Analysis Report for: 2C66D014A153C3C2DDB37B8F766FD22A.exe

Overall Functionality

This C code snippet (although presented with JavaScript syntax within a C comment block), implements a sophisticated anti-debugging and anti-tampering mechanism. It aims to prevent analysis and modification of the code by selectively aborting execution when specific function calls or stack trace patterns are detected. This is achieved by dynamically creating proxies for potentially vulnerable JavaScript properties and monitoring the stack trace for predefined "needle" patterns indicative of debugging or malicious activity.

Function Summaries

* **`uBOL_abortOnStackTrace()` (IIFE):** This is an Immediately Invoked Function Expression (IIFE) that acts as a container, isolating the code from the global scope. It initializes several variables (`scriptletGlobals`, `argsList`, `hostnamesMap`, `exceptionsMap`, `hasEntities`, `hasAncestors`, `entries`) and orchestrates the main logic of the anti-debugging/anti-tampering mechanism. It does not take parameters or return a value.

* **`abortOnStackTrace(chain, needle, ...extraArgs)`:** This function takes a property access chain (`chain`), a regular expression pattern or string (`needle`), and optional extra arguments (`extraArgs`). It creates a proxy for the specified property chain. If the `needle` pattern is found in the stack trace when the property is accessed (either read or written), a `ReferenceError` is thrown, effectively aborting execution in that path.

* **`getExceptionTokenFn()`:** This generates a random token that's used to identify stack traces related to the deliberately thrown `ReferenceError`. It temporarily overrides the `onerror` event handler to catch errors containing the token and suppress them to avoid revealing the anti-debugging mechanism. Returns the randomly generated token.

* **`matchesStackTraceFn(needleDetails, logLevel)`:** This function takes a regular expression object (`needleDetails`) and a log level (`logLevel`). It throws an error, extracts the stack trace, normalizes it, and checks if it matches the `needleDetails`. Based on the `logLevel`, it may log the stack trace. Returns `true` if the stack trace matches the pattern, `false` otherwise.

* **`safeSelf()`:** This function creates and returns a "safe" object (`safe`). This object contains numerous methods and properties providing access to standard JavaScript functions and objects, ensuring controlled access and preventing the attacker from making use of some potentially harmful functions. If `scriptletGlobals.bcSecret` is defined, it sets up a BroadcastChannel for communication with a logger (likely in a separate process).

* **`getRandomTokenFn()`:** Generates a random alphanumeric token used by `getExceptionTokenFn()`. Returns the random token as a string.

* **`collectArgIndices(hn, map, out)`:** A helper function to collect indices from the `hostnamesMap` or `exceptionsMap` based on a hostname. It adds the relevant indices to the `out` set.

* **`indicesFromHostname(hostname, suffix)`:** Processes a hostname and determines which indices from `argsList` should be processed. It handles both exact hostnames and wildcard patterns.

Control Flow

The main control flow resides within the `uBOL_abortOnStackTrace` IIFE. The key logic is:

1. **Initialization:** Several maps and sets are initialized with data related to hostnames and exceptions.
2. **Hostname Processing:** The `indicesFromHostname` function identifies relevant indices based on the current document's origin and its ancestors (if available).
3. **Application:** The code iterates through the identified indices (`todoIndices`), skipping those in `tonotdoIndices` (exceptions). For each index, it attempts to execute `abortOnStackTrace` with the corresponding arguments from `argsList`. Any errors during this process are silently caught.

The `abortOnStackTrace` function uses recursion (`makeProxy`) to traverse the property access chain. The core logic in `makeProxy` is to define getters and setters for each property, checking against the `needle` pattern within the stack trace on each access.

The `matchesStackTraceFn` function processes the stack trace, normalizing it before applying the pattern match.

Data Structures

* **`argsList`:** An array of arrays, where each inner array contains a property access chain and a corresponding pattern to watch for in stack traces.
* **`hostnamesMap`:** A `Map` associating hostnames with indices in `argsList`. Indices can be numbers or arrays of numbers.
* **`exceptionsMap`:** A `Map` similar to `hostnamesMap`, but represents exceptions; if found, an index will be excluded from processing.
* **`todoIndices`:** A `Set` containing indices of `argsList` to process.
* **`tonotdoIndices`:** A `Set` containing indices of `argsList` to exclude from processing.

Malware Family Suggestion

Based on its functionality, this code strongly resembles techniques used in **anti-analysis/anti-debugging malware**. The code's primary goal is to actively thwart reverse-engineering and debugging efforts. The sophisticated use of stack trace analysis, random token generation, and selective property proxying shows a level of intent to hinder security analysis. While not directly malicious in itself, this code could be easily incorporated into other malicious software to make it more difficult to analyze and remove. The use of a logger in a separate process further obscures its actions, making it harder to track. This is a common feature in sophisticated malware families aimed at evading detection and analysis.