Analysis Report for: BE8A03609679B43E7FDD003364BED49B.exe

Overall Functionality

This C code (which is actually JavaScript embedded in a C-style comment block) implements a sophisticated anti-debugging and anti-tampering technique. It aims to prevent the analysis and modification of a web application by selectively aborting the execution of specific JavaScript functions or properties if they are accessed from a suspected debugging or malicious environment. The mechanism achieves this by injecting a stack trace analysis into getters and setters of targeted JavaScript objects.

Function Summaries

* **`uBOL_abortOnStackTrace()` (IIFE):** This is an Immediately Invoked Function Expression (IIFE), which creates a local scope to isolate the code from the global scope. It's the main entry point of the code.
* **`abortOnStackTrace(chain, needle, ...extraArgs)`:** This function intercepts access to specified properties within the `window` object (and its nested objects). `chain` specifies the property path (e.g., "History.pushState"), `needle` is a regular expression or string pattern used to identify suspicious stack traces, and `extraArgs` contains optional arguments (e.g., logging level). It uses a proxy to intercept getters and setters.
* **`getExceptionTokenFn()`:** Generates a unique random token used to identify the stack trace generated by the anti-debugging mechanism itself, thus filtering out its own traces. Returns the token.
* **`matchesStackTraceFn(needleDetails, logLevel)`:** Analyzes the stack trace of a deliberately created error, looking for matches with the provided `needleDetails` (a regular expression or string). `logLevel` controls the logging verbosity. Returns `true` if a match is found according to `needleDetails.expect` (and logs the stack trace), `false` otherwise.
* **`safeSelf()`:** Returns a safe sandboxed object containing references to frequently used JavaScript built-in functions and objects, shielding against potentially modified globals. This is also responsible for logging to the console or through a BroadcastChannel.
* **`getRandomTokenFn()`:** Generates a random string used as a unique identifier to filter out the stack trace generated by the `getExceptionTokenFn()`.
* **`collectArgIndices(hn, map, out)`:** Collects indices from `hostnamesMap` or `exceptionsMap` based on hostname `hn`, adding them to the `Set` `out`.
* **`indicesFromHostname(hostname, suffix)`:** Extracts hostnames from URLs and uses them to find relevant indices in `hostnamesMap` and `exceptionsMap`.
* **Functions within `safeSelf()`:** This function contains several helper functions for regular expression handling, logging, argument parsing, and idle callback management.

Control Flow

* **`abortOnStackTrace`:** This function recursively traverses the object property chain specified by `chain`. For each property, it defines a getter and setter using `Object.defineProperty`. These getters and setters check if the stack trace (obtained using `matchesStackTraceFn`) matches `needle`. If it does, a `ReferenceError` is thrown.
* **`matchesStackTraceFn`:** Creates an error object, extracts its stack trace, normalizes it (removing the token, simplifying URLs), then tests it against `needleDetails` using `safe.testPattern`. Logging happens based on the `logLevel`.
* **`safeSelf()`:** This function has complex logic to check for and establish logging capabilities. If a BroadcastChannel is available (indicating a companion logging process), it logs via the channel; otherwise, it falls back to `console.log`.

Data Structures

* **`argsList`:** An array of arrays, where each inner array represents a target property path and its corresponding stack trace detection pattern.
* **`hostnamesMap`:** A `Map` associating hostnames with integers (indices into `argsList`), or arrays of indices for more complex mapping.
* **`exceptionsMap`:** Similar to `hostnamesMap` but holds exceptions (hostnames that should *not* trigger the anti-debugging mechanism).
* **`todoIndices`:** A `Set` storing the indices of `argsList` to be processed.
* **`tonotdoIndices`:** A `Set` storing exceptions; indices to be excluded from processing.
* **`safe` (object within `safeSelf()`):** A large object acting as a sandbox, containing copies of core Javascript methods and functions. This protects against global function tampering.
* **`entries`:** An array of objects, each containing a hostname and its index in the ancestorOrigins list.

Malware Family Suggestion

The functionality exhibited strongly suggests this code is part of an anti-debugging or anti-tampering mechanism, often used by malware or in software protection schemes (though not inherently malicious on its own). The level of sophistication, including the use of stack trace analysis, random tokens, and a communication channel to a logger, indicate advanced techniques employed to hinder reverse engineering and analysis. It could be part of a broader malware or software protection system. While not a stand-alone malware, its intended use is consistent with techniques employed to obscure and protect malicious or intrusive software. Its purpose is to make analysis and modification much harder, hindering security researchers and potentially hiding a separate malicious activity within the larger application.