Analysis Report for: 0B918A234D1717D66F0C077CE1495235.exe

Overall Functionality

This C code is obfuscated malware designed to download and execute additional malicious code. It uses several techniques to hide its true intentions:

1. **Base64 encoding:** The core malicious payload is encoded using Base64, making it difficult to understand at first glance.

2. **String obfuscation:** Strings are split into multiple parts and reconstructed using `codecs.decode` which makes reverse engineering more challenging.

3. **Eval and Compile:** The encoded payload is decoded using `base64.b64decode` then compiled and executed using `eval` and `compile`. This dynamic code execution prevents static analysis from fully understanding the malware's behavior.

4. **Import of malicious libraries:** It imports various libraries that may have capabilities related to network communication, file system manipulation, process management, and potentially other nefarious operations, such as those found in the `xbmc` library.

5. **Complex Logic:** The code contains many functions with intricate logic and nested loops and conditional statements, making it harder to trace the execution flow and identify critical actions.

The code appears to download and potentially execute multiple files, indicating a downloader or dropper component likely with additional functionality relating to persistence, possibly through registry entries, or network communication and potentially data exfiltration of files and other data based on the config values.

Function Summaries

* **`NoRedirection(urllib_error.HTTPError)`:** A custom exception handler for HTTP errors. It simply returns the `response` object. It doesn't actually handle the error in any meaningful way.

* **`add_log(string, level=xbmc.LOGDEBUG)`:** Logs a message with a specified level (default is debug). This is likely used for debugging during development but also for logging actions performed by the malware.

* **`makeRequest(url, headers=None)`:** Makes an HTTP request to a given URL. It handles headers, and attempts to decode the response based on the `content-type`. It uses Python's `urllib` library. Crucially, this handles UTF-8 and Latin-1 encoding of responses.

* **`getSources()`:** Attempts to read sources from a local JSON file (`source_file`). If the file doesn't exist, it defaults to a single hardcoded URL and a corresponding image URL for the main function. This also creates other file entries including a "Community Files" and "Search Other Plugins" sections.

* **`index()`:** Adds an entry called “FAVORITES” to the GUI, and retrieves data from the hardcoded "origin" url.

* **`addSource(url=None)`:** Adds a new source based on user settings or a provided URL. It attempts to parse metadata (title, thumbnail, etc.) from an XML file downloaded from the `source_url`. This is a key function for adding new sources to the malware's list. It also makes use of the xbmc library for keyboar functionality. The addition of a new source is communicated via xbmc's dialogs.

* **`rmSource(name)`:** Removes a source from the list based on the provided name. Note the lack of detailed error handling.

* **`getSoup(url, data=None)`:** Retrieves and parses the HTML content from the provided url using the `requests` library. It handles redirects and downloads the content, and accounts for various potential error conditions.

* **`processPyFunction(data)`:** Processes strings that start with `$pyFunction:`, treating them as Python code. This allows the malware to execute arbitrary Python code which would allow for substantial code expansion.

* **`getData(url, fanart, data=None)`:** Retrieves data from a URL, potentially used for fetching channel information. It uses the `getSoup` function to parse the HTML. Its key functionality involves parsing XML from the response and handling potential errors during XML parsing. This function does a lot of the heavy lifting for collecting metadata from various channels.

* **`getChanelItems(name, url, fanart)`:** Retrieves items (e.g., videos) for a specific channel. This function handles getting data from various content sources.

* **`getSubChanelItems(name, url, fanart)`:** Retrieves sub-channel items.

* **`getItems(items, fanart, dontLink=False)`:** Processes a list of items and adds them to the GUI. Handles various item types, including videos, and adds them with context menus where possible.

* **`getGoogleRecapchaResponse(captchakey, cj, type=1)`:** This is a function designed to solve a Google reCAPTCHA challenge, indicating that the malware attempts to bypass CAPTCHAs. This function is likely used to increase the obfuscation and survivability of the malware.

* **`getUrl(url, cookieJar=None, post=None, timeout=20, headers=None, noredir=False)`:** Makes an HTTP request, handling cookies, POST requests, timeouts, and redirects.

* **`get_decode(str, reg=None)`:** Decodes a string using URL decoding.

* **`javascryptUnescape(str)`:** Decodes a JavaScript-escaped string.

* **`askCaptcha(m, html_page, cookieJar)`:** Handles CAPTCHA challenges. This likely interacts with Google reCaptcha to obtain a token.

* **`askCaptchaNew(imageregex, html_page, cookieJar, m)`:** A newer version of `askCaptcha`, seemingly using a regex for the CAPTCHA image.

* **`addDir(name, url, mode, iconimage, fanart, description, genre, date, credits, showcontext=False, regexs=None, reg_url=None, allinfo={})`:** Adds an item to the XBMC GUI (Kodi). This is a crucial function for building the user interface, presenting videos and other resources to the user.

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

* **`urlsolver(url)`:** Solves URLs, potentially handling redirects or shortening services.

* **`tryplay(url, listitem, dialog=None)`:** Attempts to play a video URL in XBMC/Kodi, handling different playback methods.

* **`play_playlist(name, mu_playlist, queueVideo=None)`:** Plays a playlist of videos in XBMC/Kodi.

* **`download_file(name, url)`:** Attempts to download a file to the specified location. This contains (commented out) code that suggests it was designed to save files to a user-specified location. The lack of error handling makes the function somewhat brittle.

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

* `_unpack(p, a, c, k, e, d, iteration)`: This recursive function appears to unpack the encoded strings, possibly using a form of XOR or substitution cipher.

* `findAndReplaceWord(source_str, word_to_find, replace_with)`: This function searches for `word_to_find` within `source_str`. If found, it replaces it with `replace_with`.

* **`__itoa(num, radix)`:** Converts an integer to a string in a given radix (base).

* **`__itoaNew(cc, a)`:** A helper function for integer-to-string conversion, seemingly recursive and possibly used for obfuscation.

* **`getCookiesString(cookieJar)`:** Gets a string representation of cookies from a cookie jar.

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

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

* **`doEval(func_call, page_data, Cookie_Jar, m)`:** Evaluates Python code within the context of Kodi using `exec`. This is a very dangerous function, potentially allowing for code expansion and arbitrary file system access.

* **`doEvalFunction(func_call, page_data, Cookie_Jar, m)`:** Executes a function by dynamically generating Python code and executing it using `exec`. This is another function for dynamic code execution.

* **`import_by_string(full_name, filenamewithpath)`:** Imports a module dynamically. This enhances the flexibility of the malware and enables runtime adjustments of the script functionality.

* **`getGoogleRecapchaNew(imageregex, html_page, cookieJar, m)`:** Another version of reCAPTCHA solver.

* `takeInput(name, headname)`: Takes user input in a Kodi dialog box. The parameters are a label and the input name.

* `InputWindow(xbmcgui.WindowDialog)`: This is a class definition for creating a custom dialog box in Kodi.

* `getEpochTime()`, `getEpochTime2()`: Return the current time in epoch format (milliseconds and seconds, respectively).

* `get_params()`: Gets parameters from the XBMC/Kodi system.

* `getFavorites()`: Gets list of favorite items from a json file

* `addFavorite(name, url, iconimage, fanart, mode, playlist=None, regexs=None)`: Adds a new favorite item to a json file.

* `rmFavorite(name)`: Removes a favorite item from a json file.

Control Flow

The main execution flow involves:

1. **Initialization:** Variables are initialized, including paths and potentially sensitive data.

2. **Source Retrieval:** `getSources()` is called to obtain a list of video sources from a local file, or the default URL if none is found. The `source_file` content determines the source.

3. **Main Loop (Implicit):** While the program is running (not explicitly defined, but implied by the existence of functions like `getSources`, and the main `exec` call).

4. **Data Retrieval and Processing:** `getData` and other related functions are invoked to fetch and parse data from the sources which contain metadata and potential payloads that the malware executes.

5. **GUI Interaction:** The `addDir`, `play_playlist`, `getChanelItems`, `getSubChanelItems` functions populate the XBMC GUI.

6. **Dynamic Execution:** The Base64-encoded code (`trust`) is decoded, compiled, and executed. This is where the core malicious actions are likely performed.

Data Structures

* **Lists:** Used extensively to store URLs, video metadata, cookies.
* **Dictionaries:** Used to store metadata associated with videos and other resources.
* **Cookie Jar:** A data structure for handling cookies, enabling persistent authentication with various servers.
* **JSON:** Used for storing and retrieving configuration data such as favorite items and source details.
* **XML:** Used to parse metadata from some sources.

Malware Family Suggestion

Based on the functionality observed in the code, this malware strongly resembles a **video streaming malware or a Kodi addon dropper**. Its main goal seems to be expanding the functionality of Kodi by adding new video sources, many of which appear to point to sources that may not be legitimate, thus suggesting that this malware is designed to allow for the streaming of pirated content, potentially containing additional malware payloads. The presence of Google reCAPTCHA bypass attempts and the extensive use of obfuscation techniques confirm this behavior. Further investigation into the dynamically loaded code would be necessary to confirm the extent of the malicious capabilities. The use of a config system also suggests that this type of malware may be capable of receiving updates or new payloads dynamically, further complicating analysis and impacting the ability to fully prevent infection.