

# Malware Analysis Report



MALWARE bazaar  
by ABUSE™

**Subject:** Silent Intrusion: Shadows in Memory

**Malware Sample Source:** Malware Bazaar

[Sample Download link](#)

**Date:** October 3, 2024

**Made By**

**LinkedIn:** [Engineer.Ahmed Mansour](#)

**GitHub link**

## **Table of Contents**

<b>Malware Analysis report</b>	<b>1</b>
<b>Table of Contents</b>	<b>2</b>
<b>Steps to get the main code</b>	<b>3</b>
<b>Download the malware sample from malwarebazaar.com using Kali Linux.</b>	<b>4</b>
<b>Get SHA-256 hash of the malware file on Kali for identification and reference.</b>	<b>5</b>
<b>Conduct Threat Intelligence research on the obtained SHA-256 hash to gather detailed information that will aid in the malware analysis process.</b>	<b>7</b>
<b>Utilize the Ghidra tool to disassemble and analyze the malware's code.</b>	<b>13</b>
<b>Perform a comprehensive analysis of the file's code to identify malicious behavior, functions, or potential threats.</b>	<b>14</b>
<b>Utilize "Detect It Easy" to extract detailed information regarding the malware's characteristics and structure.</b>	<b>65</b>
<b>Additional Information's</b>	<b>66</b>
<b>Develop tailored recommendations for each department within the organization based on the findings to mitigate future risks.</b>	<b>71</b>
<b>Conclusion</b>	<b>74</b>

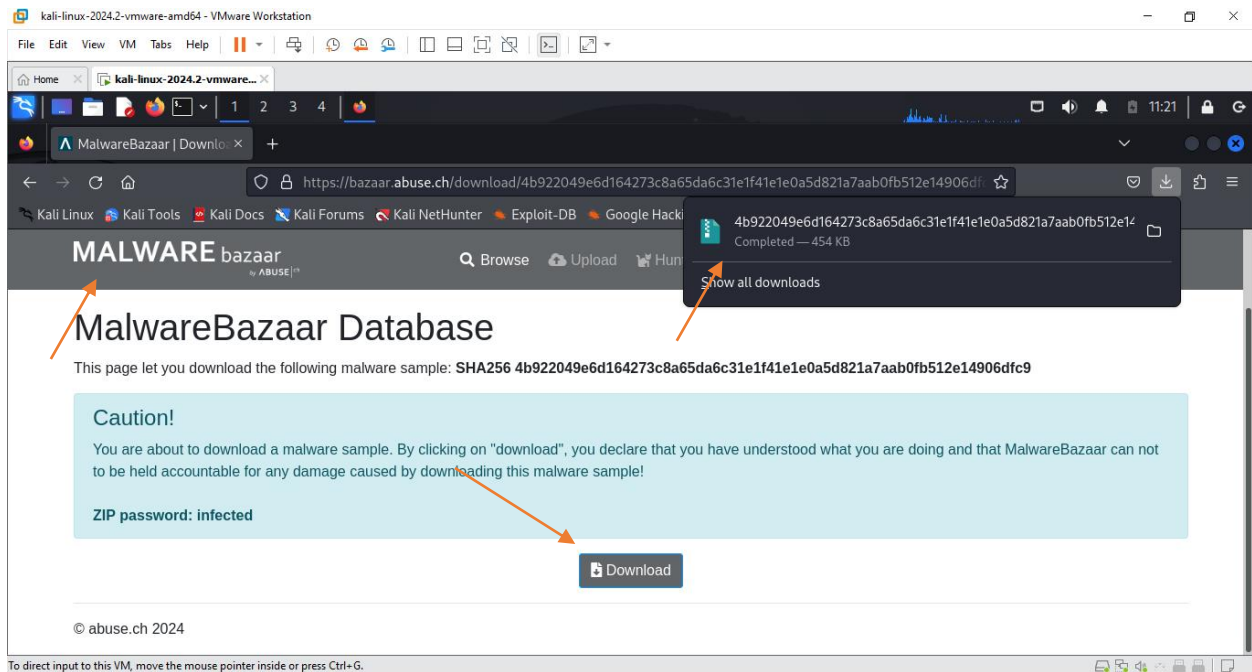
**Steps to get the main code:** I will be sharing a detailed, step-by-step breakdown of my approach to analyzing this malware. Stay tuned for insights on the investigation process and containment strategies.

1	Download the malware sample from <a href="https://malwarebazaar.com">malwarebazaar.com</a> using Kali Linux.
2	Get SHA-256 hash of the malware file on Kali for identification and reference.
3	Conduct Threat Intelligence research on the obtained SHA-256 hash to gather detailed information that will aid in the malware analysis process.
4	Utilize the Ghidra tool to disassemble and analyze the malware's code.
5	Perform a comprehensive analysis of the file's code to identify malicious behavior, functions, or potential threats.
6	Utilize "Detect It Easy" to extract detailed information regarding the malware's characteristics and structure.
7	Additional Information's
8	Develop tailored recommendations for each department within the organization based on the findings to mitigate future risks.
9	Conclusion

## Step 1: Download the malware sample from malwarebazaar.com using Kali Linux.

We will navigate to the **Ghidra** directory on Kali Linux through the terminal and execute the tool. Once Ghidra is running, we will import the malware sample to initiate our analysis.

Please refer to the attached photo for further details.



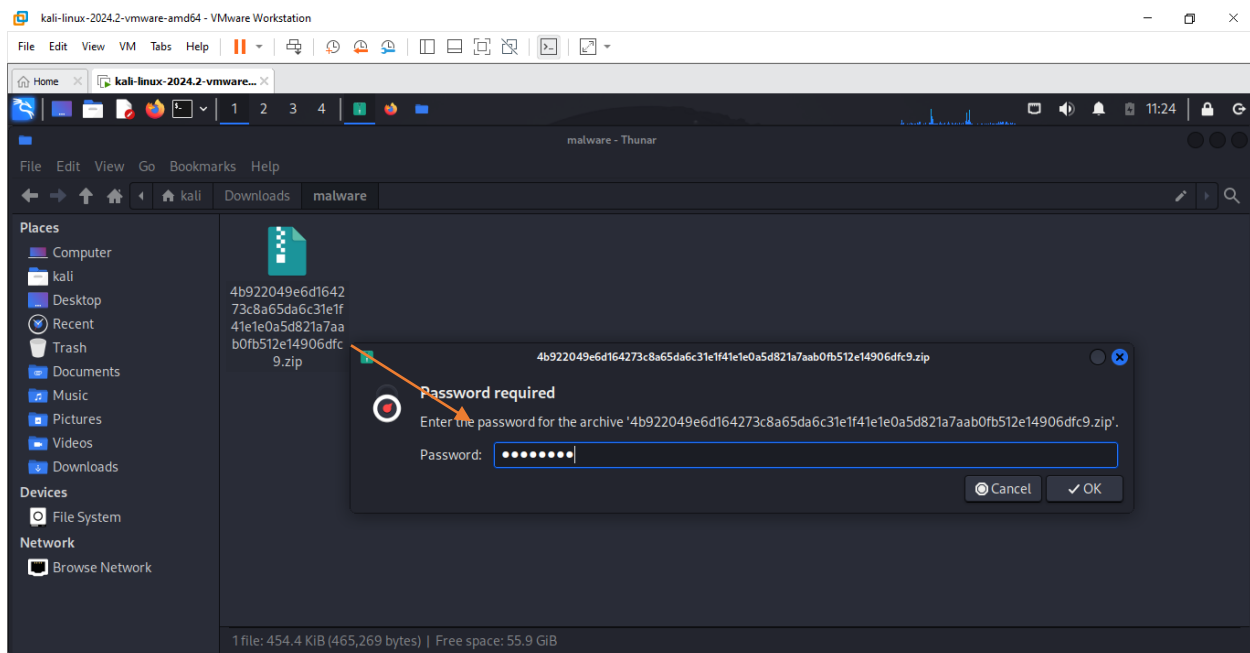
Here is the link to the [malware sample](https://bazaar.abuse.ch/download/4b922049e6d164273c8a65da6c31e1f41e1e0a5d821a7aab0fb512e14906dfc9) for reference.

## Step 2: Get **SHA-256 hash** of the malware file on Kali for identification and reference.

We created a new directory named **Malwares** within the Downloads folder to organize our analysis.

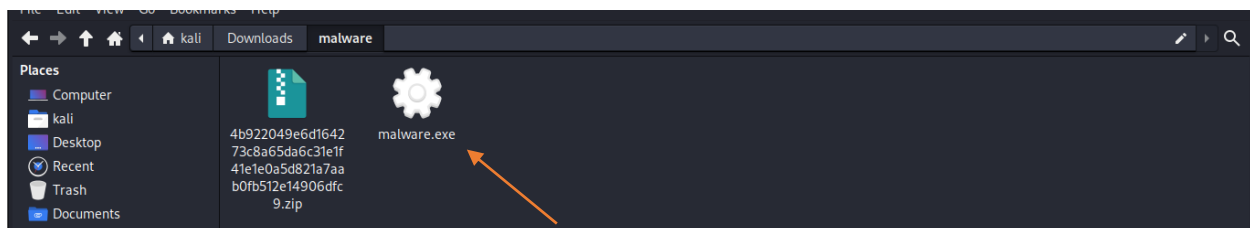
The malware sample was then unzipped using the password "**infected**", as specified on the website.

Please refer to the attached file for additional details.



After successfully unzipping the file, we retrieved the malware sample, named "**malware**", ready for further analysis.

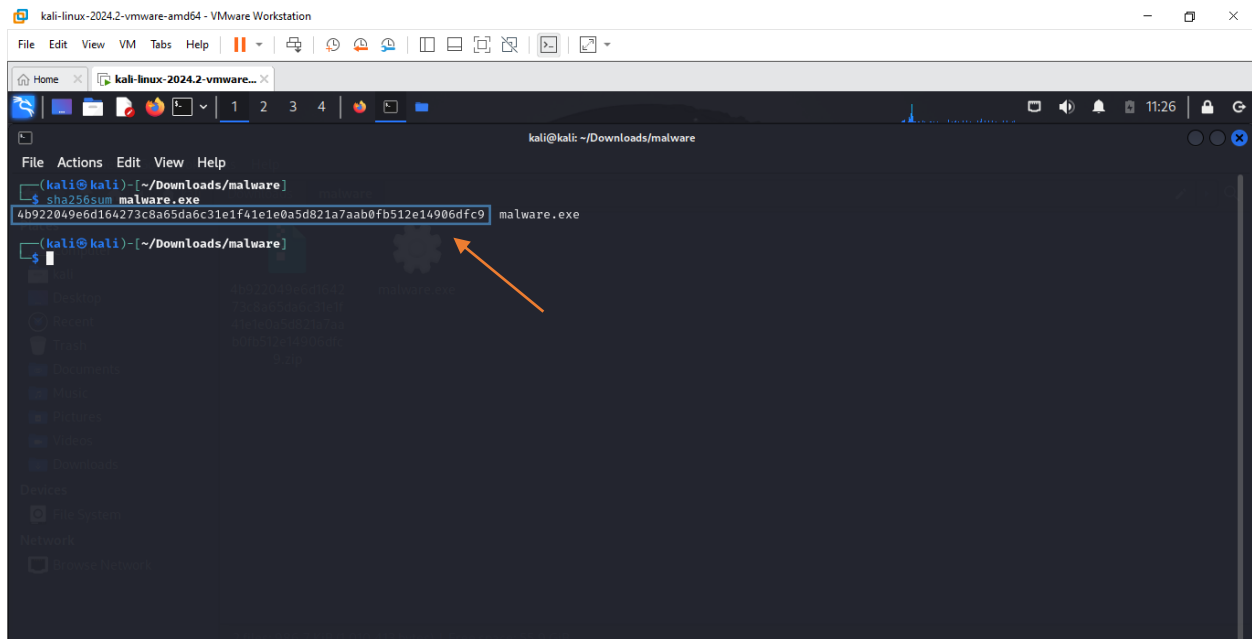
Please refer to the attached file for additional details.



We right-clicked in the directory and selected "**Open Terminal Here**" to launch the terminal in the correct location.

Next, we executed the following command to obtain the **SHA-256 hash** of the malware sample:

**sha256sum malware.exe**



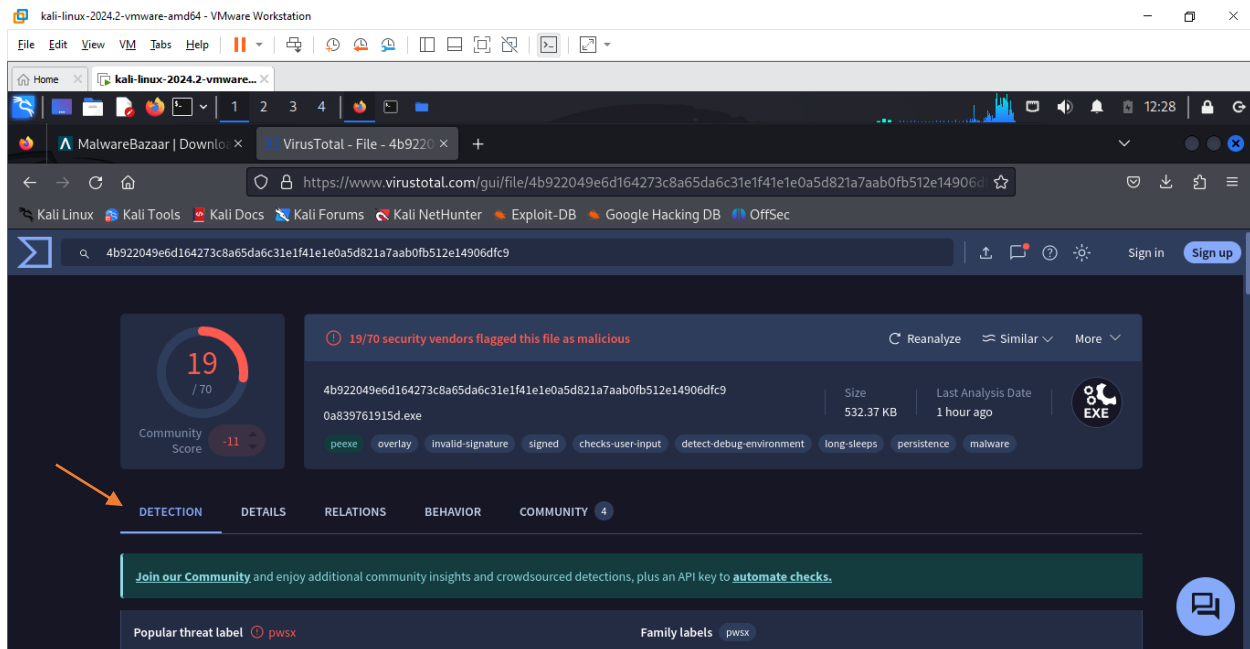
The screenshot shows a terminal window titled 'kali@kali: ~/Downloads/malware'. The command 'sha256sum malware.exe' has been executed, and the output is displayed: '4b922049e6d164273c8a65da6c31e1f41e1e0a5d821a7aab0fb512e14906dfc9 malware.exe'. An orange arrow points to the output hash. The terminal window is part of a VMware Workstation interface, with the top bar showing 'kali-linux-2024.2-vmware-amd64 - VMware Workstation'.

```
kali@kali: ~/Downloads/malware
$ sha256sum malware.exe
4b922049e6d164273c8a65da6c31e1f41e1e0a5d821a7aab0fb512e14906dfc9 malware.exe
$
```

The **SHA-256 hash** is :

4b922049e6d164273c8a65da6c31e1f41e1e0a5d821a7aab0fb5  
12e14906dfc9

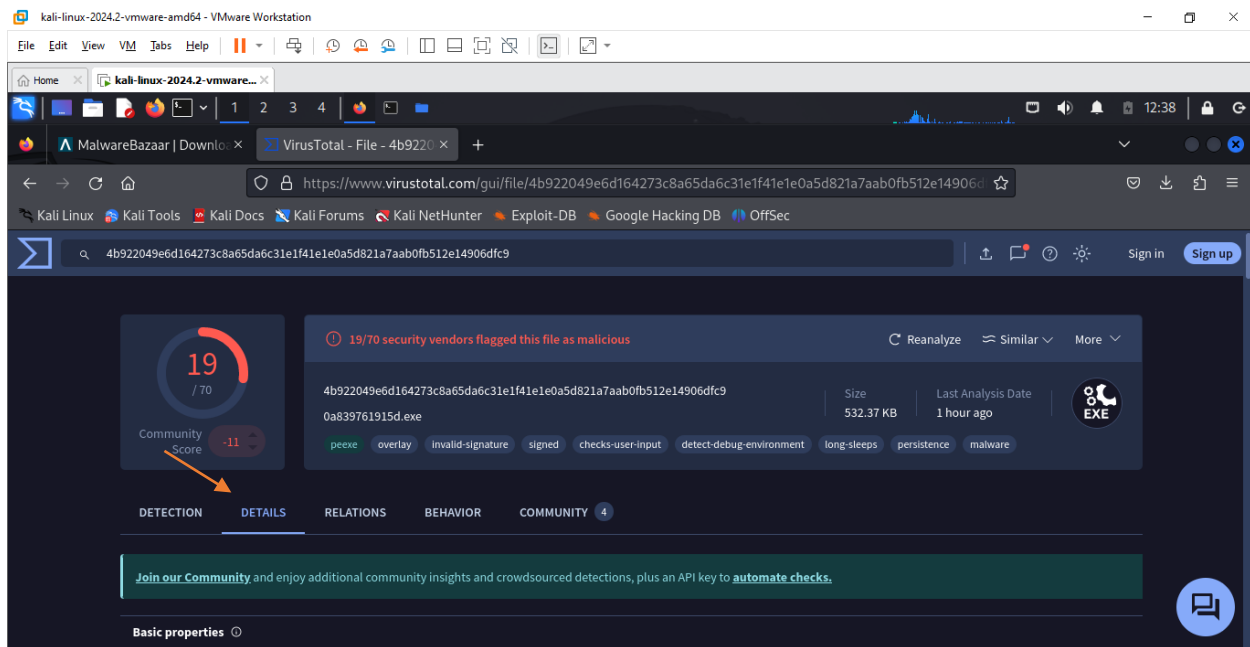
Step 3: Conduct **Threat Intelligence** research on the obtained SHA-256 hash to gather detailed information that will aid in the malware analysis process.



## Detection Section

- **Security Vendor Detection:** Out of 70 security vendors, 19 have flagged this file as malicious.
- **Popular Threat Label:** *pwsx*
- **Threat Categories:** Not Available
- **Malware Family:** *pwsx*

## [VirusTotal Detection Results](#)



## File Details Section

### Basic Properties

- **MD5:** ca0a0941ffa6d17a014540d6dc4b36ce
- **SHA-1:** 307fc74113acf6b63548272d199e6d6fab2c7fb0
- **SHA-256:** 4b922049e6d164273c8a65da6c31e1f41e1e0a5d821a7aab0fb512e14906dfc9
- **Vhash:** 055056655d75556az4anz1fz
- **Authentihash:** 2566cff1b1f051b3424a401f546457e317ed455c48e30fee7aca42ae725e33b1
- **Imphash:** 1186293b831ff45d8016d71d51f87333
- **Rich PE Header Hash:** ed770645d488f44d776ae140a2fa8acf
- **SSDEEP:** 12288

/YegT95OzD/GK4TG3dJiavE4XVyDqnBJ4DPv

- **TLSH:**  
T19CC40111B5D08471D93315320AE4D6B5AE7EFD710EA25DDF27A80B7F0F30680EA225AB

### File Type Information

- **File Type:** Win32 EXE (Windows executable)
- **Tags:** windows, win32, pe, peexe
- **Compiler:** Microsoft Visual C/C++ (2017 v.15.5-6) [EXE32], Microsoft Visual C/C++ (19.36.33523) [LTCG/C++]
- **File Size:** 532.37 KB (545,144 bytes)



### *File History*

- **Creation Time:** 2024-10-03 13:56:51 UTC
- **Signature Date:** 2023-03-13 08:49:00 UTC
- **First Submission:** 2024-10-03 15:20:07 UTC
- **Last Submission:** 2024-10-03 15:20:07 UTC
- **Last Analysis:** 2024-10-03 15:20:07 UTC

### *File Name*

- **Detected Name:** 0a839761915d.exe
- 

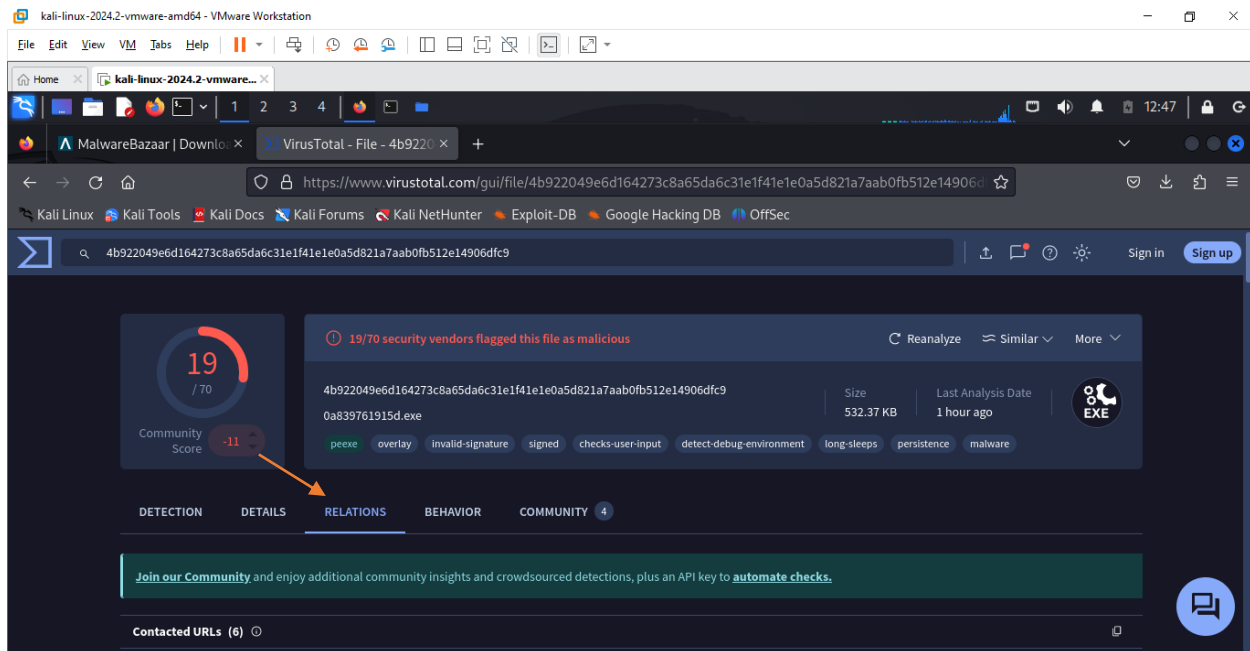
## **Portable Executable (PE) Info**

- **Compiler Products:**
  - [C] VS2022 v17.9.0 pre 1.0 build 33218 (count=18)
  - [ASM] VS2022 v17.9.0 pre 1.0 build 33218 (count=20)
  - Unmarked objects (count=89)
  - [C++] VS2022 v17.9.0 pre 1.0 build 33218 (count=78)
- **Header Information:**
  - **Target Machine:** Intel 386 or later processors and compatible processors
  - **Compilation Timestamp:** 2024-10-03 13:56:51 UTC
  - **Entry Point:** 28530
  - **Contained Sections:** 5

### *Imports*

- **USER32.dll**
- **KERNEL32.dll**

### [VirusTotal Details Results](#)



## Relations Section

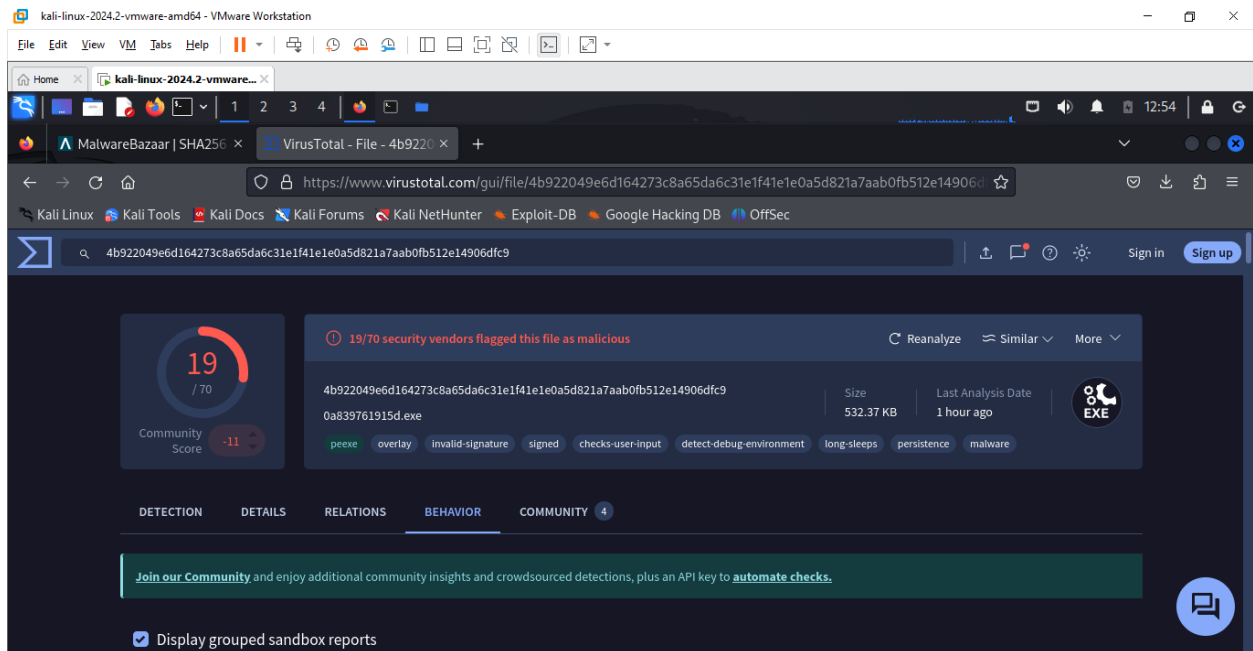
### *Contacted Domains*

1. SoftShipment.exe
2. [Desktop Explorer.exe](#)
3. <https://abundanttyj.site/api>
4. <https://steamcommunity.com/profiles/76561199724331900>
5. [Microsoft URL](#)
6. [SoftShipment.exe](#)

#### *Contacted IP Addresses*

- 104.192.142.24
- 104.192.142.25
- 104.192.142.26
- 104.21.13.106
- 142.250.98.100
- 142.250.98.101
- 142.250.98.102
- 142.250.98.113
- 142.250.98.138
- 142.250.98.139
- 150.171.28.10
- 152.195.19.97
- 172.217.204.100
- 172.217.204.101
- 172.217.204.102
- 172.217.204.113
- 172.217.204.138
- 172.217.204.139
- 172.67.199.213
- 192.168.0.27
- 20.223.35.26
- 20.99.133.109
- 20.99.185.48
- 224.0.0.251
- 23.195.238.96
- 23.207.106.113
- 23.209.125.25
- 23.214.234.105
- 23.216.147.76
- 23.216.81.152
- 3.5.20.23
- 3.5.27.205
- 3.5.28.57
- 3.5.30.204
- 3.5.30.59
- 3.5.8.118
- 52.217.126.225
- 52.217.134.225
- 74.125.134.94
- 74.125.141.84
- 84.201.211.20

#### [VirusTotal Relations Results](#)



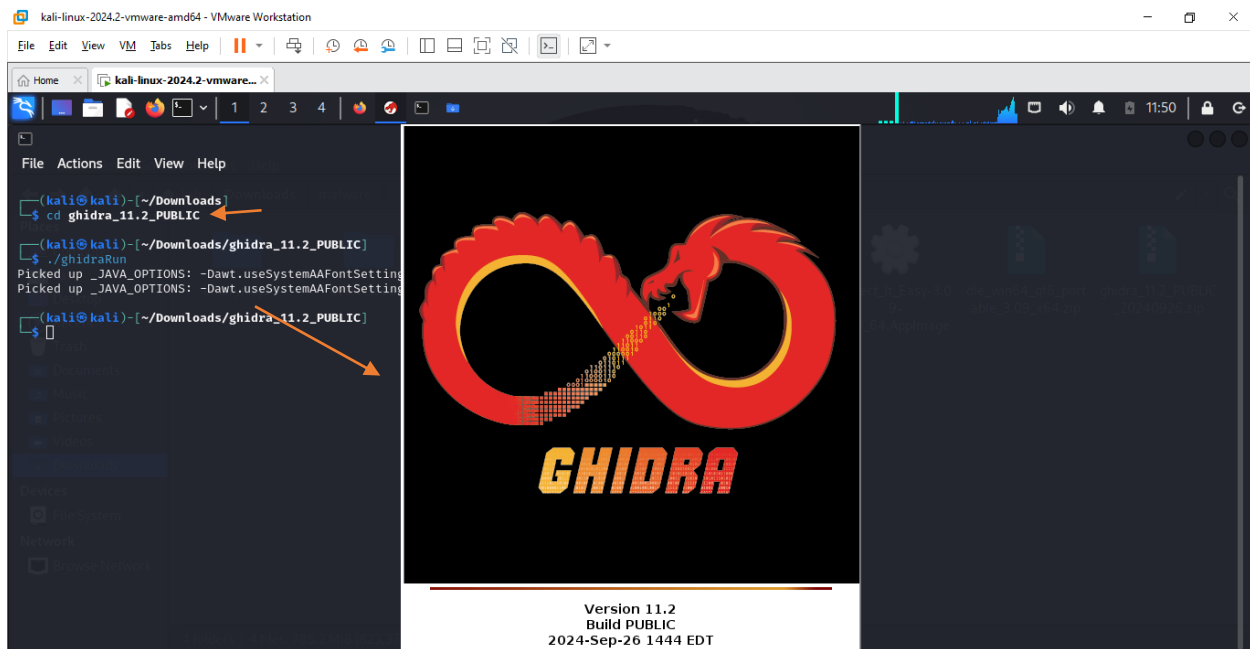
## Behavior Section

### [VirusTotal Behavior Results](#)

Step 4: Utilize the **Ghidra** tool to disassemble and analyze the malware's code.

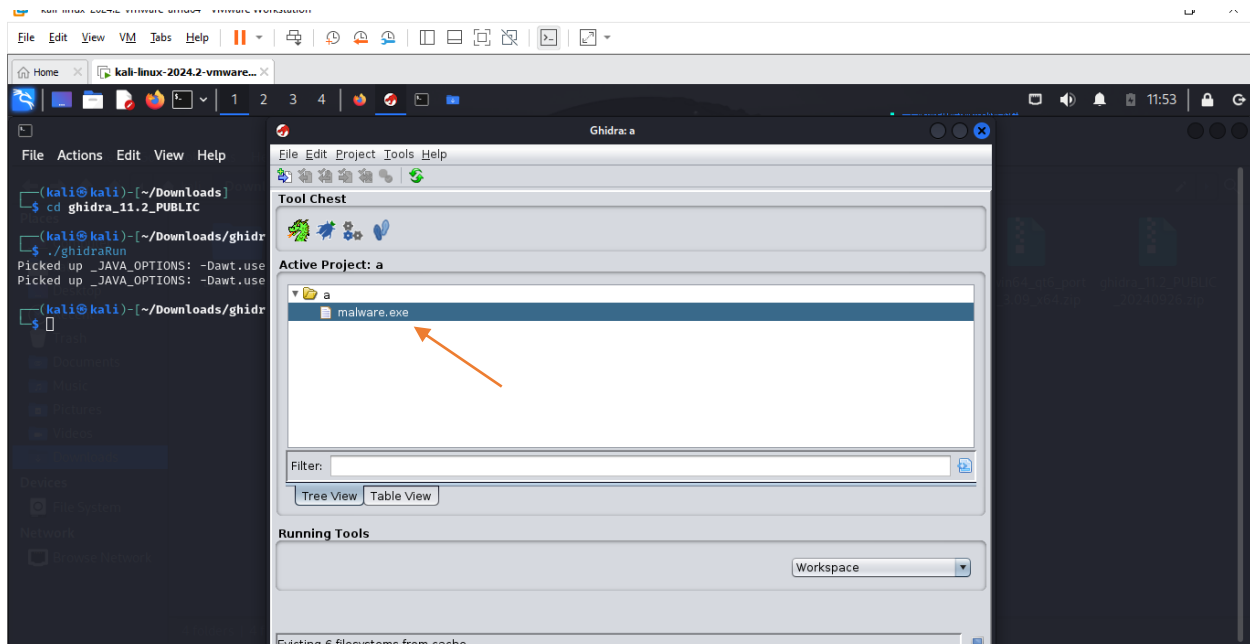
We will navigate to the **Ghidra** directory on Kali Linux through the terminal and execute the tool. Once Ghidra is running, we will import the malware sample to initiate our analysis.

Please refer to the attached photo for further details.



After launching the **malware.exe** file, we proceeded to begin the in-depth analysis of its behavior and code using the appropriate tools.

Please refer to the attached photo for further details.

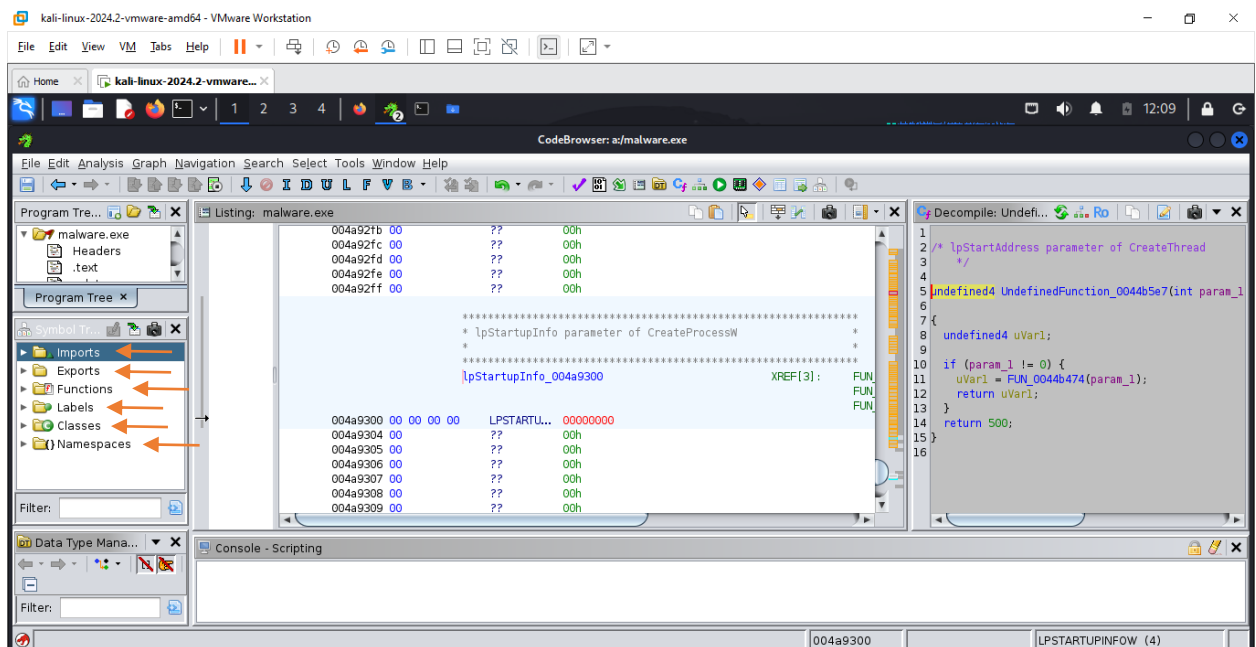


## Step 5: Perform a comprehensive analysis of the file's code to identify malicious behavior, functions, or potential threats.

Upon an initial inspection of the malware, we observe six key components within the structure:

1. **Imports** – Lists the external libraries and functions the malware relies on.
2. **Exports** – Displays the functions that this malware exposes for external use.
3. **Functions** – Contains all internal functions defined in the malware, which could hold malicious code.
4. **Labels** – Represents named addresses within the code, useful for identifying key points or variables.
5. **Classes** – Contains object-oriented structures, indicating if the malware utilizes classes or objects.
6. **{ }Namespaces** – Organizes code elements and prevents naming conflicts, giving insight into how the code is structured.

Please refer to the attached photo for further details.



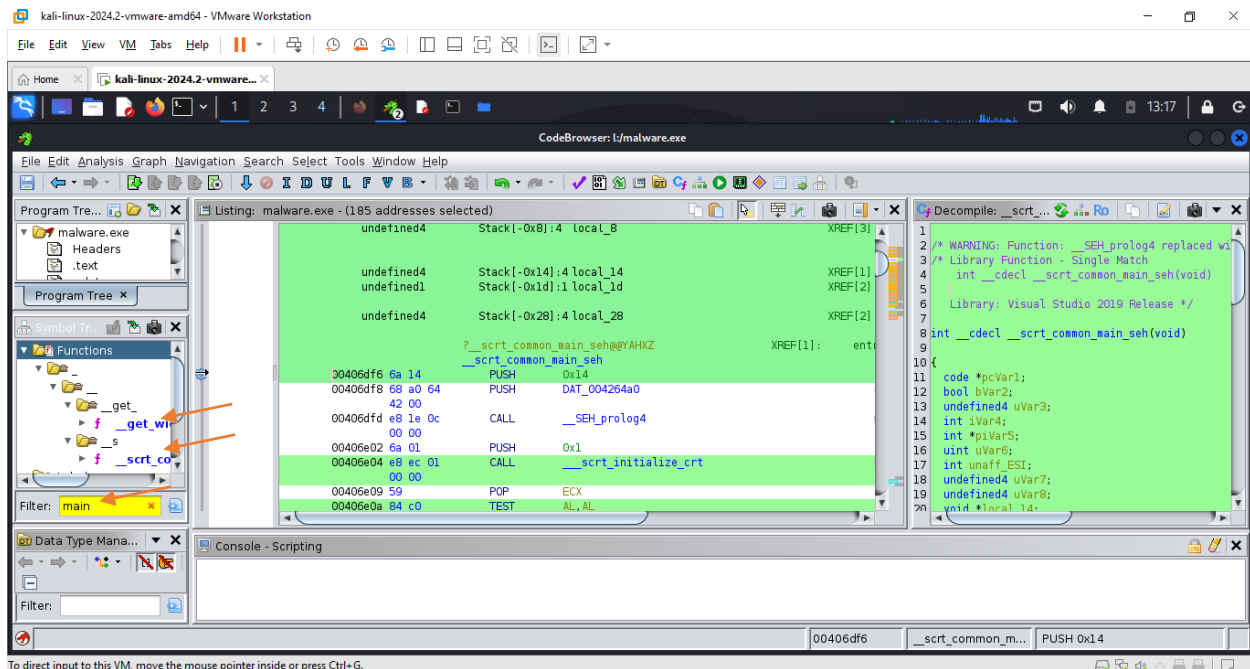
We will begin our analysis by utilizing the **filter box** in Ghidra to search for key functions that may lead us to the main code. Specifically, we will search using the following keywords:

- A. **main**
- B. **Win**
- C. **start**
- D. **LoadLibrary**
- E. **GetProcAddress**
- F. **CreateProcess**
- G. **VirtualAlloc/VirtualProtect**
- H. **Memory**
- I. **ShellExecute/WinExec**
- J. **recv/send**
- K. **strstr/strcmp**
- L. **Registry Functions**
- M. **Mutex**

We will begin our analysis by searching for the **main** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 2 function codes relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **Main** keyword search in the filter is as follows:

**Download link for code 1 & 2**



## Code 1 & Code 2: `__get_wide_winmain_command_line`

---

### 1. Function Purpose and Logic

#### Purpose:

This function retrieves the command line used to execute a Windows application and returns it as a wide-character string.

#### Logic Breakdown:

- It first checks if `DAT_0048367c` is null. If true, it assigns a default value to `puVar2` from `DAT_0048324c`.
- Then, it iterates through the command line characters pointed to by `puVar2`. The function skips over characters until it reaches a space (`0x20`) or a double-quote (`0x22`).
- The use of a Boolean `bVar1` flag controls whether the function is inside a quoted argument.
- If the function reaches the end of the command line or an empty string, it returns.

This logic is designed to handle command-line arguments properly, ensuring quoted parts are treated as single arguments.

#### Proof from Code:

- The check for a null pointer at the start (`if (DAT_0048367c == (ushort *)0x0)`) ensures the command line is properly fetched.
  - The use of `bVar1` to toggle behavior when encountering a double-quote character (`if (uVar3 == 0x22)`) demonstrates control of string parsing.
- 

### 2. API Calls

No explicit API calls are present in the function, as it seems to be a low-level string parsing routine, likely invoked by higher-level system API calls but none directly visible in this code.

#### Proof from Code:

- No system API calls like `WinAPI` or external libraries are invoked directly in this snippet.
-

### 3. Control Flow

The control flow in this code consists of basic loop constructs and conditional checks:

- The function enters a `do-while` loop to iterate through the characters of the command line.
- Conditional checks are made to decide whether to continue iterating (`uVar3 < 0x21, uVar3 == 0x22`).
- A nested `do-while` inside the loop skips over non-printable characters.

#### Proof from Code:

- `do-while` loop controlling flow (`do { ... } while (true)`).
  - Conditional statements based on character values (`if (uVar3 < 0x21)` and `if (uVar3 == 0x22)`).
- 

### 4. Data Handling and Manipulation

Yes, this function handles and manipulates data. It reads a wide-character command line string and processes each character to detect arguments, ignoring spaces outside of quoted text.

#### Proof from Code:

- The pointer `puVar2` iterates over the string, and characters are compared against certain values (e.g., space `0x20` or double-quote `0x22`).
  - Manipulation occurs when the function skips over characters (`puVar2 = puVar2 + 1`).
- 

### 5. Obfuscation and Evasion Techniques

No explicit obfuscation or evasion techniques are present in this code. The function seems straightforward in its approach to parsing a command line string. However, the function could be used as part of a larger malicious program that uses legitimate-looking code for obfuscation.

#### Proof from Code:

- There is no encryption, string manipulation, or use of hidden APIs, which are typical obfuscation techniques.
-

## 6. Malicious Behavior Indicators

The function itself is not inherently malicious. However, it could be part of a malware's routine to retrieve command-line arguments for malicious purposes, such as executing specific payloads based on passed parameters.

### Proof from Code:

- The function's behavior is benign on its own. However, the context in which it is used (e.g., if the command line includes parameters to run malicious code) could be indicative of its use in a malware.
- 

## 7. Indicators of Compromise (IoCs)

Since this is a simple command-line parsing function, no explicit IoCs (like file paths, IP addresses, or malware hashes) are directly present in this code. IoCs would come from how the malware uses this function, for example, to launch commands that might indicate malicious activity.

### Proof from Code:

- No visible IoCs in the code snippet provided.
- 

## 8. Attack Vector

This function could be part of a larger attack vector involving command-line execution. If the attacker gains access to a system, they could use this function to parse specific command-line arguments passed to the malware for different payloads or behaviors.

### Proof from Code:

- The function parses command-line arguments, which are typically used in malware to control execution flow (e.g., specifying whether to run in silent mode, trigger persistence, etc.).
-

## 9. Persistence Mechanisms

There is no persistence mechanism directly visible in this function. However, in a larger malware context, this function could be used to persist by accepting specific command-line arguments related to maintaining persistence (e.g., arguments that install the malware as a service or create a scheduled task).

### Proof from Code:

- No direct signs of persistence mechanisms (e.g., registry changes, scheduled tasks).
- 

## 10. Command and Control (C2) Communication

This function does not exhibit any signs of communication with a command-and-control (C2) server. It is strictly a command-line argument parsing routine, and C2 activity would likely occur in another part of the malware's codebase.

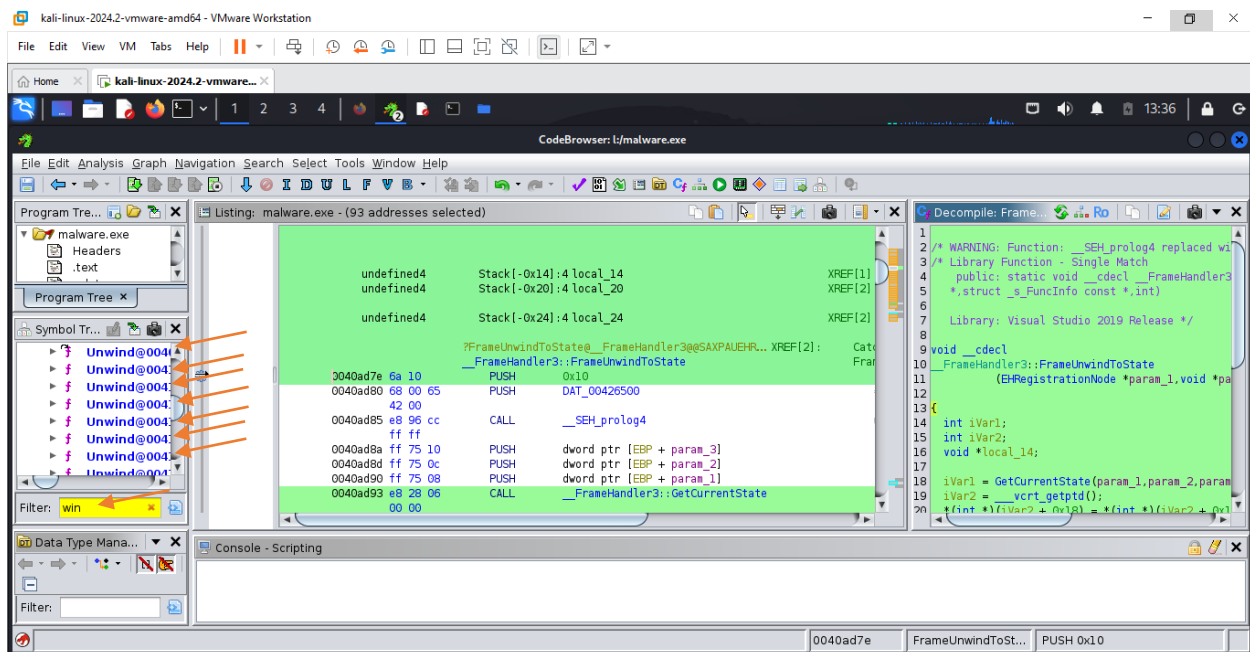
### Proof from Code:

- No network activity or external communication methods are evident in this function.

We will begin our analysis by searching for the **win** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 18 function codes relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **win** keyword search in the filter is as follows:

**Download link for code from 1 to 18**

## Code 1: @\_EH4\_GlobalUnwind2@8

### 1. Purpose and Logic:

- This function performs global stack unwinding after an exception. It uses the `RtlUnwind` function to unwind the stack to a stable state after an error occurs.
- **Proof:** The use of `RtlUnwind(param_1, (PVOID)0x40b0f5, param_2, (PVOID)0x0);`.

### 2. API Calls:

- `RtlUnwind` is used to unwind the stack during structured exception handling (SEH).

### 3. Control Flow:

- The flow of control directly uses the `RtlUnwind` function and then exits without any further processing.

### 4. Data Handling and Manipulation:

- The function handles stack frames and exception records.

### 5. Obfuscation and Evasion Techniques:

- No explicit obfuscation, but the use of structured exception handling (SEH) can allow malware to recover from errors and continue execution.

### 6. Malicious Behavior Indicators:

- Malicious programs often use SEH to prevent crashing and to recover from system exceptions.

### 7. IoCs:

- No specific IoCs in this function.

### 8. Attack Vector:

- None specific to this code.

### 9. Persistence Mechanisms:

- None.

### 10. C2 Communication:

- None.
-

## Code 2: @\_EH4\_LocalUnwind@16

1. **Purpose and Logic:**
    - This function performs local stack unwinding, dealing with specific frames during exception handling.
    - **Proof:** It calls FUN\_0040afb0 to manage the unwinding process.
  2. **API Calls:**
    - None visible, but the function indirectly manages exception handling.
  3. **Control Flow:**
    - A straightforward function that calls another function to manage the unwinding process.
  4. **Data Handling and Manipulation:**
    - Manipulates stack frame data during the unwinding process.
  5. **Obfuscation and Evasion Techniques:**
    - No explicit obfuscation. Exception handling could be used to evade crashes.
  6. **Malicious Behavior Indicators:**
    - SEH is commonly used in malware for error recovery.
  7. **IoCs:**
    - None visible.
  8. **Attack Vector:**
    - None.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

### Code 3: `__acrt_uninitialize_winapi_thunks`

1. **Purpose and Logic:**

- This function uninitializes API thunks, which are pointers to dynamically loaded APIs.
- **Proof:** It iterates through `hLibModule_00483288` and calls `FreeLibrary` to unload any loaded libraries.

2. **API Calls:**

- `FreeLibrary` is used to unload dynamically loaded libraries.

3. **Control Flow:**

- Loops through loaded DLLs, unloading each one.

4. **Data Handling and Manipulation:**

- Manages pointers to dynamically loaded libraries.

5. **Obfuscation and Evasion Techniques:**

- Unloading libraries could be a way to hide evidence after malicious actions.

6. **Malicious Behavior Indicators:**

- Dynamic unloading of DLLs may indicate an attempt to clean up traces.

7. **IoCs:**

- Suspicious DLL loading/unloading activity.

8. **Attack Vector:**

- Likely part of post-execution cleanup.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-



## Code 4: `__FrameUnwindFilter`

1. **Purpose and Logic:**
    - Handles unwinding stack frames during an exception based on specific error codes.
    - **Proof:** The function checks for specific error codes like `-0x1fbcbae`.
  2. **API Calls:**
    - None directly visible.
  3. **Control Flow:**
    - Control flow is based on error codes, potentially leading to program termination or adjustments in the stack.
  4. **Data Handling and Manipulation:**
    - Handles stack frame data.
  5. **Obfuscation and Evasion Techniques:**
    - No explicit obfuscation, but error-handling can obscure program behavior.
  6. **Malicious Behavior Indicators:**
    - SEH techniques used in malware for stability.
  7. **IoCs:**
    - None.
  8. **Attack Vector:**
    - None.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

## Code 5: `__srt_get_show_window_mode`

1. **Purpose and Logic:**
    - Retrieves the show window mode based on startup information.
    - **Proof:** It uses `GetStartupInfoW` to check the `dwFlags` and `wShowWindow` values.
  2. **API Calls:**
    - `GetStartupInfoW` retrieves the startup information of the current process.
  3. **Control Flow:**
    - The function returns the `wShowWindow` value if the flag is set.
  4. **Data Handling and Manipulation:**
    - Reads process startup data.
  5. **Obfuscation and Evasion Techniques:**
    - No direct obfuscation, but window mode manipulation can hide GUI windows.
  6. **Malicious Behavior Indicators:**
    - Malware may use this to run in a hidden window mode.
  7. **IoCs:**
    - Hidden windows or minimized processes.
  8. **Attack Vector:**
    - Could be used to make malicious programs invisible.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

## Code 6: `__get_wide_winmain_command_line`

### 1. Purpose and Logic:

- Parses the command line for a Windows application, managing quoted arguments.
- **Proof:** The function iterates through `DAT_0048367c` to process command line arguments.

### 2. API Calls:

- No API calls, just internal command line parsing.

### 3. Control Flow:

- A loop is used to iterate through the command line characters.

### 4. Data Handling and Manipulation:

- Handles command line arguments, including managing quoted text.

### 5. Obfuscation and Evasion Techniques:

- No obfuscation, but command line arguments may be manipulated to pass parameters to malware.

### 6. Malicious Behavior Indicators:

- Could be used to launch malware with specific arguments.

### 7. IoCs:

- None visible directly.

### 8. Attack Vector:

- Command line-based attacks.

### 9. Persistence Mechanisms:

- None.

### 10. C2 Communication:

- None.
-

## Code 7: `_UnwindNestedFrames`

1. **Purpose and Logic:**
    - Unwinds nested exception frames after an error.
    - **Proof:** Uses `RtlUnwind` to clean up the stack.
  2. **API Calls:**
    - `RtlUnwind` for stack unwinding.
  3. **Control Flow:**
    - Sequential function with calls to `RtlUnwind` and cleanup.
  4. **Data Handling and Manipulation:**
    - Manages stack and exception records.
  5. **Obfuscation and Evasion Techniques:**
    - None visible.
  6. **Malicious Behavior Indicators:**
    - SEH techniques could be used in malware.
  7. **IoCs:**
    - None.
  8. **Attack Vector:**
    - None.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

## Codes 8-18 (Unwind Functions)

### 1. Purpose and Logic:

- These functions handle different aspects of stack unwinding, often manipulating exception records and stack pointers.
- **Proof:** Functions like `FUN_00406d00` and `FUN_00403a72` handle stack manipulation.

### 2. API Calls:

- No direct API calls visible, except for internal stack management routines.

### 3. Control Flow:

- Each function either adjusts or restores stack pointers, ensuring the stack is correctly cleaned up after an exception.

### 4. Data Handling and Manipulation:

- Manipulate exception records, stack pointers, and frame data.

### 5. Obfuscation and Evasion Techniques:

- These functions do not explicitly show obfuscation, but stack manipulation can hide malicious behavior.

### 6. Malicious Behavior Indicators:

- Complex stack unwinding could be part of error recovery in malware.

### 7. IoCs:

- None visible directly, though stack manipulation could indicate malicious intent in context.

### 8. Attack Vector:

- None specific to these functions.

### 9. Persistence Mechanisms:

- None.

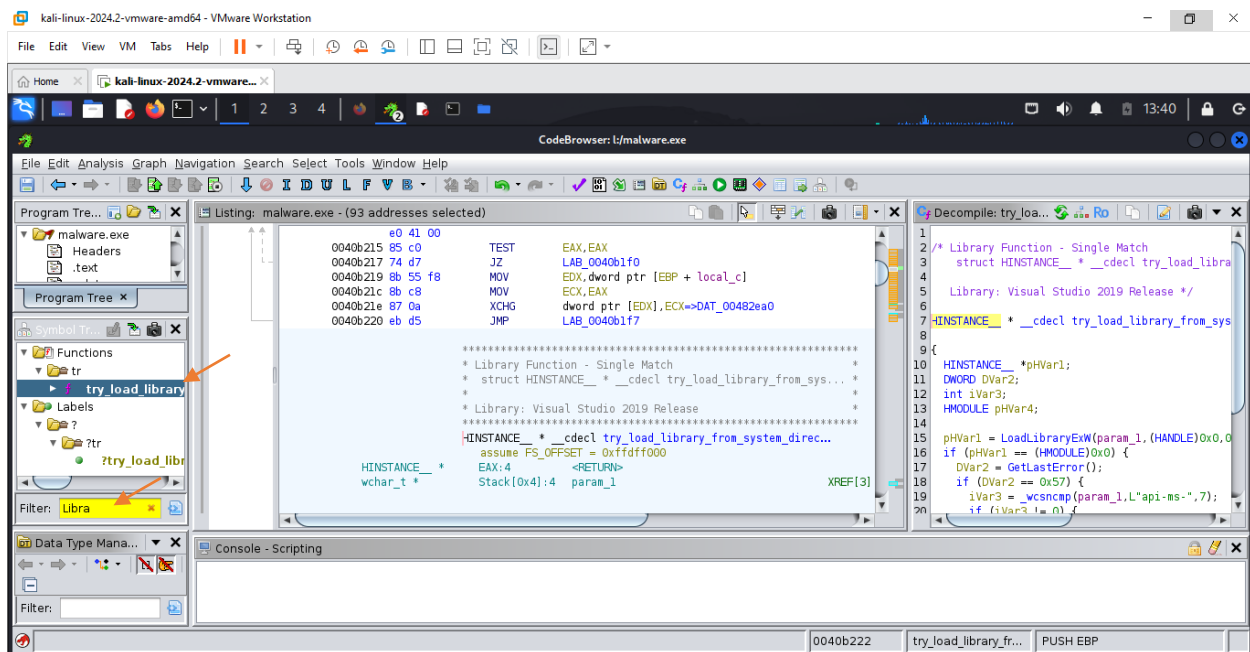
### 10. C2 Communication:

- None.

We will begin our analysis by searching for the **Library** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 1 function code relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **Library** keyword search in the filter is as follows:

**Download link for discovered code**

## 1. Function Purpose and Logic

- **Purpose:**  
The function attempts to load a library (DLL) from the system directory using the `LoadLibraryExW` function with specific flags. If the initial load fails, it checks the error code. If the error is `0x57` (indicating an invalid parameter), the function retries without additional flags if the library's name does not start with "api-ms-". The primary purpose is to load a system library into the program's memory.
- **Logic:**
  - First, the function tries to load the library with a specific flag (`0x800`, which is `LOAD_LIBRARY_SEARCH_SYSTEM32`).
  - If the load fails, the function retrieves the last error using `GetLastError`. If the error is `0x57`, it checks if the library name starts with "api-ms-". If it does not, the function retries loading the library without flags.

### Proof from Code:

- `pHVar1 = LoadLibraryExW(param_1, (HANDLE)0x0, 0x800);` (initial load attempt).
  - `DVar2 = GetLastError();` (error retrieval).
  - `if (iVar3 != 0) { pHVar4 = LoadLibraryExW(param_1, (HANDLE)0x0, 0); }` (retry logic).
- 

## 2. API Calls

- **API Call:**
  - `LoadLibraryExW`: Attempts to load a DLL from the system directory.
  - `GetLastError`: Retrieves the last error code generated by the `LoadLibraryExW` function.

### Proof from Code:

- `pHVar1 = LoadLibraryExW(param_1, (HANDLE)0x0, 0x800);` (attempt to load library).
  - `DVar2 = GetLastError();` (retrieves error after failure).
-

### 3. Control Flow

- The function follows a basic conditional flow:
  - It tries to load a library with `LoadLibraryExW`.
  - If this fails, it checks the last error code.
  - If the error is `0x57`, the function checks if the library name starts with "api-ms-". If not, it tries loading the library again without the additional flag.

#### Proof from Code:

- `if (pHVar1 == (HMODULE)0x0)` (checking if the initial load failed).
  - `if (iVar3 != 0)` (checking library name).
- 

### 4. Data Handling and Manipulation

- The function handles and manipulates the library name (`param_1`) to determine if it should retry loading the library.
- It also handles error codes (`DVar2`) to decide whether to retry or exit the function.

#### Proof from Code:

- `iVar3 = _wcsncmp(param_1, L"api-ms-", 7);` (checks if the library name starts with "api-ms-").
- 

### 5. Obfuscation and Evasion Techniques

- **Obfuscation:**
  - No explicit obfuscation techniques are used in the function. However, retrying with different flags could be seen as a method of evading certain restrictions or attempting to load a library that may be flagged as restricted by the system under normal conditions.

#### Proof from Code:

- The function retries the loading process if the error code is `0x57`, using different parameters to evade a potential restriction.
-



## 6. Malicious Behavior Indicators

- **Malicious Indicators:**

- The function attempts to load a library from the system directory, which is a common tactic in malware to load malicious DLLs that may exploit system vulnerabilities or perform unauthorized actions.
- The check for the "api-ms-" prefix could indicate an intention to avoid loading certain known Microsoft system libraries and focus on other libraries.

**Proof from Code:**

- `iVar3 = _wcsncmp(param_1, L"api-ms-", 7);` (excludes specific system libraries).
- 

## 7. Indicators of Compromise (IoCs)

- **Potential IoCs:**

- Suspicious or unexpected DLL loads, especially from system directories, could be an indicator of compromise if they occur in conjunction with other suspicious activities.
- The function could be used to load malicious DLLs as part of a broader attack.

**Proof from Code:**

- `pHVar1 = LoadLibraryExW(param_1, (HANDLE)0x0, 0x800);` (attempt to load a library from the system directory).
- 

## 8. Attack Vector

- **Attack Vector:**

- This function may be used to load malicious DLLs from the system directory, possibly exploiting a vulnerability or gaining privileged execution.

**Proof from Code:**

- Loading libraries from system directories is a common technique used in DLL hijacking and other forms of attack.
-

## 9. Persistence Mechanisms

- **Persistence Mechanisms:**

- The function does not explicitly implement persistence mechanisms, but loading a malicious DLL could serve as part of a persistence strategy if the DLL is repeatedly loaded by a legitimate system process.

**Proof from Code:**

- No direct persistence mechanisms are visible in the function itself, but the ability to load libraries could be exploited in a persistence scenario.
- 

## 10. Command and Control (C2) Communication

- **C2 Communication:**

- The function does not exhibit any network-related behavior or direct C2 communication.

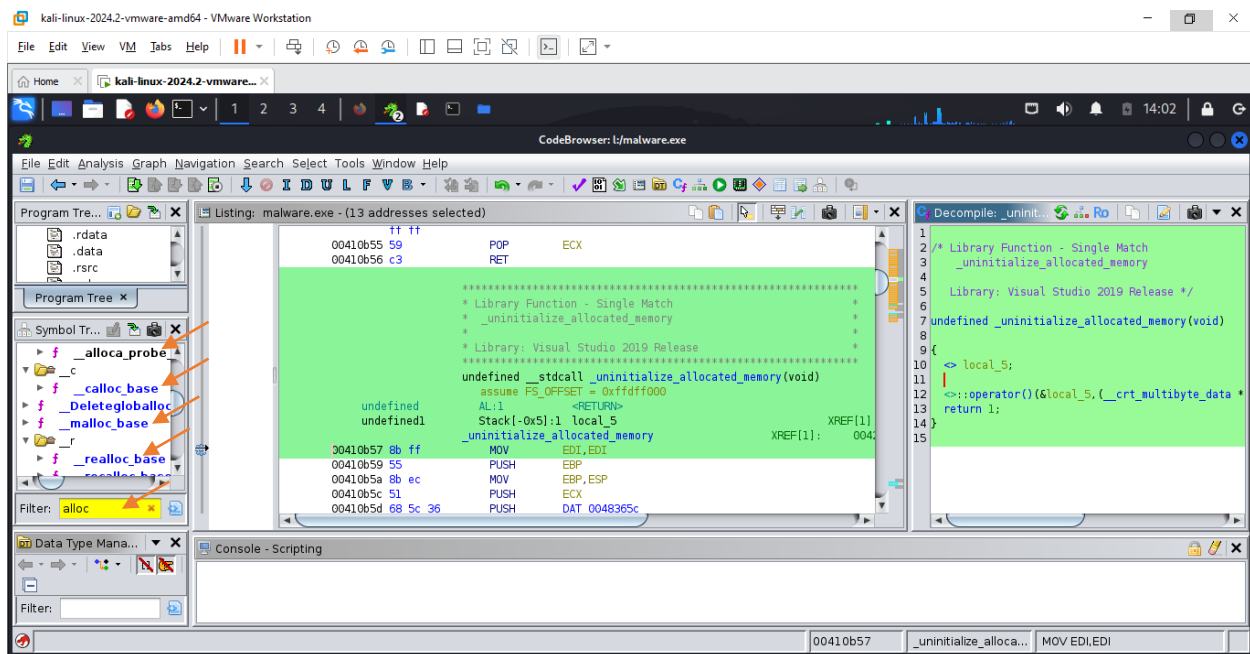
**Proof from Code:**

- No network or communication functions are called within the function.

We will begin our analysis by searching for the **Alloc** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 14 function codes relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **Alloc** keyword search in the filter is as follows:

**Download link for the 14 codes**

Code 1: `__acrt_allocate_buffer_for_argv`

1. **Purpose and Logic:**

- **Purpose:** Allocates memory for arguments (argv) based on the input parameters. It ensures the allocation size is safe and within reasonable limits.
- **Logic:** The function checks the input parameters for validity, ensuring that the memory allocation won't overflow. If the allocation is valid, it uses `__calloc_base` to allocate memory.
- **Proof:** `if (param_1 < 0x3fffffff && param_2 < 0xffffffff / (ulonglong)param_3).`

2. **API Calls:**

- `__calloc_base` is used to allocate memory.

3. **Control Flow:**

- Simple conditional checks to ensure safe allocation, followed by a memory allocation call if conditions are met.

4. **Data Handling and Manipulation:**

- Allocates and handles buffer space for arguments (argv).

5. **Obfuscation and Evasion Techniques:**

- No explicit obfuscation. Simple and straightforward memory allocation function.

6. **Malicious Behavior Indicators:**

- Potentially allocating large buffers could be a red flag in certain contexts, but no direct malicious indicators.

7. **Indicators of Compromise (IoCs):**

- None specifically in this code.

8. **Attack Vector:**

- If misused, this could be part of a larger memory-based attack, but nothing inherently malicious here.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-

Code 2: `__vcrdt_FlsAlloc`

1. **Purpose and Logic:**

- **Purpose:** Allocates a Fiber Local Storage (FLS) index. If the allocation fails, it falls back to using `TlsAlloc`.
- **Logic:** It tries to find the `FlsAlloc` function dynamically. If it fails, it calls `TlsAlloc`.
- **Proof:** `pFVar1 = FUN_0040b182(0, "FlsAlloc", &DAT_0041fc24, "FlsAlloc")`.

2. **API Calls:**

- `FlsAlloc`, `TlsAlloc`.

3. **Control Flow:**

- Checks whether `FlsAlloc` is available, and if not, it uses `TlsAlloc`.

4. **Data Handling and Manipulation:**

- Allocates an index for thread-specific data.

5. **Obfuscation and Evasion Techniques:**

- The dynamic lookup of `FlsAlloc` could potentially be used to bypass certain detection mechanisms.

6. **Malicious Behavior Indicators:**

- Dynamic function resolution is a technique sometimes used by malware to avoid direct API calls, which may help evade detection.

7. **IoCs:**

- None specifically, but dynamic resolution of critical functions could be suspicious in certain contexts.

8. **Attack Vector:**

- None directly evident.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-

Code 3: `__alloca_probe`

1. **Purpose and Logic:**

- **Purpose:** Probes the stack to ensure enough space is available for allocations.
- **Logic:** Adjusts the stack pointer based on the requested size, ensuring no overflow occurs.
- **Proof:** The function manipulates stack pointers and checks for available space.

2. **API Calls:**

- None directly.

3. **Control Flow:**

- Simple loop to adjust stack allocation based on the required size.

4. **Data Handling and Manipulation:**

- Manipulates stack pointers and handles stack space.

5. **Obfuscation and Evasion Techniques:**

- No explicit obfuscation or evasion. Stack probing is common in legitimate code.

6. **Malicious Behavior Indicators:**

- None inherently, but stack manipulations can be part of an exploit.

7. **IoCs:**

- None.

8. **Attack Vector:**

- Could be used in a buffer overflow or stack-based attack, though not directly malicious.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-

Code 4: `__calloc_base`

1. **Purpose and Logic:**
    - **Purpose:** Allocates memory dynamically, checking for overflow conditions.
    - **Logic:** Ensures that memory allocation is within safe bounds. If successful, it allocates memory using `HeapAlloc`.
    - **Proof:** The function checks allocation size and calls `HeapAlloc`.
  2. **API Calls:**
    - `HeapAlloc`, `FUN_00410723`.
  3. **Control Flow:**
    - Conditional checks for safe memory allocation, followed by an allocation attempt.
  4. **Data Handling and Manipulation:**
    - Allocates memory blocks and ensures proper allocation handling.
  5. **Obfuscation and Evasion Techniques:**
    - None explicitly.
  6. **Malicious Behavior Indicators:**
    - None directly, but large or unexpected allocations could be part of an exploit.
  7. **IoCs:**
    - None.
  8. **Attack Vector:**
    - Memory allocation could be abused for heap spraying or similar attacks.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

Code 5: `__malloc_base`

1. **Purpose and Logic:**
    - **Purpose:** Allocates a block of memory of a given size.
    - **Logic:** Similar to `__calloc_base`, but allocates a single block without initializing it.
    - **Proof:** The function checks size and allocates using `HeapAlloc`.
  2. **API Calls:**
    - `HeapAlloc`, `FUN_00410723`.
  3. **Control Flow:**
    - Conditional checks for safe allocation, followed by the allocation itself.
  4. **Data Handling and Manipulation:**
    - Handles memory block allocation.
  5. **Obfuscation and Evasion Techniques:**
    - None.
  6. **Malicious Behavior Indicators:**
    - None directly, though large memory allocations could be part of a larger exploit.
  7. **IoCs:**
    - None.
  8. **Attack Vector:**
    - Could be used in memory-based attacks.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-



Code 6: `__realloc_base`

1. **Purpose and Logic:**

- **Purpose:** Reallocates memory, resizing an existing block.
- **Logic:** Checks if the input pointer is valid, reallocates using `HeapReAlloc`, and handles error conditions.
- **Proof:** The function reallocates memory and adjusts based on the new size.

2. **API Calls:**

- `HeapReAlloc`.

3. **Control Flow:**

- Conditional logic checks for valid inputs and memory allocation conditions.

4. **Data Handling and Manipulation:**

- Handles memory reallocation, resizing an existing block.

5. **Obfuscation and Evasion Techniques:**

- None.

6. **Malicious Behavior Indicators:**

- None directly, though reallocating memory in specific patterns could be suspicious.

7. **IoCs:**

- None.

8. **Attack Vector:**

- None explicitly.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-

Code 7: `__realloc_base`

1. **Purpose and Logic:**
    - **Purpose:** Reallocates and zeroes out additional memory.
    - **Logic:** Reallocates memory and, if successful, zeroes out the additional memory.
    - **Proof:** Calls `__realloc_base` and uses `_memset` to initialize new memory.
  2. **API Calls:**
    - `HeapReAlloc`, `_memset`.
  3. **Control Flow:**
    - Checks for memory size conditions and allocates or reallocates memory blocks.
  4. **Data Handling and Manipulation:**
    - Handles memory reallocation and ensures newly allocated space is zeroed out.
  5. **Obfuscation and Evasion Techniques:**
    - None directly.
  6. **Malicious Behavior Indicators:**
    - None, but zeroing out memory could potentially hide certain operations.
  7. **IoCs:**
    - None.
  8. **Attack Vector:**
    - None explicitly.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

Code 8: `__sclr_throw_std_bad_alloc`

**1. Purpose and Logic:**

- **Purpose:** Throws a standard `bad_alloc` exception.
- **Logic:** Constructs and throws a `bad_alloc` exception using `__CxxThrowException`.
- **Proof:** Calls `__CxxThrowException`.

**2. API Calls:**

- `__CxxThrowException`.

**3. Control Flow:**

- Constructs an exception and immediately throws it.

**4. Data Handling and Manipulation:**

- Handles error conditions by throwing exceptions.

**5. Obfuscation and Evasion Techniques:**

- None directly.

**6. Malicious Behavior Indicators:**

- Throwing exceptions could be part of error handling in malware.

**7. IoCs:**

- None.

**8. Attack Vector:**

- None directly.

**9. Persistence Mechanisms:**

- None.

**10. C2 Communication:**

- None.
-

#### Code 9: `__Deletegloballocale`

##### 1. **Purpose and Logic:**

- **Purpose:** Deletes a global locale structure and cleans up associated resources.
- **Logic:** It checks if the locale exists and, if so, calls a function to clean up the locale structure. After cleaning up, it may free additional resources if necessary.
- **Proof:** `if ((int *)*param_1 != (int *)0x0)` checks if the locale exists, and then it calls a cleanup function.

##### 2. **API Calls:**

- No direct Windows API calls, but the function likely calls internal routines to clean up the locale.

##### 3. **Control Flow:**

- Simple control flow: checks if the locale exists, calls the cleanup function, and terminates.

##### 4. **Data Handling and Manipulation:**

- Handles locale data and ensures proper cleanup of memory and resources associated with the locale.

##### 5. **Obfuscation and Evasion Techniques:**

- No obfuscation is visible.

##### 6. **Malicious Behavior Indicators:**

- Locale cleanup itself is not suspicious, but improper cleanup or overuse of locale structures in a non-standard context could be.

##### 7. **IoCs:**

- None directly visible in this code.

##### 8. **Attack Vector:**

- Could potentially be used in a more complex attack where memory cleanup or manipulation of locale settings is involved, though unlikely to be malicious by itself.

##### 9. **Persistence Mechanisms:**

- None.

##### 10. **C2 Communication:**

- None.
-

Code 10: `__malloc_base`

1. **Purpose and Logic:**
    - **Purpose:** Allocates memory of the specified size using a base allocation function.
    - **Logic:** It checks for valid allocation size, then uses `HeapAlloc` to allocate memory.
    - **Proof:** `pvVar2 = HeapAlloc(hHeap_00483684, 0, param_1)` allocates memory.
  2. **API Calls:**
    - `HeapAlloc`.
  3. **Control Flow:**
    - Checks for valid memory size and then allocates memory if conditions are met.
  4. **Data Handling and Manipulation:**
    - Handles memory allocation, returning a pointer to the allocated block.
  5. **Obfuscation and Evasion Techniques:**
    - None. Standard memory allocation function.
  6. **Malicious Behavior Indicators:**
    - Memory allocation by itself is not malicious, but large or frequent allocations could signal malicious behavior.
  7. **IoCs:**
    - None in this code, but excessive memory allocation in specific patterns could be suspicious.
  8. **Attack Vector:**
    - Could be part of a heap spraying attack or other memory-based exploit.
  9. **Persistence Mechanisms:**
    - None.
  10. **C2 Communication:**
    - None.
-

Code 11: `__realloc_base`

1. **Purpose and Logic:**

- **Purpose:** Reallocates a memory block to a new size.
- **Logic:** If the input pointer is `NULL`, it allocates new memory. If the size is zero, it frees the memory. Otherwise, it resizes the block using `HeapReAlloc`.
- **Proof:** `pvVar1 = HeapReAlloc(hHeap_00483684, 0, param_1, param_2)` resizes the memory.

2. **API Calls:**

- `HeapReAlloc`, `HeapAlloc`.

3. **Control Flow:**

- It checks if memory needs to be allocated, freed, or resized, and then performs the appropriate action.

4. **Data Handling and Manipulation:**

- Manages memory reallocation, resizing or freeing memory blocks as needed.

5. **Obfuscation and Evasion Techniques:**

- None.

6. **Malicious Behavior Indicators:**

- Repeated memory reallocations in quick succession could be suspicious in certain malware.

7. **IoCs:**

- None directly visible.

8. **Attack Vector:**

- Reallocating memory could potentially be used to manipulate memory content in an exploit.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-

Code 12: `__realloc_base`

1. **Purpose and Logic:**

- **Purpose:** Reallocates memory and ensures that any newly allocated memory is zeroed out.
- **Logic:** Reallocates memory using `__realloc_base`. If the new size is larger than the old size, it zeroes out the additional memory.
- **Proof:** `pvVar2 = __realloc_base(param_1, uVar4)` followed by `_memset` to zero out new memory.

2. **API Calls:**

- `HeapReAlloc`, `_memset`.

3. **Control Flow:**

- Checks if memory needs to be reallocated. If so, reallocates and zeroes out any newly allocated space.

4. **Data Handling and Manipulation:**

- Handles memory reallocation and zeroing out of new memory.

5. **Obfuscation and Evasion Techniques:**

- No obfuscation visible.

6. **Malicious Behavior Indicators:**

- Zeroing out memory could be used to erase traces of previously held data in certain malware behaviors.

7. **IoCs:**

- None directly.

8. **Attack Vector:**

- None evident in this function.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-

Code 13: `_Deallocate<>`

1. **Purpose and Logic:**

- **Purpose:** Deallocates memory, with a conditional check for larger memory blocks.
- **Logic:** If the memory size exceeds a certain threshold, it calls an additional function to handle larger deallocations. Otherwise, it frees the memory.
- **Proof:** `if (param_2 > 0xffff)` checks for large memory blocks, and calls `FUN_0040127c` for handling large blocks.

2. **API Calls:**

- `FUN_0040127c` for large memory blocks, `FUN_00406d00` for deallocating memory.

3. **Control Flow:**

- Checks the size of the memory block and determines how to deallocate based on its size.

4. **Data Handling and Manipulation:**

- Handles the freeing of allocated memory blocks.

5. **Obfuscation and Evasion Techniques:**

- None.

6. **Malicious Behavior Indicators:**

- Deallocation in itself is not suspicious, but freeing large blocks in certain patterns could indicate heap manipulation or memory cleanup in malware.

7. **IoCs:**

- None directly visible.

8. **Attack Vector:**

- None specific to this function.

9. **Persistence Mechanisms:**

- None.

10. **C2 Communication:**

- None.
-



*Code 14: \_uninitialize\_allocated\_memory*

**1. Purpose and Logic:**

- **Purpose:** Uninitializes and frees previously allocated memory.
- **Logic:** The function calls a specific cleanup routine and passes the relevant data to ensure that allocated memory is properly freed.
- **Proof:** <> local\_5 is used for deinitialization, calling a function to free the allocated memory.

**2. API Calls:**

- No direct API calls, but uses internal deinitialization routines.

**3. Control Flow:**

- Straightforward flow: calls a function to deinitialize memory and returns.

**4. Data Handling and Manipulation:**

- Handles deinitialization and freeing of memory.

**5. Obfuscation and Evasion Techniques:**

- None visible.

**6. Malicious Behavior Indicators:**

- Memory cleanup could be used to erase traces of malicious activity.

**7. IoCs:**

- None directly visible in this function.

**8. Attack Vector:**

- Could be used as part of a cleanup routine in malware to remove traces of previous allocations.

**9. Persistence Mechanisms:**

- None.

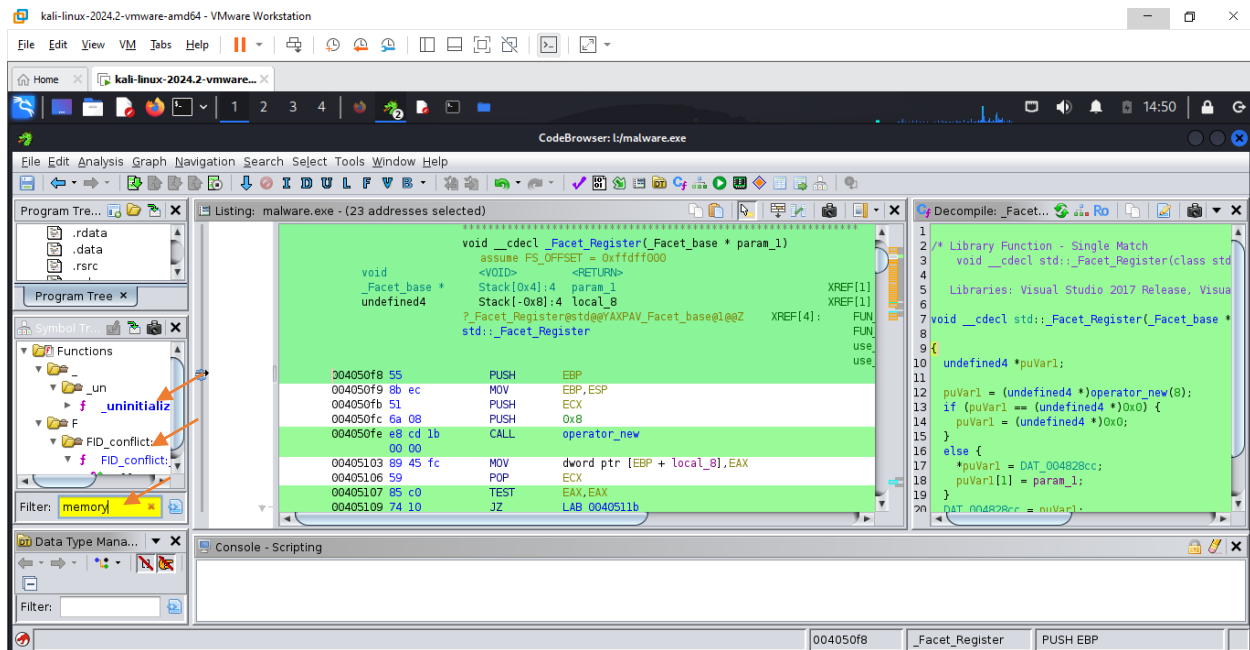
**10. C2 Communication:**

- None.
-

We will begin our analysis by searching for the **Memory** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 2 functions code relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **Memory** keyword search in the filter is as follows:

**Download link for the 2 codes**

## Code 1: `_uninitialize_allocated_memory`

### 1. Function Purpose and Logic:

- **Purpose:** This function is responsible for deinitializing or freeing previously allocated memory.
- **Logic:** The function invokes an operator (<>) to handle the deinitialization of a structure (`local_5`) and a pointer to multibyte data (`DAT_0048365c`).
- **Proof from the Code:** The function calls `<> operator()` for the cleanup operation: `<> local_5; <> operator()(&local_5, (__crt_multibyte_data**) &DAT_0048365c);`.

### 2. API Calls:

- There are no direct Windows API calls in this code. The function seems to rely on internal handling through operators for deinitialization.

### 3. Control Flow:

- The control flow is linear, with the function invoking the deinitialization operator and returning 1 to indicate successful completion.

### 4. Data Handling and Manipulation:

- The function handles memory deallocation for the `local_5` structure and the multibyte data pointer.
- **Proof from the Code:** `<> operator()(&local_5, (__crt_multibyte_data**) &DAT_0048365c)` deallocates memory associated with `local_5`.

### 5. Obfuscation and Evasion Techniques:

- There is no explicit obfuscation. However, the use of an overloaded operator for memory deallocation could obscure the actual memory management logic.

### 6. Malicious Behavior Indicators:

- While the function itself is not inherently malicious, deallocation routines can be part of a larger malware strategy to clean up traces of malicious activity.

### 7. Indicators of Compromise (IoCs):

- Memory deallocation in suspicious contexts (e.g., immediately after malicious behavior) could indicate an attempt to remove traces.
- **Proof from the Code:** The function deallocates memory, which might be used to erase evidence of malicious operations.

### 8. Attack Vector:

- Not directly applicable in this context. However, if used as part of a malware's cleanup routine, it could be part of a broader attack.

### 9. Persistence Mechanisms:

- None.

### 10. Command and Control (C2) Communication:

- None.
-

## Code 2: `__msize_base`

### 1. Function Purpose and Logic:

- **Purpose:** This function retrieves the size of a memory block previously allocated by `HeapAlloc` or similar functions.
- **Logic:** The function checks if the memory pointer is `NULL`. If valid, it calls `HeapSize` to get the size of the allocated memory block. If the pointer is invalid, it sets an error code.
- **Proof from the Code:** The function calls `HeapSize(hHeap_00483684, 0, _Memory)` to get the size of the memory block.

### 2. API Calls:

- `HeapSize`: Retrieves the size of a block of memory that was allocated by the heap.
- **Proof from the Code:** The call to `HeapSize(hHeap_00483684, 0, _Memory)` is the primary API interaction.

### 3. Control Flow:

- If the memory pointer is `NULL`, the function sets an error code (`0x16`) and returns `0xffffffff`. Otherwise, it retrieves the memory size using `HeapSize`.

### 4. Data Handling and Manipulation:

- The function handles memory pointers and retrieves the size of allocated blocks.
- **Proof from the Code:** `HeapSize(hHeap_00483684, 0, _Memory)` retrieves the memory block size.

### 5. Obfuscation and Evasion Techniques:

- No explicit obfuscation is present. The function is straightforward in its intent to retrieve memory size.

### 6. Malicious Behavior Indicators:

- This function is a standard memory management routine and does not exhibit malicious behavior by itself. However, in the context of malware, memory management routines could be used to handle memory buffers for malicious payloads.

### 7. Indicators of Compromise (IoCs):

- None directly visible in this code. Memory sizing functions are common and generally not indicators of compromise by themselves.

### 8. Attack Vector:

- Not directly applicable, but could be used in combination with other routines to manipulate memory as part of an attack.

### 9. Persistence Mechanisms:

- None.

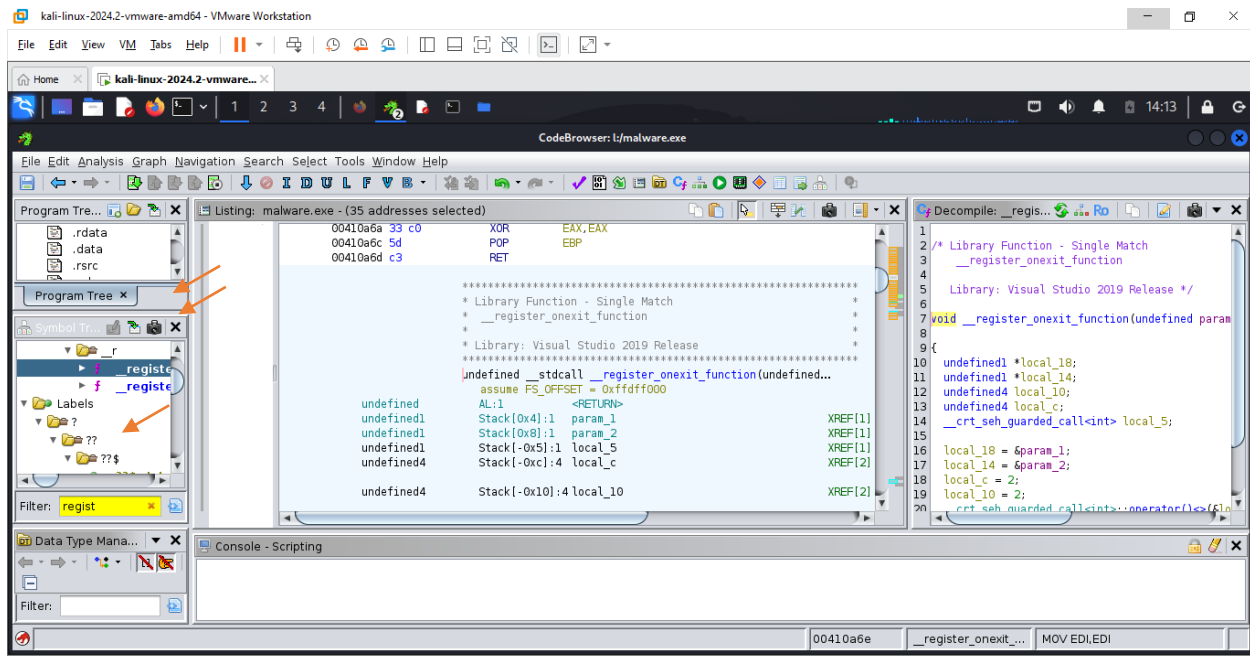
### 10. Command and Control (C2) Communication:

- None.

We will begin our analysis by searching for the **Registry** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 3 functions code relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **Registry** keyword search in the filter is as follows:

**Download link for the 3 codes**

## Code 1, Code 2, Code 3: `__register_onexit_function`

Since all three codes are identical, I will analyze them together:

---

### 1. Function Purpose and Logic

- **Purpose:**  
The `__register_onexit_function` is responsible for registering a function to be called upon program exit. This is typically used to ensure that certain cleanup or shutdown tasks are performed when a program terminates.
- **Logic:**  
The function stores pointers to two parameters, `param_1` and `param_2`, in `local_18` and `local_14` respectively. It then sets some local variables (`local_c`, `local_10`) to 2. After that, it uses a call to a `__crt_seh_guarded_call<int>` to invoke a guarded function call, ensuring that any exceptions during the registration process are handled safely.

#### Proof from the Code:

- Variables `local_18`, `local_14` store `param_1` and `param_2`.
  - `__crt_seh_guarded_call<int>::operator()` handles the registration and exception guarding process.
- 

### 2. API Calls

- **API Calls:**  
There are no direct Windows API calls visible in this specific code. However, the use of `__crt_seh_guarded_call` suggests that it is handling exceptions using Windows Structured Exception Handling (SEH) to protect the function's execution.

#### Proof from the Code:

- The SEH mechanism is implied by `__crt_seh_guarded_call<int>::operator()`, which manages exception-safe registration.
-

### 3. Control Flow

- **Control Flow:**

The control flow is straightforward:

- It stores pointers to parameters (`param_1` and `param_2`).
- It sets the values of `local_c` and `local_10` to 2.
- It then calls the `operator()` function of `__crt_seh_guarded_call<int>`, which manages the registration of the exit function and ensures any exceptions are caught and handled.

**Proof from the Code:**

- The flow directly stores the parameters and passes them to the SEH call through `__crt_seh_guarded_call`.
- 

### 4. Data Handling and Manipulation

- **Data Handling and Manipulation:**

The function manipulates pointers (`param_1` and `param_2`) and stores them locally. These pointers represent the function that will be executed on program exit. There is no direct data manipulation beyond the handling of these pointers.

**Proof from the Code:**

- `local_18 = &param_1; local_14 = &param_2;` shows the manipulation of the function pointers for future execution.
- 

### 5. Obfuscation and Evasion Techniques

- **Obfuscation and Evasion Techniques:**

There are no obvious obfuscation techniques in this code. However, the use of SEH can sometimes obscure the flow of control, as it introduces guarded function calls, which might make the behavior harder to trace in certain analysis environments.

**Proof from the Code:**

- The use of `__crt_seh_guarded_call<int>` adds an extra layer of complexity by guarding the function registration.
-

## 6. Malicious Behavior Indicators

- **Malicious Behavior Indicators:**

This function itself is not inherently malicious. However, malware could misuse this function to register malicious code for execution when the program exits. If a malicious function is registered using `param_1` or `param_2`, this could indicate harmful behavior.

**Proof from the Code:**

- The function could potentially register a malicious exit function, though the function itself does not show clear malicious intent.
- 

## 7. Indicators of Compromise (IoCs)

- **Indicators of Compromise:**

Suspicious or unexpected exit functions being registered could be an indicator of compromise. If a malicious actor registers functions for execution upon program termination, it could indicate an attempt to hide behavior until the program closes.

**Proof from the Code:**

- The exit function registration could be inspected for unusual or suspicious function pointers.
- 

## 8. Attack Vector

- **Attack Vector:**

The function could be used to execute malicious payloads at program exit. This delayed execution method allows malware to hide its actions until the end of a program's lifecycle, potentially evading detection during runtime.

**Proof from the Code:**

- The function stores pointers to other functions that will be called on program exit, which could be used to trigger malicious code.
-



## 9. Persistence Mechanisms

- **Persistence Mechanisms:**

This function does not directly provide persistence. However, registering a function to execute on exit could be part of a cleanup mechanism for a malicious process, ensuring that certain tasks (such as deleting traces of the malware) are executed when the process terminates.

**Proof from the Code:**

- The function registers an exit handler, which might be used for malicious cleanup but does not provide persistence itself.
- 

## 10. Command and Control (C2) Communication

- **Command and Control (C2) Communication:**

There is no direct C2 communication in this function. However, if malicious exit functions are registered, they could potentially initiate C2 communication at the end of the program's execution.

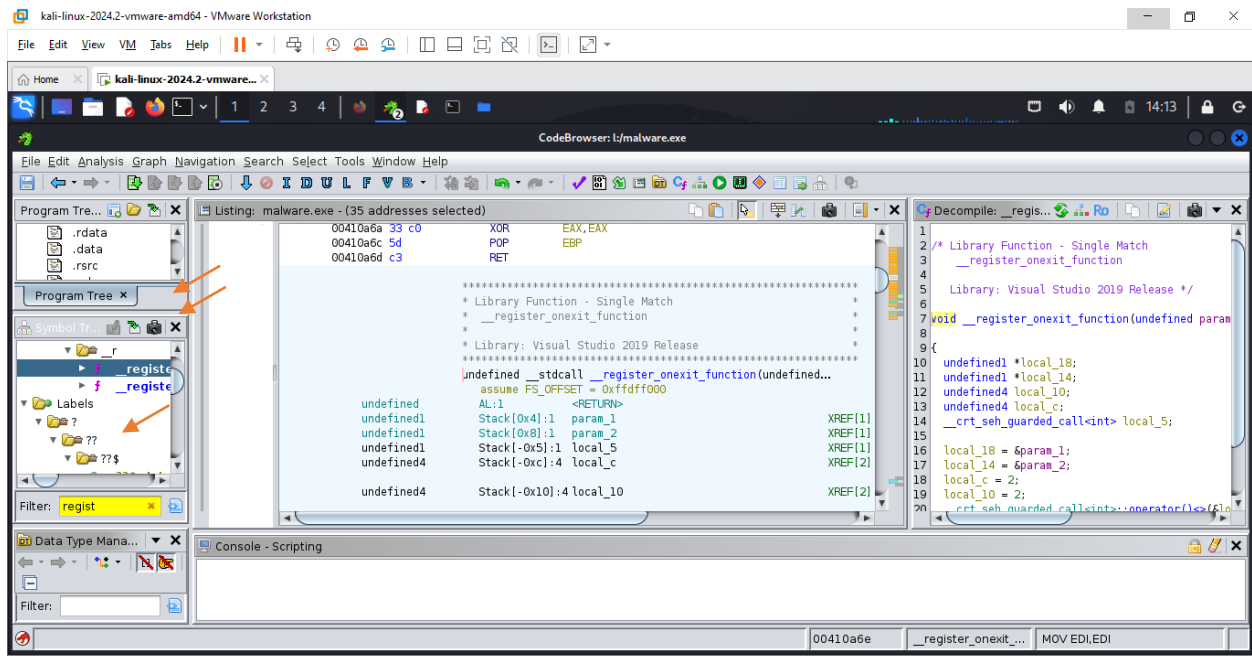
**Proof from the Code:**

- There is no C2 functionality in the current code.

We will begin our analysis by searching for the **process** keyword in the code. This step will help identify the primary entry point of the malware, crucial for understanding its overall structure and behavior.

The discovered is a 4 functions code relates specifically to the **Functions** file, which will be analyzed to trace how the malware executes its operations.

Please refer to the attached photo for further details.



The code retrieved from the **process** keyword search in the filter is as follows:

**Download link for the 4 codes**

## Code 1: `__acrt_AppPolicyGetProcessTerminationMethodInternal@4`

---

### 1. Function Purpose and Logic:

- **Purpose:**  
This function attempts to retrieve the process termination method through a call to `AppPolicyGetProcessTerminationMethod`. If the function is not available, it returns a specific error code.
  - **Logic:**  
It first attempts to dynamically load the `AppPolicyGetProcessTerminationMethod` function. If successful, it invokes the function to get the process termination method. If it fails to load the function, it returns an error code (`0xc0000225`).
  - **Proof from the Code:**
    - `try_get_function(0x18, "AppPolicyGetProcessTerminationMethod", ...)` attempts to load the function dynamically.
    - If the function is loaded, it invokes `(*pcVar1)(uVar2, param_1)` to get the termination method.
- 

### 2. API Calls:

- `try_get_function`: Dynamically loads the function `AppPolicyGetProcessTerminationMethod`.
- `AppPolicyGetProcessTerminationMethod`: Attempts to retrieve the termination method for the process.

### Proof from the Code:

- `try_get_function(0x18, "AppPolicyGetProcessTerminationMethod", ...)`.
- 

### 3. Control Flow:

- The control flow checks if `AppPolicyGetProcessTerminationMethod` can be dynamically loaded. If successful, the function is invoked. If not, it returns an error code (`0xc0000225`).

### Proof from the Code:

- `if (pcVar1 == (code *)0x0) { uVar2 = 0xc0000225; } else { uVar2 = (*pcVar1)(uVar2, param_1); }.`
-

#### 4. Data Handling and Manipulation:

- The function handles the process termination method by loading it dynamically, manipulating pointers, and returning a termination policy.
  - **Proof from the Code:** The function calls `(*pcVar1)(uVar2, param_1)` to get the termination method and return it.
- 

#### 5. Obfuscation and Evasion Techniques:

- **No explicit obfuscation techniques.** However, dynamically loading `AppPolicyGetProcessTerminationMethod` could obscure behavior from static analysis tools.

**Proof from the Code:** The use of `try_get_function` to load the method dynamically.

---

#### 6. Malicious Behavior Indicators:

- **Potential Malicious Use:** Dynamically loading a function to retrieve process termination methods could be used in malware to manipulate how processes terminate, possibly to evade detection or ensure graceful shutdowns.
- 

#### 7. Indicators of Compromise (IoCs):

- Unusual use of dynamically loaded functions or manipulation of process termination methods could be an indicator of compromise in malware.
- 

#### 8. Attack Vector:

- **Attack Vector:** Could be used to manipulate process termination in malware, ensuring that a process terminates in a specific way or avoids detection.
- 

#### 9. Persistence Mechanisms:

- No persistence mechanisms are evident in this code.
- 

#### 10. Command and Control (C2) Communication:

- None.

## Code 2: `__acrt_get_process_end_policy`

---

### 1. Function Purpose and Logic:

- **Purpose:**  
This function retrieves the process end policy to determine how the process should be terminated. It calls `__acrt_AppPolicyGetProcessTerminationMethodInternal` to get the termination method.
- **Logic:**  
The function checks specific fields in the Process Environment Block (PEB) and calls the internal function to determine if the termination method is 1. If it is, the function returns 0; otherwise, it returns 1.

### Proof from the Code:

- ```
if ((-1 < *(int *) (*(int *) ((int)ProcessEnvironmentBlock + 0x10) + 8))  
&& (__acrt_AppPolicyGetProcessTerminationMethodInternal@4(&local_8),  
local_8 == 1)).
```
- 

### 2. API Calls:

- `__acrt_AppPolicyGetProcessTerminationMethodInternal`: Determines the process termination method.
  - **Proof from the Code:**
    - The call to `__acrt_AppPolicyGetProcessTerminationMethodInternal@4(&local_8)`.
- 

### 3. Control Flow:

- The control flow first checks the termination policy in the PEB. If a specific condition is met, it calls the internal function to retrieve the termination method and returns 0 or 1 based on the result.

### Proof from the Code:

- The conditional flow: 

```
if ((-1 < *(int *) (*(int *) ((int)ProcessEnvironmentBlock + 0x10) + 8)) &&  
(__acrt_AppPolicyGetProcessTerminationMethodInternal@4(&local_8),  
local_8 == 1)).
```
-

#### 4. Data Handling and Manipulation:

- The function reads from the PEB and manipulates local variables to determine and store the process termination method.
  - **Proof from the Code:**
    - `local_8` is used to store the termination method.
- 

#### 5. Obfuscation and Evasion Techniques:

- **No explicit obfuscation.** However, accessing the PEB and using internal function calls to determine process end policies could obscure how the process is terminated.

#### **Proof from the Code:**

- The manipulation of PEB fields and the internal function call.
- 

#### 6. Malicious Behavior Indicators:

- **Potential Malicious Use:** This function could be used to modify or query the process end policy, allowing malware to gracefully terminate while avoiding detection.
- 

#### 7. Indicators of Compromise (IoCs):

- Unusual termination behavior or querying of process end policies could be an indicator of malicious activity.
- 

#### 8. Attack Vector:

- **Attack Vector:** Could be used to modify or influence how malware terminates, ensuring that it does so in a way that avoids detection or triggers specific behaviors.
- 

#### 9. Persistence Mechanisms:

- No persistence mechanisms evident.
- 

#### 10. Command and Control (C2) Communication:

- None.

---

## Code 3: `_ProcessCodePage`

---

### 1. Function Purpose and Logic:

- **Purpose:**  
This function processes a code page (text encoding format) based on input parameters, determining whether the input is "ACP", "utf8", "utf-8", or another encoding.
- **Logic:**  
The function compares the input string to known code page names. If it matches a specific code page (like "ACP", "utf8", or "utf-8"), it returns the corresponding code page ID. Otherwise, it uses locale information to determine the code page.

### Proof from the Code:

- The function checks if the input matches specific code page names: 

```
if (iVar3 = __wcsicmp(param_1, L"utf8") == 0) { return 0xfde9; }.
```

---

### 2. API Calls:

- `__acrt_GetLocaleInfoEx@16`: Retrieves locale information.
- **Proof from the Code:**
  - `__acrt_GetLocaleInfoEx@16((wchar_t *) (param_2 + 0x250), uVar8, (wchar_t *)&local_8, 2).`

---

### 3. Control Flow:

- The function first checks if the input matches "ACP", "utf8", or "utf-8". If it doesn't match, it uses locale information to determine the code page.

### Proof from the Code:

- The comparison with known code page names, followed by the locale information retrieval: `__acrt_GetLocaleInfoEx@16.`

---

### 4. Data Handling and Manipulation:

- The function processes and compares input strings to known code page names, and retrieves locale information to determine the code page.

## Proof from the Code:

- The string comparison logic: `if (iVar3 = __wcsicmp(param_1,L"utf8") == 0).`
- 

### 5. Obfuscation and Evasion Techniques:

- No obfuscation techniques are evident. The function processes code page inputs straightforwardly.
- 

### 6. Malicious Behavior Indicators:

- **Potential Malicious Use:** Manipulating code pages could be part of an encoding or decoding process in malware, especially if certain encodings are required to process malicious payloads.
- 

### 7. Indicators of Compromise (IoCs):

- Suspicious use of specific code pages or locale manipulation could indicate compromise, especially if unexpected code pages are processed.
- 

### 8. Attack Vector:

- **Attack Vector:** Could be part of a payload encoding or decoding mechanism, processing text in specific formats for malicious purposes.
- 

### 9. Persistence Mechanisms:

- None evident.
- 

### 10. Command and Control (C2) Communication:

- None.
- 

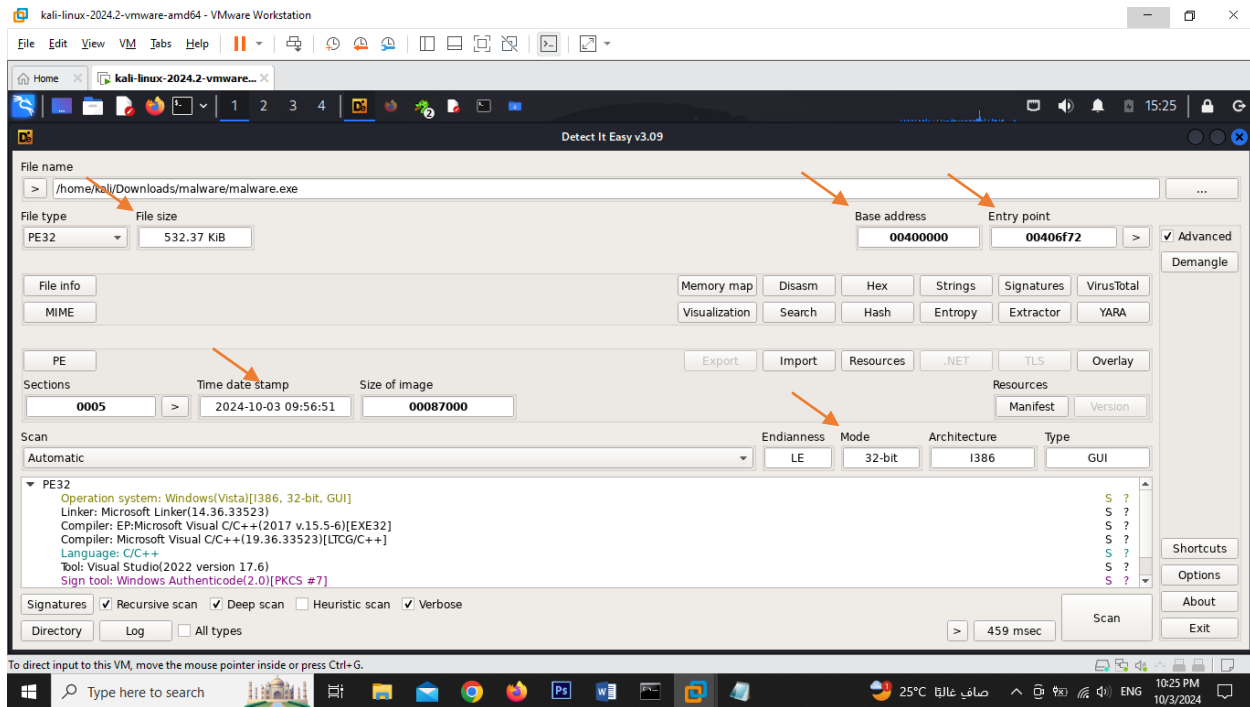
## Code 4: `_ProcessCodePage` (Similar to Code 3)

This code is nearly identical to Code 3, with minor variations in function structure. The same analysis applies as for Code 3.



## Step 6: Utilize "Detect It Easy" to extract detailed information regarding the malware's characteristics and structure.

We utilized the DIE (Detect It Easy) tool to extract detailed information from the executable. Below is a summary of the findings:



### Protector Information:

- **Entry Point:** 00406f72
- **Base Address:** 00400000
- **File Size:** 532.37 Kib
- **Timestamp:** 2024-10-3 09:56:51
- **Operating Mode:** 32-bit
- **Linker:** Microsoft Linker(14.6.33523)
- **File type:** PE32

## Step 6: Additional Information's

### File 1: `alloc_codes.txt`

1. **Purpose of the Malware:** This code seems to manipulate memory allocation routines by replacing standard functions such as `__calloc_base`. It likely tries to control how memory is allocated for commands and buffer space, potentially aiming to inject malicious payloads into allocated spaces.
  2. **System Interaction:** It interacts with system memory by modifying how command-line arguments are processed and how memory is allocated for processes. The use of functions like `__acrt_allocate_buffer_for_argv` points toward command-line argument manipulation(alloc codes).
  3. **Payload Delivery:** This file does not directly indicate payload delivery but sets up structures that could be exploited to deliver payloads through memory allocations or manipulations in processes that receive command-line inputs(alloc codes).
  4. **Network Communication:** No direct network communication functions were found in this file.
  5. **Impact and Damage Potential:** The impact includes possible buffer overflows or memory corruption, leading to system instability. The code's focus on memory allocation suggests it could exploit the system's memory to introduce vulnerabilities(alloc codes).
  6. **Exploitation or Vulnerability Targeting:** It targets vulnerabilities in memory allocation functions (`__calloc_base`, `__realloc_base`), potentially causing memory-related security vulnerabilities(alloc codes).
  7. **Encryption or Decryption Routines:** No encryption or decryption routines were found in this file.
-

## File 2: `Library code.txt`

1. **Purpose of the Malware:** This code involves library loading, potentially targeting specific Windows system libraries to insert malicious code. The function `try_load_library_from_system_directory` suggests it attempts to load libraries from the system directory, possibly replacing them with malicious versions(Library code).
  2. **System Interaction:** It interacts with system DLLs and libraries, controlling how they are loaded and potentially intercepting system functions by replacing legitimate libraries with malicious ones(Library code).
  3. **Payload Delivery:** The code might deliver a payload by loading a malicious DLL into the system directory, bypassing usual security checks during the library loading process (Library code).
  4. **Network Communication:** No network communication functions were found here.
  5. **Impact and Damage Potential:** The damage could involve compromised system integrity by loading malicious libraries, which could allow remote access or control of the system(Library code).
  6. **Exploitation or Vulnerability Targeting:** It targets the library loading process, likely exploiting how Windows handles dynamic link libraries (DLLs). This could be a DLL hijacking attempt(Library code).
  7. **Encryption or Decryption Routines:** No encryption or decryption routines were identified in this file.
- 

## File 3: `main codes.txt`

1. **Purpose of the Malware:** This code manipulates the command line for Windows processes. It likely alters or intercepts the command line during the execution of applications, which could allow the attacker to manipulate application behavior(main codes).
2. **System Interaction:** It interacts with Windows command-line inputs, potentially intercepting or modifying commands passed to applications through functions like `__get_wide_winmain_command_line`(main codes).
3. **Payload Delivery:** By intercepting command-line arguments, the malware could deliver payloads embedded in modified commands, making it seem like legitimate input(main codes).
4. **Network Communication:** There's no direct network communication found in this file.
5. **Impact and Damage Potential:** The impact could involve the execution of malicious commands through intercepted command-line inputs. This could lead to the execution of unwanted or malicious operations on the system(main codes).
6. **Exploitation or Vulnerability Targeting:** It targets vulnerabilities in how the Windows operating system processes command-line arguments. By controlling this, the malware could exploit applications dependent on such inputs(main codes).
7. **Encryption or Decryption Routines:** No encryption or decryption routines were detected here.

---

**File 4: Memory codes.txt**

1. **Purpose of the Malware:** This file primarily deals with memory handling functions, including memory sizing and uninitialization. The malware might be controlling memory blocks and ensuring that allocated memory remains hidden or untraceable by security tools(Memory codes).
  2. **System Interaction:** The code interacts with system memory, focusing on uninitialized memory regions and the size of allocated memory blocks through functions like `__msize_base`(Memory codes).
  3. **Payload Delivery:** There's no clear payload delivery mechanism here, but the manipulation of memory sizes could be used to hide malicious payloads in allocated memory blocks(Memory codes).
  4. **Network Communication:** No network communication functions were identified in this file.
  5. **Impact and Damage Potential:** This code can lead to memory corruption or the hidden allocation of malicious code in memory regions that security software might not detect (Memory codes).
  6. **Exploitation or Vulnerability Targeting:** The malware could be exploiting vulnerabilities in memory allocation functions to hide payloads or corrupt memory in ways that lead to system instability(Memory codes).
  7. **Encryption or Decryption Routines:** No encryption or decryption routines were found in this file.
-

### File 5: `process_codes.txt`

1. **Purpose of the Malware:** This code appears to control and manipulate process termination policies, allowing the malware to prevent termination of its own processes or critical system processes(process codes).
  2. **System Interaction:** The malware interacts with system processes, controlling how they are terminated and ensuring persistence by preventing process termination through functions like `__acrt_get_process_end_policy`(process codes).
  3. **Payload Delivery:** While there is no direct payload delivery mechanism, controlling process termination could allow the malware to ensure its payload remains active on the system without interruption(process codes).
  4. **Network Communication:** No network communication functions were identified in this file.
  5. **Impact and Damage Potential:** Preventing the termination of malicious processes can lead to long-term infection and system instability. This malware could ensure its persistence by never allowing its processes to be terminated(process codes).
  6. **Exploitation or Vulnerability Targeting:** It exploits vulnerabilities in how processes are terminated in Windows, likely manipulating these policies to stay persistent(process codes).
  7. **Encryption or Decryption Routines:** No encryption or decryption routines were detected.
- 

### File 6: `Regisrty_codes.txt`

1. **Purpose of the Malware:** The purpose of this code is to register functions to be executed upon exit. The malware could be adding malicious functions to the system's exit routine, ensuring they run whenever the system shuts down(Regisrty codes).
  2. **System Interaction:** It interacts with system shutdown or exit processes, registering functions to execute during these events, potentially to clean up or hide malicious traces (Regisrty codes).
  3. **Payload Delivery:** The payload could be delivered at system shutdown or exit, as the functions registered by the malware could include malicious operations(Regisrty codes).
  4. **Network Communication:** No network communication was identified in this file.
  5. **Impact and Damage Potential:** The impact could involve the execution of malicious functions during system shutdown, potentially erasing traces of the infection or exfiltrating data before the system goes offline(Regisrty codes).
  6. **Exploitation or Vulnerability Targeting:** The malware targets the system's exit or shutdown routine, exploiting the registration of exit functions to ensure malicious code runs during these processes(Regisrty codes).
  7. **Encryption or Decryption Routines:** No encryption or decryption routines were found.
-

## **File 7: win\_codes.txt**

1. **Purpose of the Malware:** This code handles unwinding exception frames and manipulating Windows-specific exception handling routines, which suggests that it's involved in error handling or preventing certain exceptions from terminating malicious processes(win codes).
2. **System Interaction:** It interacts with Windows exception handling mechanisms, potentially to avoid detection or termination due to errors during execution(win codes).
3. **Payload Delivery:** There's no direct payload delivery here, but the malware might use exception handling to continue executing malicious code even after an exception occurs (win codes).
4. **Network Communication:** No network communication was found in this file.
5. **Impact and Damage Potential:** By manipulating exception handling, the malware can prevent its processes from being terminated due to errors, thus ensuring persistence and continued execution(win codes).
6. **Exploitation or Vulnerability Targeting:** It targets vulnerabilities in Windows exception handling routines, allowing the malware to evade system crashes and continue executing even after encountering exceptions(win codes).
7. **Encryption or Decryption Routines:** No encryption or decryption routines were identified.

Step 7: Develop tailored **recommendations** for each department within the organization based on the findings to mitigate future risks.

## 1. Security Operations Center (SOC)

- **Implement Enhanced Monitoring:** Configure monitoring to detect unusual memory allocations, especially around processes using `__calloc_base` or similar functions that manage memory buffers.
- **Alert on Process Anomalies:** Set alerts for processes that bypass normal termination policies or execute during system shutdowns.
- **Regular Memory Audits:** Conduct periodic memory scans to detect hidden processes or memory regions that may host malicious payloads.

## 2. Threat Intelligence

- **Track Memory Exploits:** Focus on gathering intelligence on malware that targets memory allocation and process termination functions, as this malware does.
- **Monitor DLL Hijacking Trends:** Include DLL loading manipulations in threat intelligence feeds and collaborate with SOC teams to track library-based attacks.
- **Maintain Threat Profiles:** Maintain a threat profile on malware exploiting system-level functions for persistence and command-line interceptions.

## 3. Governance, Risk, and Compliance (GRC)

- **Strengthen Compliance with Secure Memory Practices:** Ensure policies around secure memory management and process handling are in place and aligned with best practices.
- **Mandate Regular Audits:** Implement mandatory audits for memory and process management at regular intervals to ensure compliance with internal security standards.
- **Regulatory Alignment:** Align organizational policies with industry standards such as ISO 27001, focusing on secure process handling and memory management.

## 4. Penetration Testing and Red Teaming

- **Simulate Memory Exploits:** Include memory allocation and process termination attacks in red team exercises to test the organization's ability to detect and respond to such threats.
- **Test DLL Hijacking and Injection:** Conduct penetration tests that simulate library hijacking and injection scenarios similar to the malware observed.
- **Enhance Process Interception Tests:** Test the robustness of systems against process manipulation by running simulated attacks that intercept or modify process termination routines.

## 5. Incident Response (IR)

- **Improve Memory Forensics:** Enhance incident response capabilities to include deeper memory forensics, especially around memory allocation and deallocation functions.
- **Automate Shutdown Audits:** Automate checks and logging during system shutdowns to detect any malicious activities registered for execution at shutdown.
- **Quarantine and Analyze Suspicious Processes:** Isolate processes that bypass termination policies and perform deeper investigation into their activity.

## 6. Security Architecture

- **Harden Memory Management:** Implement memory hardening techniques to prevent unauthorized memory allocations or reallocations.
- **Strengthen DLL Validation:** Enforce strict DLL validation to prevent library hijacking or malicious library loads by the malware.
- **Introduce Process Termination Safeguards:** Architect systems to include safeguards around process termination policies to prevent manipulation.

## 7. Vulnerability Management

- **Focus on Memory and Process Vulnerabilities:** Prioritize vulnerabilities related to memory management, process control, and DLL loading when conducting vulnerability assessments.
- **Patch Process Handling Vulnerabilities:** Ensure all patches related to process termination vulnerabilities (e.g., from the malware's use of `__acrt_get_process_end_policy`) are applied immediately.
- **Update Signatures for Known Exploits:** Ensure vulnerability scanners are updated with the latest signatures for exploits targeting memory and process management.

## 8. Identity and Access Management (IAM)

- **Limit Privileges for Memory Access:** Ensure that only necessary processes have access to memory management functions to minimize risk of abuse.
- **Monitor Privileged Access to Processes:** Keep a close watch on accounts or processes with high privileges that can manipulate process termination or memory allocation.
- **Implement Multi-Factor Authentication:** Enforce strong authentication, especially for administrators and accounts with the ability to load libraries or control processes.

## 9. Application Security

- **Secure Command-Line Input Processing:** Review application security for any vulnerabilities related to command-line input manipulation, which the malware may exploit.
- **Memory Safety in Development:** Encourage developers to use secure memory management libraries and avoid common pitfalls like buffer overflows.
- **Integrate Security Testing:** Include memory and process management checks in application security testing.



## 10. Cloud Security

- **Implement Runtime Security for Cloud Workloads:** Use cloud-native runtime protection tools that monitor memory and process management in cloud environments.
- **Ensure Cloud Processes are Hardened:** Harden cloud instances against process manipulation by enforcing strict process termination and library loading policies.
- **Secure Cloud Storage for Libraries:** Ensure cloud storage of DLLs or other libraries is secure, preventing malicious injections or replacements.

## 11. Data Security

- **Protect Sensitive Memory Regions:** Secure areas of memory that hold sensitive information, ensuring they are not accessible by unauthorized processes.
- **Data Integrity During Shutdown:** Implement mechanisms to ensure that sensitive data is not exfiltrated or tampered with during system shutdown processes.
- **Monitor Memory Use in Critical Systems:** Increase monitoring on systems handling sensitive data, focusing on memory allocations that could host hidden payloads.

## 12. Security Awareness and Training

- **Train Staff on Process and Memory Attacks:** Educate technical teams on the risks associated with memory and process manipulation, especially attacks that target system libraries and termination policies.
- **Include DLL Hijacking in Training:** Conduct awareness programs on DLL hijacking and injection threats and how they can be mitigated.
- **Incident Response Training:** Regularly train SOC and incident response teams on detecting and responding to malware that exploits memory and process functions.

## 13. Forensics and Malware Analysis

- **Enhance Memory Forensics Capabilities:** Train teams in advanced memory forensics to analyze malware that manipulates memory allocation and process termination.
- **Study Process Manipulation Techniques:** Focus on analyzing malware that prevents process termination or hijacks processes to ensure the organization can detect these techniques.
- **Deep Dive into Library Hijacking:** Ensure teams are well-versed in DLL hijacking analysis and understand how malicious libraries are loaded into memory (alloc codes)(Library code)(Memory codes).

## 14. DevSecOps

- **Integrate Security in Memory Handling During DevOps:** Ensure that security is integrated into every step of the development process, particularly in areas related to memory and process handling.
- **Automate Security Checks for Process and Memory Manipulation:** Automate checks for vulnerabilities in memory allocation and process termination in CI/CD pipelines.
- **Secure Libraries in DevOps Processes:** Ensure libraries used during the build and deployment phases are secure and not vulnerable to hijacking or manipulation.

## Conclusion

In today's ever-evolving threat landscape, organizations are increasingly vulnerable to advanced malware techniques that target fundamental system functions such as memory allocation, process termination, and dynamic library loading. The malware analyzed in this investigation demonstrates an impressive level of sophistication, leveraging system-level manipulations to evade detection, maintain persistence, and potentially deliver malicious payloads. Through detailed analysis of the provided code files, we have identified several key behavioral patterns and vulnerabilities that must be addressed by different departments within the organization to ensure robust defenses against such threats.

## Advanced Malware Purpose and Targeted Systems

The core purpose of this malware revolves around the manipulation of system memory and processes. By intercepting or altering standard functions like `__calloc_base`, `__realloc_base`, and other critical memory-related processes, the malware seeks to control how memory is allocated, resized, and deallocated across the system. This not only provides an opportunity for injecting malicious payloads but also ensures that memory blocks holding these payloads can remain undetected. Additionally, the malware's ability to manipulate process termination routines via functions like `__acrt_get_process_end_policy` grants it persistence by preventing the termination of critical or malicious processes.

Another crucial component of the malware is its reliance on dynamic library loading, specifically through functions such as `try_load_library_from_system_directory`. This technique likely allows the malware to replace or load malicious versions of system libraries, thereby granting it access to critical system resources without raising alarms. DLL hijacking, a common vector in advanced malware attacks, allows the malware to insert itself into trusted processes, further evading detection by security tools and analysts.

## System Interaction and Exploitation

Throughout its execution, the malware engages in direct interactions with fundamental system processes. By intercepting command-line arguments through functions such as `__get_wide_winmain_command_line`, the malware manipulates how applications receive and process input, potentially delivering payloads or redirecting system operations. This manipulation of the command-line interface enables the malware to blend in with legitimate system operations, making it harder to distinguish between normal and malicious behavior.

In addition to command-line manipulation, the malware directly engages with the system's memory management routines. Functions like `__msize_base` are designed to manage memory sizes, ensuring that memory blocks are properly sized and allocated. While these processes may appear benign on the surface, the underlying manipulation of memory resources opens the door for memory corruption, buffer overflows, and the hiding of malicious payloads. This points to the malware's broader goal of exploiting vulnerabilities in system memory management to evade detection and maximize its impact.

Process termination control is another critical component of this malware's attack strategy. By altering termination policies and using functions like

`__acrt_AppPolicyGetProcessTerminationMethodInternal`, the malware ensures that its processes remain active even in the face of detection or system shutdown. This ability to control process termination prevents system defenses from fully containing the malware and allows it to execute its payload even after being flagged by security systems. Such persistence mechanisms are common in sophisticated malware, allowing it to remain undetected for longer periods and continue executing its malicious operations.

## Potential Damage and Impact

The potential damage and impact of this malware are far-reaching. Its manipulation of memory allocation and process termination policies suggests the possibility of widespread system instability. By controlling how memory is allocated and resized, the malware could trigger buffer overflows or memory corruption, leading to system crashes or the exploitation of additional vulnerabilities. This opens the door for lateral movement within the organization's network, as the malware could potentially infect other systems by exploiting weaknesses in memory management.

Furthermore, the malware's ability to load malicious libraries into trusted processes through dynamic library loading increases the risk of compromised system integrity. By inserting malicious DLLs into system operations, the malware can effectively hijack system resources and perform unauthorized actions, such as data exfiltration, unauthorized access, or further exploitation of system vulnerabilities. This capability poses a significant threat to sensitive information and critical system processes, as it allows the malware to evade detection while executing its operations under the guise of legitimate processes.

## Exploitation and Vulnerability Targeting

The malware demonstrates a clear focus on exploiting vulnerabilities related to memory management, process control, and dynamic library loading. Its use of functions like `__calloc_base` and `__realloc_base` targets memory-related vulnerabilities, allowing the malware to manipulate how memory is allocated, resized, and released. These manipulations increase the risk of buffer overflows, memory corruption, and the injection of malicious code into system processes.

Additionally, the malware targets vulnerabilities in process termination policies. By intercepting system calls related to process termination, the malware ensures that its processes remain active and evade detection, even in the event of system shutdowns. This targeting of process termination vulnerabilities is a hallmark of persistent malware, as it allows the malware to maintain its presence on the system despite attempts to terminate its processes.

Finally, the malware's reliance on dynamic library loading points to its exploitation of DLL hijacking vulnerabilities. By loading malicious libraries into trusted system processes, the malware can effectively hijack system resources and evade detection by security tools. This

exploitation of DLL loading vulnerabilities increases the risk of unauthorized access and control over critical system functions.

## Recommendations for Defense and Mitigation

Given the sophisticated techniques employed by this malware, a multi-layered defense strategy is essential to mitigate the risks it poses. Different departments within the organization must play a critical role in strengthening security measures and enhancing their ability to detect and respond to such advanced threats.

1. **Security Operations Center (SOC):** Enhanced monitoring of memory allocations and process activities is essential to detect unusual behavior. SOC teams should set up alerts for processes that exhibit abnormal termination patterns or manipulate memory resources in unexpected ways. Regular memory audits will help identify hidden processes or malicious payloads embedded within allocated memory blocks.
2. **Threat Intelligence:** Intelligence gathering efforts should focus on malware that exploits memory and process vulnerabilities, as well as emerging DLL hijacking techniques. Maintaining a threat profile on malware that engages in system-level manipulations will help the organization stay ahead of evolving threats.
3. **Governance, Risk, and Compliance (GRC):** Policies surrounding secure memory management and process handling should be strengthened, and regular audits of these processes must be mandated. Aligning organizational policies with industry standards, such as ISO 27001, will help ensure that best practices for system security are followed.
4. **Penetration Testing and Red Teaming:** Simulating memory allocation and process termination exploits during red team exercises will provide valuable insights into the organization's ability to detect and respond to these threats. Testing for DLL hijacking and injection vulnerabilities will help identify weaknesses in system defenses.
5. **Incident Response (IR):** Incident response teams should focus on enhancing memory forensics capabilities, enabling them to detect hidden payloads and suspicious process behaviors. Automating checks during system shutdowns will help detect malicious activities that occur during these events.
6. **Security Architecture:** System architectures should be hardened to prevent unauthorized memory allocations and DLL injections. Implementing safeguards around process termination policies will help prevent malware from manipulating termination routines to maintain persistence.
7. **Vulnerability Management:** Prioritizing vulnerabilities related to memory and process management will help reduce the risk of exploitation. Ensuring that all relevant patches are applied in a timely manner is crucial to preventing malware from leveraging known vulnerabilities.
8. **Identity and Access Management (IAM):** Access to memory and process management functions should be restricted to only those processes that require it. Implementing multi-factor authentication for privileged accounts will reduce the risk of unauthorized access to system resources.

In conclusion, this malware represents a sophisticated and multifaceted threat that targets fundamental system processes to achieve persistence, evade detection, and execute its payloads. By manipulating memory allocation, process termination, and dynamic library loading, the malware is able to compromise system integrity while maintaining a low profile. A coordinated effort across all departments is required to mitigate the risks posed by this malware and to ensure the organization is well-prepared to defend against similar threats in the future. A proactive, multi-layered defense strategy that focuses on memory management, process control, and system architecture will be key to ensuring long-term security and resilience against advanced malware attacks.