Analysis Report for: http-2.9.5.tm

Overall Functionality

This C code (actually, it's Tcl code embedded within a C-like structure due to the `proc` keyword) implements a client-side HTTP library for making GET, POST, and HEAD requests. It's designed to be safe for use in untrusted contexts (like safe interpreters in Tcl) by utilizing a callback mechanism instead of blocking operations like `vwait`. The library handles persistent connections (keep-alive), pipelining, various encoding schemes, proxies, and provides options for customization and error handling. It's robust in managing asynchronous operations and potential network issues.

Function Summaries

Function Name	Purpose	Parameters	Return Value
`http::init`	Initializes internal data structures, including URL type mappings and quoting maps. Closes any existing open sockets.	None	None
`http::register`	Registers a new URL protocol handler (e.g., `https`).	`proto` (protocol), `port`, `command` (socket creation command)	List of port and command registered
`http::unregister`	Unregisters a URL protocol handler.	`proto` (protocol)	List of port and command unregistered, or error code
`http::config`	Configures the HTTP library's global settings.	Key-value pairs for options	Config value or error code
`http::Finish`	Cleans up a socket connection and handles callbacks.	`token` (connection token), `errormsg` (optional), `skipCB` (optional)	None
`http::KeepSocket`	Manages persistent sockets, connecting them to queued tasks.	`token` (connection token)	None
`http::CheckEof`	Checks for EOF on a socket and closes it if necessary.	`sock` (socket)	None
`http::CloseSocket`	Closes a socket and removes it from the persistent socket table.	`s` (socket), `token` (optional)	None
`http::CloseQueuedQueries`	Clears read and write queues associated with a connection.	`connId` (connection ID), `token` (optional)	None
`http::Unset`	Unsets variables associated with a connection.	`connId` (connection ID)	None
`http::reset`	Resets a connection's state.	`token` (connection token), `why` (reason)	None
`http::geturl`	Makes an HTTP request.	`url`, key-value pairs for options	Connection token or error code
`http::Connected`	Callback function called when a connection is established.	`token`, `proto`, `phost`, `srvurl`	None
`http::registerError`	Registers an error for a connection.	`sock` (socket), error message	(optional) Error message
`http::DoneRequest`	Handles request completion, queuing the next request or response.	`token` (connection token)	None
`http::ReceiveResponse`	Connects a token to its socket for reading.	`token` (connection token)	None
`http::EventGateway`	Wrapper around `http::Event` to handle coroutine recursion.	`sock`, `token`	Result/options from the coroutine or error
`http::NextPipelinedWrite`	Manages pipelined writes.	`token` (connection token)	None
`http::CancelReadPipeline`	Cancels pipelined responses.	Trace variables	None
`http::CancelWritePipeline`	Cancels queued requests.	Trace variables	None
`http::ReplayIfDead`	Re-attempts a request on a fresh socket if the previous connection died.	`tokenArg`, `doing` ("read" or "write")	None
`http::ReplayIfClose`	Handles queued requests when the server sends "Connection: close".	`Wstate`, `Rqueue`, `Wqueue`	None
`http::ReInit`	Re-initializes a connection token for retry.	`token` (connection token)	Boolean (success/failure)
`http::ReplayCore`	Core logic for replaying requests.	`newQueue` (list of connection tokens)	None
`http::data`, `http::status`, `http::code`, `http::ncode`, `http::size`, `http::meta`, `http::error`	Accessors for connection data.	`token` (connection token)	Corresponding data
`http::cleanup`	Cleans up a connection token.	`token` (connection token)	None
`http::Connect`	Callback when an asynchronous connection completes.	`token`, `proto`, `phost`, `srvurl`	None
`http::Write`	Writes POST data to the socket.	`token` (connection token)	None
`http::Event`	Core coroutine for handling socket input.	`sock` (socket), `token` (connection token)	None
`http::TestForReplay`	Tests whether a request should be replayed after a socket closure.	`token`, `doing`, `err`, `caller`	Boolean (replay/no replay)
`http::IsBinaryContentType`	Checks if a content type indicates binary data.	`type` (content type)	Boolean
`http::getTextLine`	Reads a line from a socket in CRLF mode.	`sock` (socket)	Line of text
`http::BlockingRead`	Blocking read for coroutines.	`sock` (socket), `size` (number of bytes)	Data read
`http::BlockingGets`	Blocking gets for coroutines.	`sock` (socket)	Line of text
`http::CopyStart`	Starts asynchronous copying of data using `fcopy`.	`sock`, `token`, `initial` (boolean)	None
`http::CopyChunk`	Handles a chunk of data during chunked transfer.	`token`, `chunk`	None
`http::CopyDone`	Callback for `fcopy` completion.	`token`, `count`, `error` (optional)	None
`http::Eot`	Handles end-of-transmission conditions.	`token`, `reason` (optional)	None
`http::wait`	Waits for a connection to complete.	`token` (connection token)	Connection status
`http::formatQuery`	Formats a query string.	Name-value pairs	Query string
`http::mapReply`	URL-encodes a string.	`string`	Encoded string
`http::ProxyRequired`	Default proxy filter.	`host` (destination host)	Proxy settings
`http::CharsetToEncoding`	Maps a charset to a Tcl encoding.	`charset`	Tcl encoding or "binary"
`http::ContentEncoding`	Returns the list of content-encoding transformations	`token`	List of transformations
`http::ReceiveChunked`	Receives chunked data.	`chan`, `command`	None
`http::make-transformation-chunked`	Creates a chunked data transformation coroutine.	`chan`, `command`	None

Control Flow (Significant Functions)

* **`http::geturl`**: This function is the main entry point for making HTTP requests. Its control flow is complex due to the asynchronous nature of the operations. It involves:
1. **Initialization**: Creates a token, initializes the connection state.
2. **Option Processing**: Parses and validates command-line options.

3. **URL Parsing and Validation**: Extracts and validates the URL components, ensuring RFC 3986 compliance if `-strict` is set.
4. **Connection Handling**: Tries to reuse an existing persistent connection if `-keepalive` is enabled, otherwise creates a new socket. Handles both proxy and direct connections.
5. **Request Queuing**: If the connection is reused and another request is in progress, queues the current request.
6. **Asynchronous Connection**: Uses `fileevent` to trigger `http::Connect` when the socket becomes writable.
7. **Synchronous or Asynchronous Completion**: If no `-command` callback is provided, waits synchronously using `vwait` for the connection to complete; otherwise, completes asynchronously using callbacks.
8. **Error Handling**: Handles various errors during connection establishment and request processing.

* **`http::Event`**: This coroutine is the heart of the asynchronous response processing. It uses a `while 1` loop with `yield` to handle incoming data from the socket in an event-driven manner. Its control flow depends on the current state (`connecting`, `header`, `body`, `complete`). It parses headers, handles chunked transfer encoding, manages callbacks (`-handler`, `-progress`), and detects EOF conditions. It heavily relies on conditional statements and `catch` blocks for error handling.

* **`http::Finish`**: This function is responsible for cleaning up a connection when a transaction completes, including closing the socket, cancelling timers, and calling user-defined callbacks. It checks various conditions (timeout, error, EOF, `-keepalive` status, connection close) to determine whether to close the socket immediately or keep it open for reuse.

Data Structures

* **`http` (array):** Stores global configuration options for the HTTP library.
* **`urlTypes` (array):** Maps URL protocols (e.g., "http", "https") to default ports and socket creation commands.
* **`socketMapping` (array):** Maps connection IDs (host:port) to sockets, allowing for persistent connections.
* **`socketRdState`, `socketWrState`, `socketRdQueue`, `socketWrQueue`, `socketClosing`, `socketPlayCmd` (arrays):** These arrays manage the state of persistent sockets, tracking which token has read/write access, queued requests/responses, whether the connection is closing, and commands to execute if the connection is closed.
* **`token` (namespace variable):** Represents a connection; it's the name of an array that holds the state of an HTTP transaction. Each token is a unique identifier.

Malware Family Suggestion

Based solely on its functionality, this code is not inherently malicious. It's a sophisticated HTTP client library. However, a malicious actor could potentially modify or misuse this code in several ways:

* **Command Injection:** If the `command` argument in `http::register` is not carefully sanitized, it could be vulnerable to command injection attacks.
* **Data Exfiltration:** A malicious version could be designed to exfiltrate sensitive data gathered from HTTP requests.
* **Denial of Service (DoS):** Modifying the code to make excessive or malformed requests could be used for a DoS attack.

Therefore, while the original code is benign, the *potential* for malicious use makes a categorical malware family assignment impossible without examining a specific instance. The underlying functionality has no direct correspondence to any known specific malware family. Its potential for abuse is similar to that of many legitimate libraries or tools that can be twisted for harmful purposes.