Analysis Report for: 0B918A234D1717D66F0C077CE1495235.js

Overall Functionality

This C code is obfuscated malware designed to download and execute additional code from remote servers. It uses several techniques to hide its malicious intent, including base64 encoding, code obfuscation through hexadecimal representation of strings and variable names, and the use of Python's `eval()` function (indirectly through the Python interpreter which must be present on the system). The code's primary actions involve:

1. **Initialization:** Sets up environment variables, paths, and other configurations based on system information. It gathers data about the system, including the OS, path to the program, and user-agent.

2. **Fetching Remote Data:** Retrieves data from a hardcoded URL (`https://gitlab.com/afonsoscosta/magellan_matrix/-/raw/main/magellan.txt`) and potentially other URLs depending on configuration settings, likely containing further instructions or payloads. This fetch operation handles various response encodings (UTF-8, Latin-1). The code also checks for existence of specific files ("favorites", "source_file") presumably containing additional configuration or payloads.

3. **Processing and Decoding:** The downloaded data undergoes decoding steps, including base64 decoding and UTF-8/Latin-1 decoding based on the response headers.

4. **Execution:** The decoded data is compiled and then executed using `compile()` and `exec()`, enabling the malware to run arbitrary code. The existence of a `get_decode` function suggests further layers of encoding.

5. **Persistence:** The code attempts to establish persistence using a key-value store (represented by the `addon` object), storing at least the program path and version. It adds a new source file that will be used to add further payloads or to update itself.

6. **Additional Modules:** Uses and imports several modules suggesting possible interactions with the system (likely file system operations and network communication) in other sections.

Function Summaries

* **`NoRedirection(urllib_error.HTTPError)`:** A custom exception handler for HTTP errors; it simply returns the response.

* **`makeRequest(url, headers=None)`:** Makes an HTTP request to a given URL, handling headers and different content-types, including charset detection. Returns the response body.

* **`addon_log(string, level=xbmc.LOGDEBUG)`:** Logs a message with a specified log level.

* **`getSources()`:** Attempts to retrieve sources of further payloads either from a local "favorites" file, a "source_file", or from a default URL (`origem`).

* **`index()`:** Similar to getSources(). Calls addDir for the favorites

* **`addSource(url=None)`:** Adds a new source, either from a user-defined URL or a setting "new_file_source".

* **`rmSource(name)`:** Removes a source based on its name

* **`getSoup(url, data=None)`:** Fetches data from a URL (potentially using POST method), processes it as XML using `ElementTree`, and returns an `ElementTree` object (or None). The function handles redirects and potentially downloads a webpage to get a specific section as result.

* **`processPyFunction(data)`:** Processes a string to extract a Python function from a `$pyFunction:` tag

* **`getData(url, fanart, data=None)`:** Retrieves data using `getSoup`, extracts data from the XML response, and potentially handles channels and items.

* **`getChanelItems(name, url, fanart)`:** Retrieves items from a channel.

* **`getSubChanelItems(name, url, fanart)`:** Retrieves subitems from a sub-channel.

* **`getItems(items, fanart, dontLink=False)`:** Processes a list of items, handles different item types (including YouTube links), potentially adds items to a playlist, downloads additional data for each item. This appears to be where much of the payload execution happens.

* **`getGoogleRecapchaResponse(captchakey, cj, type=1)`:** Interacts with Google reCAPTCHA to solve a captcha. Handles requests for different tasks (get, post, rawpost).

* **`getURL(url, cookieJar=None, post=None, timeout=20, headers=None, noredir=False)`:** Makes an HTTP request, handling cookies and headers and different http methods; returns the response body.

* **`get_decode(str, reg=None)`:** Decodes a string. If `reg` is provided it will process it

* **`javascriptrUnescape(str)`:** Unescapes a JavaScript string.

* **`askCaptcha(m, html_page, cookieJar)`:** Asks for captcha resolution.

* **`askCaptchaNew(imageregex, html_page, cookieJar, m)`:** Asks for captcha resolution (new method).

* **`addDir(name, url, mode, iconimage, fanart, description, genre, date, credits, showcontext=False, regexs=None, reg_url=None, allinfo={})`:** Adds an item (presumably to a playlist or directory listing) with metadata.

* **`rmFavorte(name)`:** Removes a favorite

* **`urlsolver(url)`:** Solves URLs using a resolver module to resolve short links.

* `tryplay(url, listitem, dialog=None)`: Attempts to play a given video URL using a custom player.

* `play_playlist(name, mu_playlist, queueVideo=None)`: Plays a video playlist, using a dialog for selection if necessary. The function appears to handle both single videos and videos with multiple streams (perhaps different resolutions).

* `download_file(name, url)`: Attempts to download a file. However the code is commented out and probably not functional.

* `_search(url, name)`: Searches for videos on various platforms (YouTube, Dailymotion, Vimeo).

* `__unpack(p, a, c, k, e, d, iteration, v=1)`: Recursive function used in a decoder. Looks like a custom decoder to unpack data.

* `__itoa(num, radix)`: Converts an integer to a string representation of a specific base.

* `__itoaNew(cc, a)`: Another integer to string conversion function

* `getCookiesString(cookieJar)`: Generates a string representation of cookies.

* `saveCookieJar(cookieJar, COOKIEFILE)`: Saves a cookie jar to a file.

* `getCookieJar(COOKIEFILE)`: Loads a cookie jar from a file.

* `doEval(func_call, page_data, Cookie_Jar, m)`: Executes a Python function (dynamically generated) and returns the result.

* `doEvalFunction(func_call, page_data, Cookie_Jar, m)`: Executes a Python function and returns the result. This function is mostly identical to `doEval` except it writes the result to a file.

Control Flow

The control flow is complex due to obfuscation, but the general pattern is:

1. **Initialization:** The program starts by initializing variables and settings.
2. **Main Loop (implicit):** The main logic is not in an explicit `while` or `for` loop, but rather a sequence of function calls.
3. **Conditional Branches:** Many functions contain conditional statements (`if`, `elif`, `else`) based on the existence of files, success of HTTP requests, content types, and various system checks. These conditionals determine the flow, particularly in `getSources()`, `makeRequest()`, `getSoup()`, `getItems()`, `addDir()`, `tryplay()`, `play_playlist()`.
4. **Exception Handling:** The `try...except` blocks handle potential errors, such as network issues or file I/O errors.
5. **Recursive Calls:** The `__unpack` function utilizes recursion to decode layers of data.

Data Structures

* **Lists (`[]`):** Used extensively to store arrays of strings (URLs, filenames, parameters). Examples: `resolvable_url`, `g_ignorSetResolved`, `sources`.

* **Dictionaries (`{}`):** Used to store key-value pairs (metadata, headers, parameters). Examples: `allinfo`, `headers`.

* **`cookieJar` (from `http.cookiejar`):** Used for handling HTTP cookies.

* **`ElementTree` objects (from `xml.etree.ElementTree`):** Used to parse XML data downloaded from remote servers.

* **`xbmc` object:** Suggests interaction with the XBMC/Kodi media center application.

* **`sys` object:** Python system object (used via the Python interpreter called indirectly)

Malware Family Suggestion

Based on the functionality, this code strongly resembles a **downloader/dropper** type of malware. It downloads further code from remote locations

and executes it, the key indicator that this is a downloader/dropper malware is the use of the `exec` function which allows to execute arbitrary code fetched from the network which in itself is a very common characteristic of downloaders/droppers.

The obfuscation techniques and the structure of the code indicate an attempt to evade detection. The use of Google reCAPTCHA interaction points to an increased sophistication, suggesting a malware author who is trying to make it more difficult to analyze the code. It's also likely the downloaded code will be a fully fledged malware such as a remote access trojan, information stealer, or other kind of payload. Without executing the code, the precise classification beyond a downloader/dropper is impossible.