Forensic Scenario Bot

Technical Manual

Contents

Requirements

Operating System: Windows 7, Windows 8, Windows 8.1, Windows 10

Processor: 1GHz x86 or x64 processor.

RAM: 2GB RAM.

Additional Software:

- TrueCrypt: Optional
- Nmap: Optional

All additional software are required for some of the Bot functions. Without the required software, the Bot will continue to work but the relevant functions will be disabled.

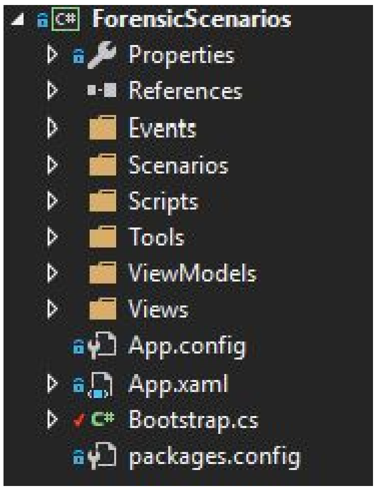
Permissions: User must have permission to create files\folders and browse the internet for some functions to run.

Introduction

This tool is built using the Windows Presentation Foundation (WPF), following the Model-View-ViewModel (MVVM) pattern, and the Caliburn Micro framework. The MVVM pattern allows for separation of concerns and provides increased scalability and maintainability (for more information, refer to Apendix A - 1.). It makes possible the realization of the main design goal of the tool - adding new scenarios with as little effort as possible. With that said, at least basic knowledge of WPF and the MVVM pattern is beneficial when performing such task. Naturally, extending the tool' core functionality is also made easier thanks to the MVVM pattern, however deeper knowledge of Caliburn Micro is required. Because of this, the manual is separated in two parts: basic (adding new scenarios) and advanced (extending/changing functionality).

For more information about the UI and existing scenarios, refer to the User Manual.

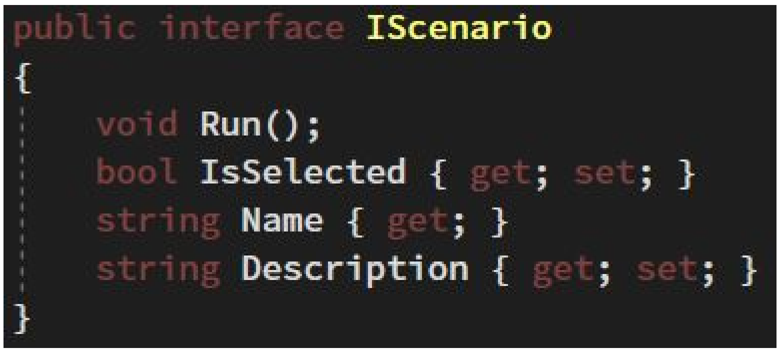
Project structure



Starting with the **"ViewModels"** folder, it contains all the view models used in the tool. In MVVM, a view model is responsible for handling the view logic and requesting data from models (if any), which it then passes to the view. Following this logic, the **"Views"** folder contains all the views for the tool. In this case, the views contain only XAML and no code behind them as Caliburn Micro is responsible for linking the views to the view models (see Appendix A - 3.). The **"App.xaml"** is where all the custom styles are located. The **"Scenarios"** folder is where all the scenario implementations are. They also act as the "model" part in MVVM. The **"Events"** folder contains classes which represent different events and carry data with them. They are all to be used by the event aggregator. A closer look is provided in the basic section. Next, the **"Tools"** folder contains helper classes and services. The **"Scripts"** folder contains scripts and tools used by some of the scenarios. This is out of the scope of this manual and is not used in the tool implementation. Finally the **"Bootstrap.cs"** class is where Caliburn Micro is configured. It also contains a dependency injection (DI) container, since the framework utilizes the Inversion of Control (IoC) pattern, which provides instances of classes or services (dependencies) to other classes which "depend" on them. This means that all view model and model classes which use other view model or model classes should be registered. Classes and services are registered inside the "Configure" method. With this pattern, dependency classes should never be instantiated inside the constructor, but simply provided as a parameter. The container will provide an instance at runtime (see Appendix A - 4.). The other important method is "OnStartup". This is where Caliburn Micro takes over. Inside it, an entry window should be specified. This is done by calling the “DisplayRootViewFor” method and providing a view model type as a generic parameter.

Basic extension

In this section everything about adding a new forensic scenario is covered. In order to be integrated, every scenario has to inherit from the **“PropertyChangedBase”** class (provided by Caliburn Micro) and implement the **“IScenario”** interface - this is how all scenarios are represented in the tool’ implementation.



```
public interface IScenario
{
    void Run();
    bool IsSelected { get; set; }
    string Name { get; }
    string Description { get; set; }
}
```

The interface requires three properties and one method to be implemented. The “IsSelected” property is used by the view model, so no customization is needed. The “Name” property is what is used to display the name of the scenario in the UI. The “Description” provided is displayed in a text box in the UI when a scenario is hovered on. The most important member of the interface is the “Run” method, which is called when the scenario is to be executed. All the code which is required to successfully run a scenario should be in there. Note that this method is called asynchronously, so the UI is not blocked during execution. After that, in order to integrate the scenario with the tool a few things are required. But first, a quick introduction of the event aggregator.

The **event aggregator** in Caliburn Micro is a service which allows for messages (data) to be published between objects. When an object publishes a message, all subscribers to the event aggregator will be notified and those who know how to handle the particular kind of message will do so (refer to Appendix A - 5.). This is what is used to notify the view model that a scenario has finished execution and also to receive status messages, such as if a file has been successfully created. These two messages are represented by the **“ScenarioCompleted"** and **“ScenarioStatusUpdated”** classes in the “Events” folder. The former has to be ***always*** sent during execution. This doesn’t necessarily have to happen in the “Run” method (see the encryption scenarios). *If not, a scenario will run forever and browsing, selecting and executing new scenarios will be disabled in the UI.* The latter is optional, however is user friendly and is used to let the user know what is going on and if there are any errors during execution. These status messages will be displayed inside the scenario output textbox in the UI. To send a message the **“BeginPublishOnUIThread”** method should be called using the event aggregator and an instance of the message type should be passed as a parameter. Since some scenarios might just create some folders and files and move them around while others might require the user to do something and then provide an input, the time it takes to complete each scenario will greatly differ. With this design choice, such situations are accounted for.

To make a scenario browsable and selectable inside the UI, it needs to be added as a dependency (constructor parameter) to one of the view models which inherit from **“ScenarioCategoryViewModel”.** Each of them represents a category tab in the UI. So for example, if it is an encryption scenario, it should be added to the “EncryptionScenarioViewModel” class.

However, if a new category is desired, a new view model should be created which ***has to*** inherit from **“ScenarioCategoryViewModel”.** This, should then be added as a dependency to the **“MainWindowViewModel”.** The last step is, to let the IoC container know about the new scenario so it can inject it as a dependency. This done inside the **“Configure”** method in the **“Bootstrap”** class. Note that, if a new category view model was created it should be added too.

To summarize the above in 4 easy to follow steps:

1. Implement the new scenario logic and then make sure it inherits from **“PropertyChangedBase”** and **“IScenario”**
2. Ensure that the **“BeginPublishOnUIThread”** method is called with an instance of the **“ScenarioCompleted"** class
3. Add the scenario as a dependency to a view model which inherits from **“ScenarioCategoryViewModel”**
    a. Optionally, create a new view model for this, to display it in a new category
4. Add the scenario (and the new view model if created) to the DI container inside the **“Configure”** method, located in the **“Bootstrap”** class

A good starting point, is to see the implementation of the existing scenarios.

Advanced extension

In this section some guidelines are provided for extending the tool’ functionality. As mentioned before, a strong knowledge of MVVM and Caliburn Micro are required. The best way to do this, is to breakdown some points of interest in the **“MainWindowViewModel”** class and how it is connected to its view with the help of Caliburn Micro.

Starting from the class definition, the “MainWindowViewModel” inherits from a class provided by the framework (see Appendix A - 6.), which signifies that it holds a collection of objects and only one is active at the time. The collection in this case is scenario categories, which are represented by the view models, mentioned in the previous section. The two, **“IHandle”** generic interfaces require only one method - “Handle” to be implemented, taking the generic type specified as a parameter. This is what signifies to the event aggregator that this class knows how to handle a particular type of message. The **“SelectedScenarios”** property holds and observable collection, which contains the scenarios selected by the user for execution. Items are added in the **“SelectionChanged”** method, which is called in when the selection has changed in the list view.

This is a good moment, to showcase the ways which Caliburn Micro allows methods and properties to be connected to events and bindings in XAML. It offers a method called “Message.Attach” which takes a method name string, corresponding to the method name in the view model. This can be done multiple times, for different events. The syntax required for this is as following: [Event EventName] = [Action MethodName] , where EventName is the name of the event and MethodName is the name of the method in the view model. To add multiple handlers, each handler should be separated by a semicolon. Additionally, a method parameter can be specified in brackets, following the method name (see Appendix A - 7.). Examples of this can be found in **“MainWindowView”**, located inside the **“Views”** folder. To add a guard for whether a method can be executed, a boolean property with the method name which the guard is for, preceded by “Can” can be added to the view model. One such example in the view model is the **“CanRunScenarios”** property which guards the **“RunScenarios”** method. When the property is false, the button will be disabled in the UI. The rest of the properties and methods are bound to other elements in the XAML.

The **“Tools”** folder contains two classes of interest. The first is the **“TabContent”** which makes possible browsing through different tabs without losing state when navigating back to them. The code for this is provided by Ivan Krivyakov in a Code Project tutorial (see Appendix A - 8.). The other is the **“ProcessService”** service, which is used to keep track of the running processes spawned by the tool and makes sure that they are closed when the program shuts down. It should be used to start processes which stay active for a long time. This is used with the reverse shell scenario, for example, to ensure that no malicious processes stay active after the tool is closed.

Appendix A

Useful links

1. The MVVM Pattern
2. Caliburn Micro
3. Caliburn - Naming Conventions
4. Caliburn - IoC Container
5. Caliburn - Event Aggregator
6. Caliburn - Screens, Conductors and Composition
7. Caliburn - Actions
8. WPF TabControl: Turning Off Tab Virtualization