Analysis Report for: payload.cs

Overall Functionality

The C# code in `Flutter.cs` appears to be a component of a malware designed to execute arbitrary code. It uses several techniques to obfuscate its actions and evade detection. The code decrypts several byte arrays from a dictionary (`FastFS`), allocates memory using `VirtualAlloc`, copies decrypted code into that memory, and then executes it using a dynamically resolved function pointer. The code relies heavily on memory manipulation, decryption, and dynamic function loading, all strong indicators of malicious intent. The "DOTNETv9" check acts as a simple conditional to only execute the core payload if the correct value is found.

Function Summaries

* **`VirtualAlloc` (DllImport):** Imports the Windows API function `VirtualAlloc`, used to allocate a region of memory. Not implemented by the code itself, but crucial to its functionality.

* **`VirtualProtect` (DllImport):** Imports the Windows API function `VirtualProtect`, used to change the protection of a memory region (e.g., to make it executable). Again, crucial for executing the loaded code.

* **`GetModuleHandle` (DllImport):** Imports the Windows API function `GetModuleHandle`, used to retrieve the base address of a loaded module (like "kernel32.dll"). Used to find addresses of functions like `GetProcessHeap`, `HeapAlloc`, and `HeapFree`.

* **`GetProcAddress` (DllImport):** Imports the Windows API function `GetProcAddress`, used to retrieve the address of a specific function within a loaded module. Used to get addresses of Windows API functions dynamically.

* **`memcmp` (DllImport):** Imports the C runtime library function `memcmp`, used for comparing byte arrays. Used for comparing a decrypted section with the string "DOTNETv9".

* **`DecryptBytes`:** This function decrypts a byte array using Rijndael (AES) encryption in CBC mode with PKCS7 padding. It takes the encrypted byte array (`b`) and a 32-byte key/IV (`pharse`) as input and returns the decrypted byte array.

* **`RedistBuildArgs`:** This function copies a Unicode-encoded string (from `Flutter.Arguments`) and a two-byte array into a specified memory location (`lpUserData`). It seems to prepare arguments to pass to the injected code.

* **`Draw`:** This is the main function of the class. It acts as the orchestrator, coordinating the decryption, memory allocation, and execution of malicious code. It takes a decryption key (`pharse`), configuration data (`ConfigData`), and a dictionary of encrypted data (`FastFS`) as input. It decrypts and executes code based on the data contained within `FastFS`.

* **`DelegateAOTMain` (delegate):** A delegate defining the signature of the injected code that will be executed.

* **`DelegateBuildArgsFunc` (delegate):** A delegate defining the function signature for `RedistBuildArgs`, which builds arguments for the injected code.

Control Flow

* **`DecryptBytes`:** Straightforward implementation of AES decryption. No loops or complex conditionals.

* **`RedistBuildArgs`:** Simple function, copies data from byte arrays into a given memory location.

* **`Draw`:** This function contains nested conditional checks to validate the integrity of the data read from `FastFS`. It will only execute the malicious code if several conditions are met:
1. Entry 5 in `FastFS` exists and is 16 bytes long.
2. Decrypting entry 5 reveals "DOTNETv9".
3. Entry 2 in `FastFS` exists and its length is a multiple of 16 bytes.
4. Entry (0 or 1 based on architecture) in `FastFS` exists and its length is a multiple of 16 and less than 57344 bytes.
5. Entry 32 in `FastFS` exists and is 16 bytes long.
6. Entry 4 in `FastFS` exists, its length is a multiple of 16 bytes and it's 2048 bytes long.
If all conditions are true, the code then allocates memory, copies decrypted code and functions into it, sets appropriate memory protections, and executes the injected code (`delegateAOTMain`).

Data Structures

* **`FastFS` (Dictionary):** A dictionary storing encrypted data indexed by integers. This appears to be a container for different sections of the malicious payload, each encrypted and identified by a numerical key.

* **`pharse` (byte[]):** A 32-byte key and initialization vector (IV) used for AES decryption.

Malware Family Suggestion

Based on its functionality, this code strongly resembles a **Dropper** or **Loader** malware. It doesn't directly perform malicious actions itself, but it

downloads, decrypts, and executes other malicious code. The dynamic loading of functions, memory allocation and manipulation, and encryption/decryption techniques are all common characteristics of sophisticated malware designed to evade detection and analysis. The structure hints at a modular design, likely allowing for the easy swapping of payloads for future modifications. The use of AES suggests a level of sophistication beyond simpler malware.

Additional Notes

* The use of `IntPtr` and low-level memory operations (e.g., `Marshal.Copy`, `Marshal.WriteIntPtr`) indicates a deliberate attempt to work directly with memory and bypass normal code execution flow.

* The error handling is minimal; the `try-catch` block around `delegateAOTMain` does nothing, implying that the malware might not care about errors during execution.

* The use of a 32-byte "phrase" for a 128-bit AES key/IV is unusual. A standard AES-128 uses a 16-byte key and 16-byte IV. This oddity further contributes to the obfuscation. It might be splitting a single 32-byte key into 16-byte key and IV.

This code should be treated as highly suspicious and dangerous. Further analysis would require examining the content of `ConfigData` and the decrypted byte arrays from `FastFS` to determine the precise nature of the payload.