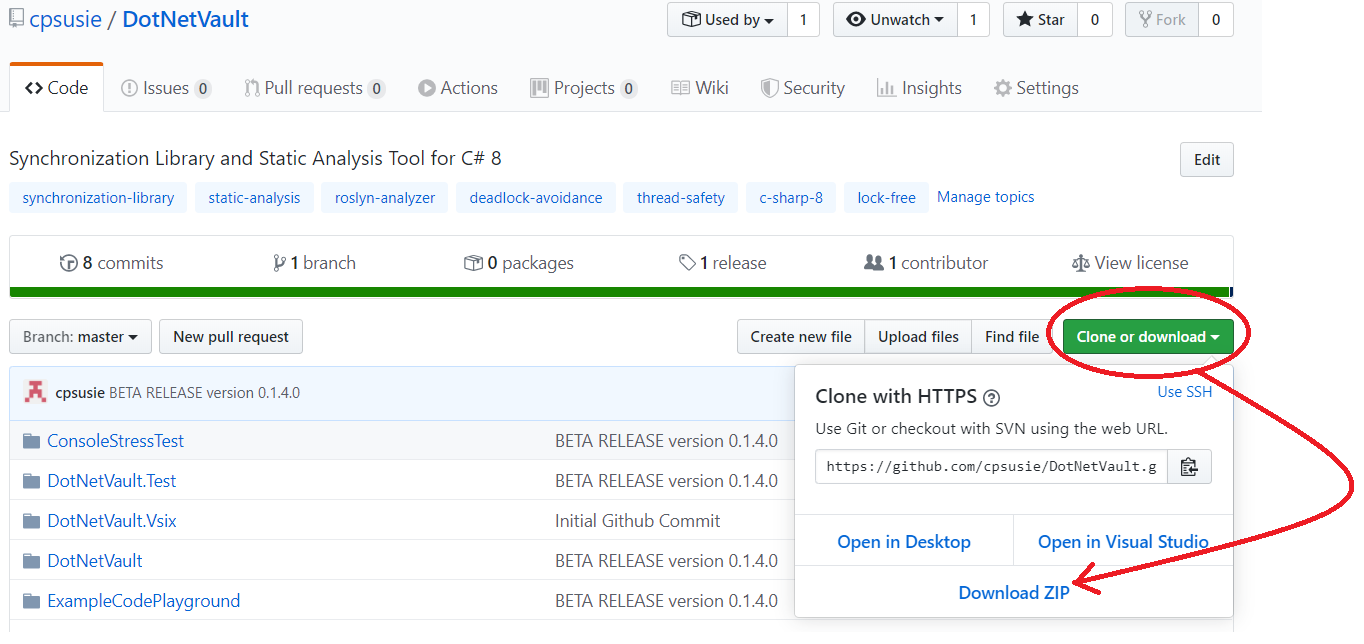
* 1. The error is that you **must** guard the return value from a Lock() or SpinLock() method (or any other method whose return value you choose to annotate with the *UsingMandatory* attribute)with a using statement or declaration. Failure to ensure that the lock is promptly released would cause a serious error in your program … it would timeout whenever in the future you attempted to obtain the lock.
  2. To fix the error, on line 16, change “var lck =…” to “using var lck =…” as shown then Build again. This time, the build should succeed as shown:



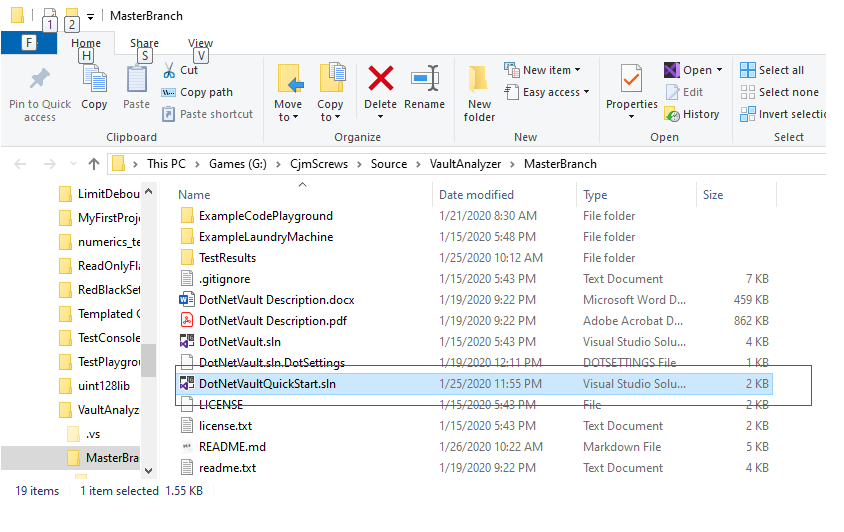
* 1. Now you should be able to run the application as shown:

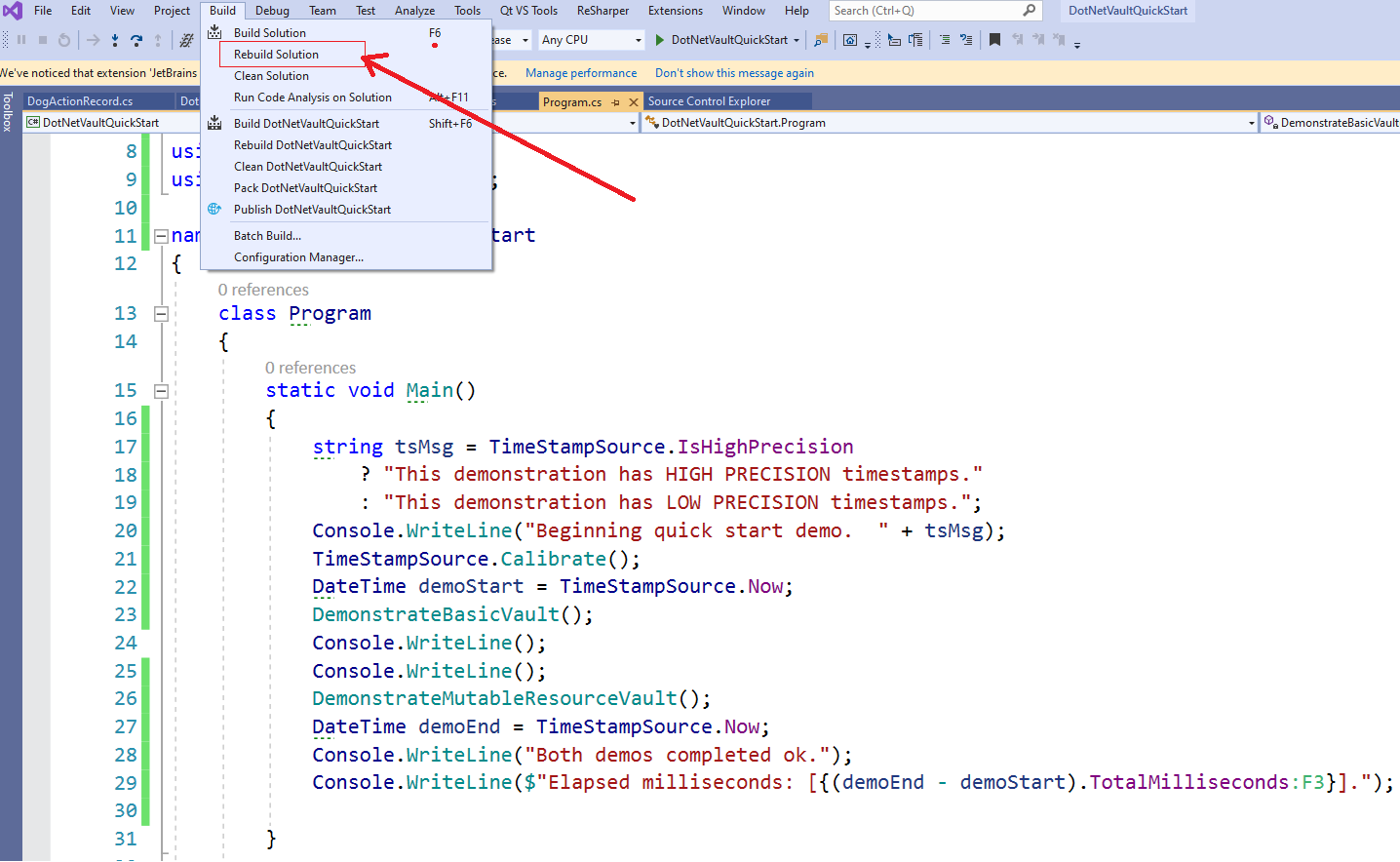


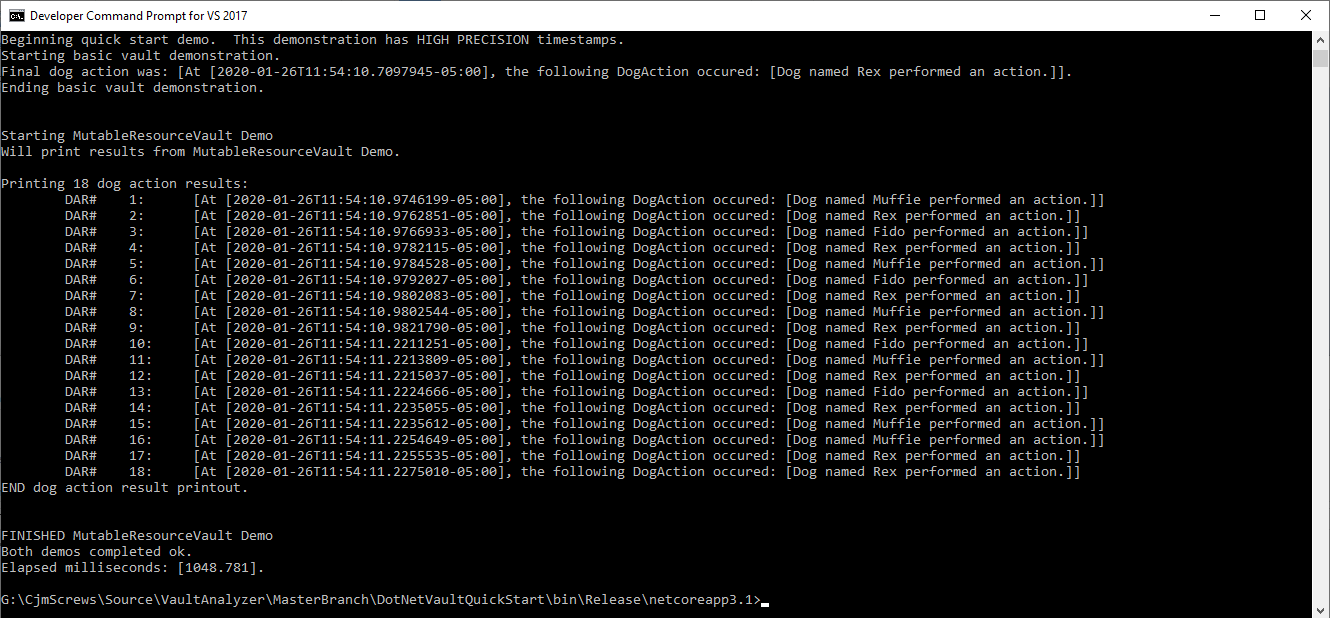
1. Congratulations, you have successfully built an application using DotNetVault. Next, we will look at a QuickStart application that shows the very basics of what can be done with this library and its integrated static analyzer.
   1. download the source code for DotNetVault from GitHub.



* 1. Extract the zip file contents to a folder in a convenient place
  2. Go into the folder structure and open the file “DotNetVaultQuickStart.sln” using Visual Studio 2019



* 1. Build the solution.
  2. Run the solution and check output, which should be something like this:



* 1. As you can see, from Main(), the demo demonstrates the use of the BasicVault, which stores VaultSafe objects and the MutableResourceVault which stores objects that are not VaultSafe.
     1. VaultSafe objects are ones that do not require effort to keep isolated. They include
        1. Unmanaged value types (e.g. long, enums, DateTime, TimeSpan, etc.)
        2. Sealed immutable reference types that are annotated with the VaultSafe attribute (string is automatically considered vault-safe; Immutable collections from System.Collections.Immutable that have only vault-safe type arguments are also considered vault-safe automatically).
        3. Value types that are annotated with the VaultSafe attribute and contain only other types that comply with #1, 2 and 3
        4. Unmanaged value types, strings and qualifying immutable collections are considered VaultSafe without need for annotation with the VaultSafe attribute.

These objects are easy to isolate because copies of them are either true deep copies (unmanaged types), totally immutable reference types (no danger of a stored reference changing the protected value), or value types that contain only types that are unmanaged value types and immutable reference types. The resource protected in the DemonstrateBasicVault() in *Program.cs* is a *DogActionRecord*, found in DogActionRecord.cs.

DogActionRecord is VaultSafe because:

1. It is annotated with the VaultSafe attribute
2. It is a value type with field members that include
   1. An unmanaged value type (DateTime) and
   2. A sealed immutable reference type (string)
   3. The DemonstrateBasicVault() in *Program.cs* creates a BasicVault<DogActionRecord>, a vault that protects a DogActionRecord and can be used to obtain locks for synchronized access to the value. It then creates a List of Dogs (Dog.cs) then calls their DoDogActions() methods, which causes each to spawn a thread that obtains a lock on the vault and overwrites the value stored therein with an action.

The syntax is straight forward:



Figure 1

At the end of the demonstration, the main thread obtains the lock and prints out the *DogActionRecord* that happens to be there (because the order is non-deterministic, there will be different results):



Figure 2

* 1. The DemonstrateMutableResourceVault() method in Program.cs shows protection of a resource that IS NOT vault-safe. It is a SortedSet of DogActionRecords. The dogs in this demo, instead of overwriting a single DogActionRecord, will add their DogActionRecords to the SortedSet (which maintains them ordered by timestamp, convenient for printing in order at end).

Resources that are not VaultSafe are protected by a MutableResourceVault, the lock objects of which are more restrictive because they need to make sure that only VaultSafe types are passed into the protected resource or received out from the protected resource; otherwise, mutable state (to which a reference may exist outside) could mingle with the protected resource or mutable state inside the protected resource could leak to the outside and cause unsynchronized access to the protected resource.

The syntax for creating a MutableResourceVault is shown:



Figure 3

Accessing the mutable resource through the lock is mediated by delegates:



Figure 4

At the end of the simulation, a lock is acquired that returns the SortedSet to the main thread as an ImmutableSortedSet<DogActionRecord> as shown:



Figure 5

* 1. Further resources.
     1. The DotNetVault itself comes with an example of a vault customized to protect StringBuilder resources without resort to the inconvenient delegate syntax. It contains directions for making your own customized vaults and locked resource objects.
     2. Included in the source repository is the ConsoleStressTest using .NET Core 3.1. It can be run in a Linux or Windows environment and demonstrates the effectiveness of DotNetVault at protecting a mutable resource with a high degree of thread contention.
     3. The LaundryMachine.sln and the LaundryStressTest project therein requires Windows because it uses WPF. It demonstrates the usage of many vaults and many threads in contention in a highly (unnecessarily so, but good to demonstrate DotNetVault’s usefulness even in convoluted scenarios) complex multithreaded state machine scenario where LaundryMachines have their own threads and loader and unloader robots (also with their own threads) contend for access to laundry machine. The simulation runs until all soiled articles are cleaned. The loader robots put dirty laundry into one of the machines then start the cycle, the unloader robots take dirty laundry from the machine and put them in the clean bin. The robots constantly contend with each other for access to the machine and since each Laundry Machines state machine thread is independent, the robots also contend with the state machine threads as well as each other to access the machines.
     4. The ExampleCodePlayground project is used to get a feel for the static analyzer’s rules and how they work.
     5. There is a unit test project primarily oriented around ensuring that the static analyzer rules work properly.
  2. The DotNetVault Description.pdf provides detailed information on the DotNetVault, the vaults and LockedResourceObjects, its reasoning and static analysis rules as well as examples.