Analysis Report for: 21FAC19FAA57FF7ED00C7046CF1249F7.cs

Overall Functionality

This C# code implements a console application that appears to be designed to execute a PowerShell script (`postApiCoyote1h_V1.ps1`) embedded as a resource within the application. The application handles command-line arguments, checks for input/output redirection, and provides a custom PowerShell host implementation to manage console interaction and error handling. The PowerShell script itself is not provided, but the application suggests it's intended for interacting with an API (possibly a REST API given the name "postApi"). The `-extract` option allows the user to extract the PowerShell script to a file. The application uses a custom UI to handle console output and input, enabling features like custom colors and secure string input for credentials.

Function Summaries

* **`Console_Info.GetStdHandle(Console_Info.STDHandle stdHandle)`:** A P/Invoke method that gets a handle to the standard input, output, or error stream. It takes an enum representing the handle type (`STDHandle`) and returns a `UIntPtr` handle.

* **`Console_Info.GetFileType(UIntPtr hFile)`:** A P/Invoke method that retrieves the type of a file handle. It takes a `UIntPtr` file handle and returns a `FileType` enum.

* **`Console_Info.IsInputRedirected()`:** Checks if standard input is redirected. Returns `true` if input is redirected (not a character device), `false` otherwise.

* **`Console_Info.IsOutputRedirected()`:** Checks if standard output is redirected. Returns `true` if output is redirected (not a character device), `false` otherwise.

* **`Console_Info.IsErrorRedirected()`:** Checks if standard error is redirected. Returns `true` if error output is redirected (not a character device), `false` otherwise.

* **`MainApp.Main(string[] args)`:** The main entry point of the application. Parses command-line arguments, initializes the PowerShell runspace and executes the embedded PowerShell script. It handles exceptions, provides a wait option, and sets the application's exit code.

* **`MainApp.CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)`:** An event handler for unhandled exceptions in the application domain. It throws a new exception containing information about the unhandled exception.

* **`MainModule.MainModule(MainAppInterface app, MainModuleUI ui)`:** Constructor for the custom PowerShell host. Initializes the host with references to the application interface and UI objects.

* **`MainModule.PrivateData`:** Gets the private data for the PowerShell host. In this case, it returns a `PSObject` representing a `ConsoleColorProxy` to allow the PowerShell script to interact with console colors.

* **`MainModule.CurrentCulture`, `MainModule.CurrentUICulture`, `MainModule.InstanceId`, `MainModule.Name`, `MainModule.UI`, `MainModule.Version`:** Override methods for the `PSHost` interface, providing information about the custom host.

* **`MainModule.EnterNestedPrompt()`, `MainModule.ExitNestedPrompt()`, `MainModule.NotifyBeginApplication()`, `MainModule.NotifyEndApplication()`:** Placeholder methods for the `PSHost` interface that are not implemented in this class.

* **`MainModule.SetShouldExit(int exitCode)`:** Sets the application's exit flag and code.

* **`MainModule.ConsoleColorProxy`:** A nested class which acts as a proxy to set console colors from the powershell script.

* **`MainModuleRawUI.ReadConsoleOutput`, `MainModuleRawUI.WriteConsoleOutput`, `MainModuleRawUI.ScrollConsoleScreenBuffer`, `MainModuleRawUI.GetStdHandle`:** P/Invoke methods that interact directly with the Windows console API. Used for advanced console manipulation.

* **`MainModuleRawUI.BackgroundColor`, `MainModuleRawUI.BufferSize`, `MainModuleRawUI.CursorPosition`, `MainModuleRawUI.CursorSize`, `MainModuleRawUI.ForegroundColor`, `MainModuleRawUI.GetBufferContents`, `MainModuleRawUI.KeyAvailable`, `MainModuleRawUI.MaxPhysicalWindowSize`, `MainModuleRawUI.MaxWindowSize`, `MainModuleRawUI.ReadKey`, `MainModuleRawUI.ScrollBufferContents`, `MainModuleRawUI.SetBufferContents`, `MainModuleRawUI.WindowPosition`, `MainModuleRawUI.WindowSize`, `MainModuleRawUI.WindowTitle`:** Override methods for the `PSHostRawUserInterface` interface, implementing functionality related to console input/output and manipulation.

* **`MainModuleUI.MainModuleUI()`:** Constructor of custom PowerShell UI which initializes the raw UI.

* **`MainModuleUI.Prompt`, `MainModuleUI.PromptForChoice`, `MainModuleUI.PromptForCredential`, `MainModuleUI.RawUI`, `MainModuleUI.ReadLine`, `MainModuleUI.ReadLineAsSecureString`, `MainModuleUI.Write`, `MainModuleUI.WriteDebugLine`, `MainModuleUI.WriteErrorLine`, `MainModuleUI.WriteLine`, `MainModuleUI.WriteProgress`, `MainModuleUI.WriteVerboseLine`, `MainModuleUI.WriteWarningLine`:** Override methods for the `PSHostUserInterface` interface, handling console interaction like prompts, input, and output with color coding for different message types.

Control Flow

The `MainApp.Main` function is the central control flow:

1. **Initialization:** It creates instances of `MainApp`, `MainModuleUI`, and `MainModule`.
2. **Argument Parsing:** Parses command-line arguments, looking for options like `-wait`, `-extract`, `-end`, and `-debug`.
3. **Runspace Setup:** Creates and opens a PowerShell runspace using the custom `MainModule` host.
4. **Script Execution:** Loads the PowerShell script from an embedded resource. If `-extract` is specified, it writes the script to a file. Otherwise, it adds the script to the PowerShell pipeline. Command line arguments are processed, adding parameters and arguments to the PowerShell command.
5. **Pipeline Setup:** Sets up a PowerShell pipeline that sends output to the custom UI.
6. **Execution and Monitoring:** Starts the PowerShell script execution asynchronously and waits for completion or the `ShouldExit` flag set by the host.
7. **Error Handling:** Checks the invocation state, writes error messages from the PowerShell script to the console via the UI.
8. **Cleanup:** Closes the runspace.
9. **Wait for Key (Optional):** If `-wait` is specified, waits for a key press before exiting.
10. **Exit:** Returns the application's exit code.

The `MainModuleUI` methods handle the various forms of console I/O including colored output, prompts for various input types and credential handling. Error handling is extensively implemented through try-catch blocks and the handling of the PowerShell `Streams.Error` event.

Data Structures

* **Enums:** `Console_Info.FileType`, `Console_Info.STDHandle` are used to represent types of file handles and standard handles.
* **`MainModule.ConsoleColorProxy`:** A simple class used to expose console color properties to the PowerShell script.
* **`MainModuleRawUI.CHAR_INFO`:** A struct representing a character cell in the console buffer (using P/Invoke).
* **`MainModuleRawUI.COORD`:** A struct representing coordinates in the console buffer (using P/Invoke).
* **`MainModuleRawUI.SMALL_RECT`:** A struct representing a rectangular region in the console buffer (using P/Invoke).
* **PSObjects, PSDataCollection:** PowerShell objects and collections used for data exchange in the pipeline.

Malware Family Suggestion

Based on the provided code, it is not possible to definitively classify this as belonging to a specific malware family. The code itself is not malicious; it's a fairly sophisticated C# application designed to run a PowerShell script. However, the *potential* for malicious behavior exists. The embedded PowerShell script (`postApiCoyote1h_V1.ps1`), which is not provided, is the critical factor. If this script contained malicious code, then the application would become a dropper or a loader, a tool delivering the actual payload.

The following elements raise concerns regarding potential abuse:

* **Embedded PowerShell Script:** The core functionality relies on an external script, which is the likely place for malicious actions.
* **API Interaction:** The script's name ("postApiCoyote1h_V1.ps1") suggests interaction with an external server, which could be used for command and control (C2), exfiltration of data or other malicious purposes.
* **Argument Parsing:** The ability to parse command line arguments might allow the behavior of the script (and hence the malware) to be modified.
* **Obfuscation (Potential):** While not obvious in the provided code, the embedded script might be obfuscated to make reverse engineering harder.

Conclusion: The provided code itself is not malware. It's a tool that could be used for legitimate purposes or could be abused to distribute malware. A thorough analysis of the missing `postApiCoyote1h_V1.ps1` PowerShell script is necessary to determine if this is malicious. Without the script, the code is suspicious but not definitively classifiable as a specific malware family. The characteristics suggest it could be used as a part of a larger malware operation, acting as a dropper or loader.