Analysis Report for: 94BAE5BE8B148A1E37B689C1B54728CC.cs

Overall Functionality

This C# code implements a system for interacting with various Nota Fiscal de Serviços Eletrônica (NFSe) web services. It appears to be designed to automate the process of generating, canceling, and querying NFSe documents for different Brazilian municipalities. The code heavily obfuscates its functionality through the use of meaningless function and variable names (e.g., `a`, `b`, `c`, `0`, `1`), and delegate types with the same names, making reverse engineering difficult. It dynamically loads and uses a native DLL ("Protect4a647d98.dll" or "Protectf51f1960.dll"), suggesting an attempt to further hinder analysis and possibly introduce additional functionality.

Function Summaries

* **`.static ()`**: This static constructor is the entry point for initialization. It determines the system's architecture (32-bit or 64-bit) and calls either `g` or `h` accordingly, passing the major version of the .NET framework and an IUnknown pointer. Finally, it releases the IUnknown pointer.

* **`.a(string)`**: A P/Invoke wrapper for the `GetModuleHandle` Windows API function. It takes a module name as input and returns a handle to the module if found, otherwise IntPtr.Zero.

* **`.b(string)`**: A P/Invoke wrapper for the `LoadLibrary` Windows API function. It takes a DLL path as input and returns a handle to the loaded library if successful, otherwise IntPtr.Zero.

* **`.c(string A_0, string A_1)`**: Extracts an embedded resource (DLL) from the assembly, writes it to a file at path `A_0`, and creates a mutex with the name `A_1` to prevent race conditions during file writing. The mutex permissions are set to allow full control to everyone ("World"). Error handling is minimal.

* **`.d()`**: Attempts to load a native DLL ("Protect4a647d98.dll" or "Protectf51f1960.dll" depending on architecture) using `a` (GetModuleHandle). If not found, it extracts the DLL using `c` (extract & write to temp directory) and attempts loading it via `b` (LoadLibrary) up to 100 times, trying different temp directories in the process.

* **`.e(IntPtr, string)`**: A P/Invoke wrapper for the `GetProcAddress` Windows API function. It takes a library handle and a function name as input, and returns a pointer to the function if found, otherwise IntPtr.Zero.

* **`.f(string A_0, Type A_1)`**: Gets a delegate of type `A_1` for a function in the native DLL, identified by name `A_0`. This uses `e` to retrieve the function address from the loaded DLL (`d`).

* **`.g(int A_0, IntPtr A_1)`**: Calls a function from the native DLL (identified as "P0") via the delegate obtained by `f`. Returns a boolean value. This appears to be the 32-bit version of the main native DLL function call.

* **`.h(int A_0, IntPtr A_1)`**: Similar to `g`, but calls "P0" from the native DLL via a delegate, for 64-bit systems. Returns a boolean value.

* **`0(int, IntPtr)`**: A delegate type that matches the signature of the function called by `g` and `h`. The use of '0' as a name is highly obfuscatory.

* **`1()`**: A delegate type with no parameters and a `void` return type (obfuscated name).

* **`a.c(global::a)`**: A delegate type, obfuscated, presumably to interact with the underlying NFSe system.

* **`a.a(global::a, X509Certificate2 A_0)`**: Another obfuscated delegate, likely for certificate handling.

* **`a.f(global::a, object A_0, EventArgs A_1)`** & other similarly named functions in class `a` & other classes: All these functions in various classes (`a`, `b`, `c`, `d`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, `m`, `o`, `p`, `q`, `s`, `t`, `u`, `w`, `x`) call functions within the `` class via the static `Invoke` field (which is an `Invoker` delegate). This indicates that the majority of the application's core logic resides in the obfuscated native DLL. The numeric argument passed to `.Invoke` likely serves as an index or identifier for the function within the DLL.

* **`DotfuscatorAttribute.DotfuscatorAttribute(string a, int c)`**: A custom attribute providing obfuscation metadata. Its getter methods (`A` and `C`) also utilize `.Invoke`, pointing again to the core logic residing in the native DLL.

* **`v.a()`**: A static method that calls an obfuscated function (`1`) via `.Invoke`, likely the main entry point of the application.

* **`y.c(this string A_0)`, `y.b(this string A_0)`, `y.a(this string A_0)`**: Extension methods on the `string` type, suggesting manipulation of strings as part of the NFSe process, possibly involving path or data handling. These also make use of `.Invoke`.

* **`Acesso.TableConsultaDocumentos`**: A property to get/set the `DataTable` for storing document information. Getters and setters rely on `.Invoke`

* **`Acesso.ErroRetorno`**: A property to get/set error messages. Getters and setters rely on `.Invoke`

* **`Acesso.car_cid()`**: Function to retrieve array list, likely containing data, this too uses `.Invoke`.

* **`Acesso.IniciaAcesso(...)`**: The main initialization function for the `Acesso` class. It receives all necessary parameters (including a certificate) and calls into the hidden DLL via `.Invoke`.

Control Flow

The crucial control flow is largely hidden within the native DLL called by `.d()`, `.g()`, `.h()`, and all the various delegate calls from other classes. The managed code primarily acts as a wrapper, handling resource extraction, DLL loading, and marshalling of data to and from the native code. The `for` loop in `.d()` shows a retry mechanism, which is a common characteristic of malware that tries to persist even when the malware is removed.

Data Structures

* **`IntPtr`**: Used extensively for pointers to unmanaged memory (DLL handles and function pointers).

* **`Delegate`**: Used to represent functions in the native DLL.

* **`Mutex`**: Ensures exclusive access to files during writing.

* **`byte[]`**: Used to store the contents of embedded resources (DLLs).

* **`DataTable`**: Likely used to represent structured data related to NFSe documents.

* **`ArrayList`**: Used to store lists of data.

* **`X509Certificate2`**: Represents an X.509 certificate, vital for digital signatures.

Malware Family Suggestion

Given the heavy obfuscation, dynamic loading of a native DLL, the creation of mutexes with world access, and the overall functionality mimicking legitimate NFSe interaction, this code is highly suspicious and strongly suggests a **backdoor or information stealer**. The "Protect..." names in the DLLs are also a red flag, disguising their true purpose. The extensive use of delegates to call functions within the native DLL makes it difficult to definitively classify the malware family without analyzing the contents of the native DLL. However, its behavior is consistent with techniques used by advanced persistent threats (APTs) and other sophisticated malware designed for persistence and stealthy data exfiltration. The fact that it interacts with NFSe systems suggests a possible motive: the stolen data might be financial information related to invoices.