Analysis Report for: http.tcl

Overall Functionality

This C code (although written in Tcl, a scripting language often embedded in C applications) implements a client-side HTTP library for performing GET, POST, and HEAD requests. It's designed with security in mind, aiming to be usable even within a restricted "safe" environment (like a sandboxed Tcl interpreter), avoiding potentially dangerous functions like `vwait`. The library uses a callback mechanism for handling asynchronous events, such as data arrival from the server. It includes features for configuring proxies, setting headers, handling timeouts, and processing responses, including handling various content types and sizes.

Function Summaries

* **`package provide http 1.0`**: This line declares the package version. It is specific to Tcl's package management.

* **`array set http { ... }`**: Initializes an array `http` to store global configuration options (e.g., `-accept`, `-proxyhost`, etc.).

* **`http_config {args}`**: Configures HTTP settings. It takes a variable number of arguments. If no arguments are provided, it returns the current configuration. Otherwise, it sets or retrieves individual configuration options. Returns a configuration value or an error message.

* **`httpFinish {token {errormsg ""}}`**: Cleans up resources associated with an HTTP request. It takes a request token and an optional error message. It closes sockets, cancels timers, and executes a user-provided callback. Doesn't explicitly return a value.

* **`http_reset {token {why reset}}`**: Resets the state of an HTTP request. It takes a request token and a reason for the reset. It sets the request status and stops any ongoing network events. Doesn't explicitly return a value.

* **`http_get {url args}`**: Performs an HTTP GET, POST, or HEAD request. Takes a URL and optional arguments. It creates a socket, sends the request, and manages the asynchronous response handling. Returns a token identifying the request.

* **`http_data {token}`**: Retrieves the body of the HTTP response. Takes a request token. Returns the response body.

* **`http_status {token}`**: Retrieves the status of an HTTP request. Takes a request token. Returns the request status ("ok", "error", "eof", etc.).

* **`http_code {token}`**: Retrieves the HTTP status code from the response. Takes a request token. Returns the HTTP status code (e.g., "200 OK").

* **`http_size {token}`**: Retrieves the number of bytes received. Takes a request token. Returns the number of received bytes.

* **`httpEvent {token}`**: Callback function triggered by socket read events. Handles reading HTTP headers and body data. Doesn't explicitly return a value.

* **`httpCopyStart {s token}`**: Initiates a background data copy using `fcopy`. Used for efficient handling of large responses. Doesn't explicitly return a value.

* **`httpCopyDone {token count {error {}}}`**: Callback function for `fcopy`. Handles completion or errors during data transfer. Doesn't explicitly return a value.

* **`httpEof {token}`**: Handles end-of-file conditions during data reception. Takes a request token. Doesn't explicitly return a value.

* **`http_wait {token}`**: Waits for an HTTP request to complete. Takes a request token. Returns the request status.

* **`http_formatQuery {args}`**: Formats query parameters for POST requests. Takes key-value pairs and creates a URL-encoded query string. Returns the formatted query string.

* **`httpMapReply {string}`**: Performs URL encoding of a string. Takes a string. Returns the URL-encoded string.

* **`httpProxyRequired {host}`**: A default proxy filter that checks HTTP configuration for proxy settings. Takes a hostname. Returns a list containing proxy host and port or an empty list.

Control Flow

* **`http_get`**: This function is crucial. It follows this flow:
1. **Initialization**: Sets up request state variables, including default values.
2. **Argument Parsing**: Parses and validates the provided arguments.
3. **URL Parsing**: Extracts host, port, and path from the URL.
4. **Proxy Handling**: Checks for and uses proxy settings if configured.
5. **Socket Creation**: Creates a socket to the server (or proxy).
6. **Request Sending**: Sends the HTTP request to the server.
7. **Event Registration**: Registers a callback (`httpEvent`) for socket read events.
8. **Waiting (optional)**: If no immediate command is provided, it waits for the request to complete via `http_wait`.

* **`httpEvent`**: This callback function is the heart of the asynchronous handling:
1. **EOF Check**: Checks for end-of-file.
2. **Header Parsing**: If in the "header" state, it reads and parses HTTP headers.
3. **Body Reading**: If in the "body" state, it reads and processes the response body. This can involve a custom handler or simple block reads.
4. **Progress Reporting**: If a progress callback is provided, it's invoked.

* **`httpFinish`**: This function handles cleanup and error handling across different scenarios.

Data Structures

* **`http` (array)**: A global array storing HTTP configuration parameters.
* **`state` (array)**: An array used to store the state of an individual HTTP request (for each request token). This is a crucial data structure for managing the asynchronous nature of the library.

Malware Family Suggestion

The code itself is not inherently malicious. It's a functional HTTP client library. However, its functionality *could* be exploited by malware. A sophisticated malware program could leverage this library to:

* **Command and Control (C&C):** Communicate with a remote server to receive instructions or exfiltrate stolen data, obfuscating its actions by using seemingly benign HTTP requests. This is a common tactic for many malware families, not limited to a specific one.
* **Data Exfiltration:** Download malicious payloads or upload stolen information discreetly.
* **Information Gathering:** Perform reconnaissance on the infected system by making HTTP requests to gather network information or identify software vulnerabilities.

Therefore, the malware family classification is not specific to any one type but rather depends on how this code is used within a larger context. It could be a component within a wide range of malware, enhancing its capabilities to interact with the network in a more subtle fashion. The ability to handle GET, POST, and HEAD requests means a wide range of actions could be performed.