Malware Analysis Report

MALWARE bazaar

Subject: StealthRAT Advanced Malware Analysis and Threat Assessment

Malware Sample Source: Malware Bazaar

Sample Download link

Date: October 1, 2024

Made By

LinkedIn: Engineer.Ahmed Mansour

GitHub link

Table of Contents

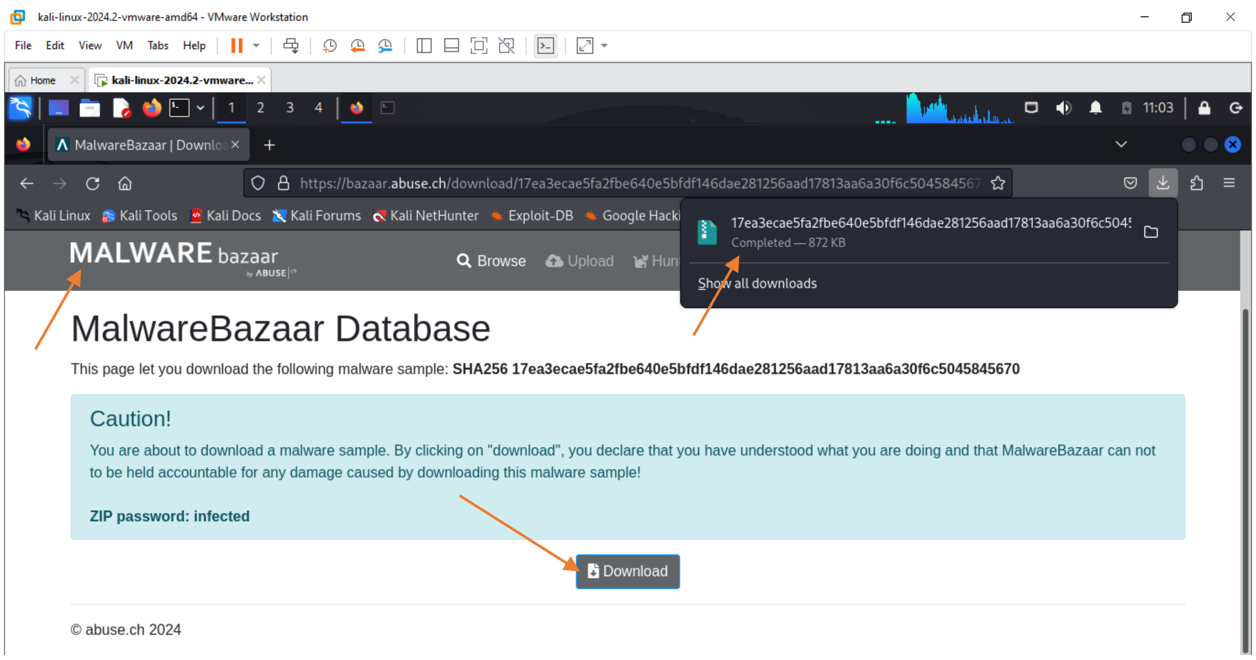
Steps to get the main code : I will be sharing a detailed, step-by-step breakdown of my approach to analyzing this malware. Stay tuned for insights on the investigation process and containment strategies.

1	**Download the malware sample from** malwarebazaar.com **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	**Develop tailored** recommendations **for each department within the organization based on the findings to mitigate future risks.**
7	**Conclusion**

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

We launched **Kali Linux** and proceeded to download the malware sample for in-depth analysis.

Please refer to the attached photo for further details.


kali-linux-2024.2-vmware-amd64 - VMware Workstation
File Edit View VM Tabs Help
Home
kali-linux-2024.2-vmware...
11:03
MalwareBazaar | Downlo
Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB Google Hacki
Completed — 872 KB
Show all downloads
MALWARE bazaar
Browse Upload Hun

https://bazaar.abuse.ch/download/17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c50458456

17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c504

MalwareBazaar Database

This page let you download the following malware sample: **SHA256 17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670**

Caution!

You are about to download a malware sample. By clicking on "download", you declare that you have understood what you are doing and that MalwareBazaar can not to be held accountable for any damage caused by downloading this malware sample!

ZIP password: infected

Download

© abuse.ch 2024

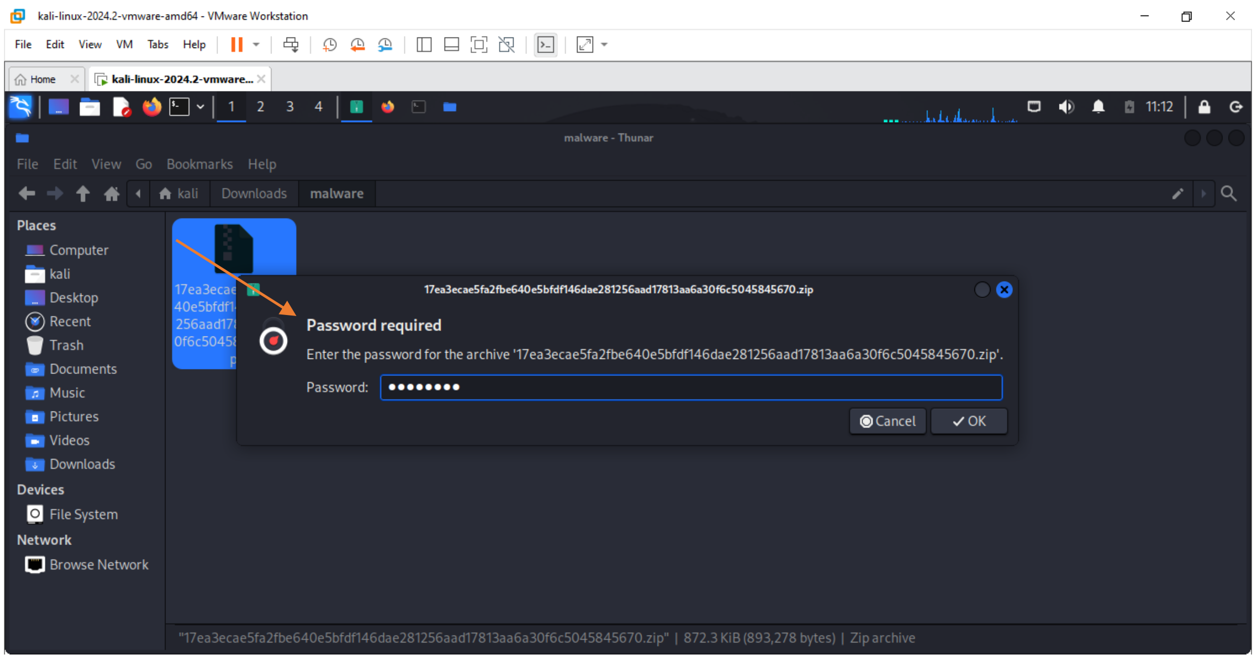
Here is the link to the **malware sample** for reference.

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.



17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670.zip

Password required

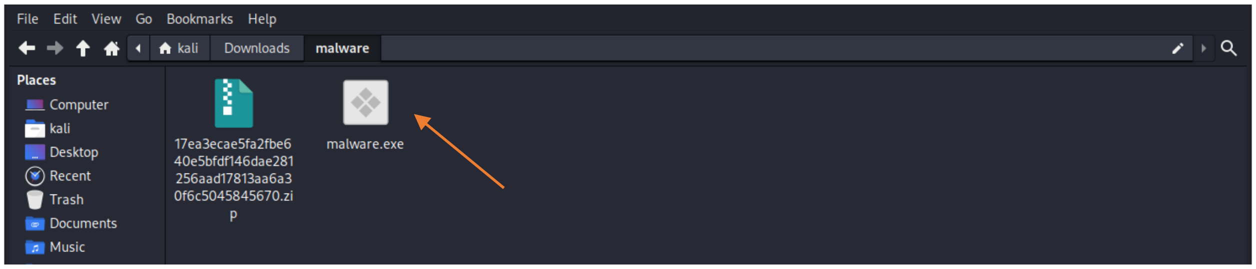
Enter the password for the archive '17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670.zip'.

Password: ••••••••

"17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670.zip" | 872.3 KiB (893,278 bytes) | Zip archive

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.



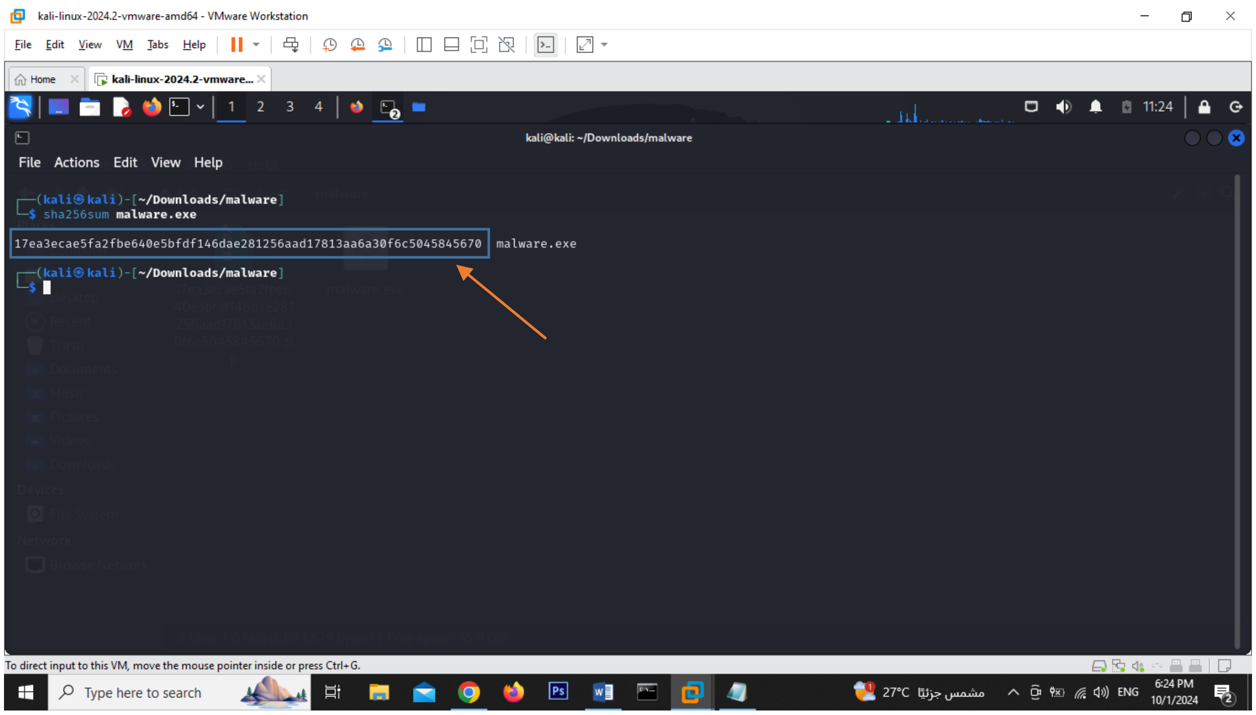
17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670.zip

malware.exe

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



kali-linux-2024.2-vmware-amd64 - VMware Workstation
kali@kali: ~/Downloads/malware
File Actions Edit View Help

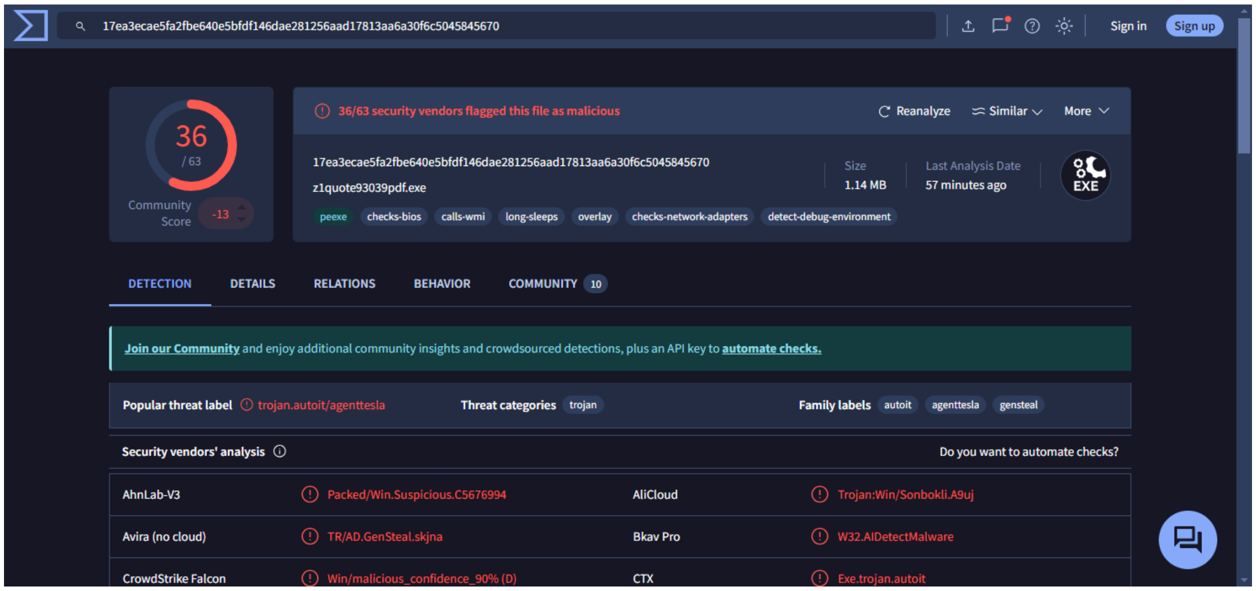
```
┌──(kali㉿kali)-[~/Downloads/malware]
└─$ sha256sum malware.exe
17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670  malware.exe

┌──(kali㉿kali)-[~/Downloads/malware]
└─$ 
```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.
Type here to search
6:24 PM
10/1/2024

The **SHA-256 hash is :**
17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670

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

I will utilize **VirusTotal.com** for comprehensive Threat Intelligence. By entering the malware's hash, I will gather and analyze as much information as possible to aid in the investigation.


17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670

36 / 63

Community Score -13

36/63 security vendors flagged this file as malicious

17ea3ecae5fa2fbe640e5bfdf146dae281256aad17813aa6a30f6c5045845670

z1quote93039pdf.exe

peexe checks-bios calls-wmi long-sleeps overlay checks-network-adapters detect-debug-environment

Size 1.14 MB

Last Analysis Date 57 minutes ago

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label trojan.autoit/agenttesla | Threat categories trojan | Family labels autoit agenttesla gensteal

Security vendors' analysis | Do you want to automate checks?

AhnLab-V3	Packed/Win.Suspicious.C5676994	AliCloud	Trojan:Win/Sonbokli.A9uj
Avira (no cloud)	TR/AD.GenSteal.skjna	Bkav Pro	W32.AIDetectMalware
CrowdStrike Falcon	Win/malicious_confidence_90% (D)	CTX	Exe.trojan.autoit

Detection Section:

1. **File Detected by Multiple Vendors**:
   - **36 out of 63** security vendors flagged this file as malicious, indicating a high level of threat associated with it.
2. **Popular Threat Label**:
   - The malware is commonly labeled as **trojan.autoit/agenttesla**, suggesting it belongs to the **AgentTesla** trojan family, a well-known spyware and credential-stealing malware.

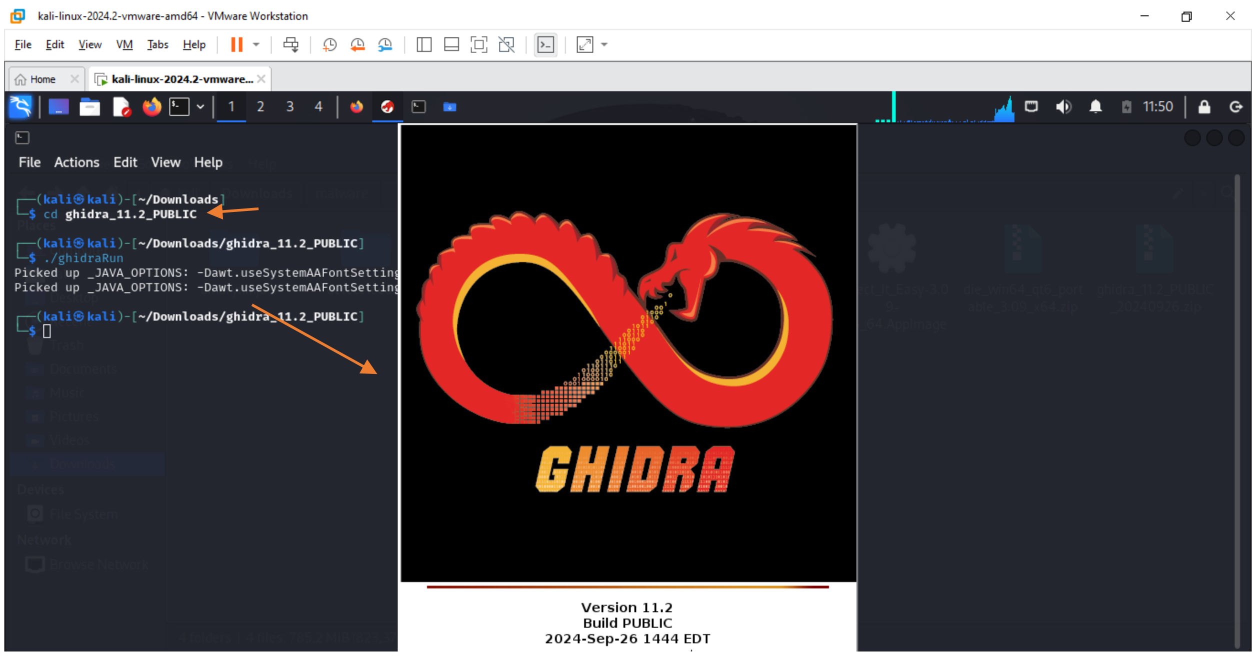
For detailed results from the **Detection Section**, including vendor-specific analyses and threat intelligence regarding the malware, you can view the comprehensive report on **VirusTotal** via the following link:

VirusTotal Detection Report(VirusTotal).

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.

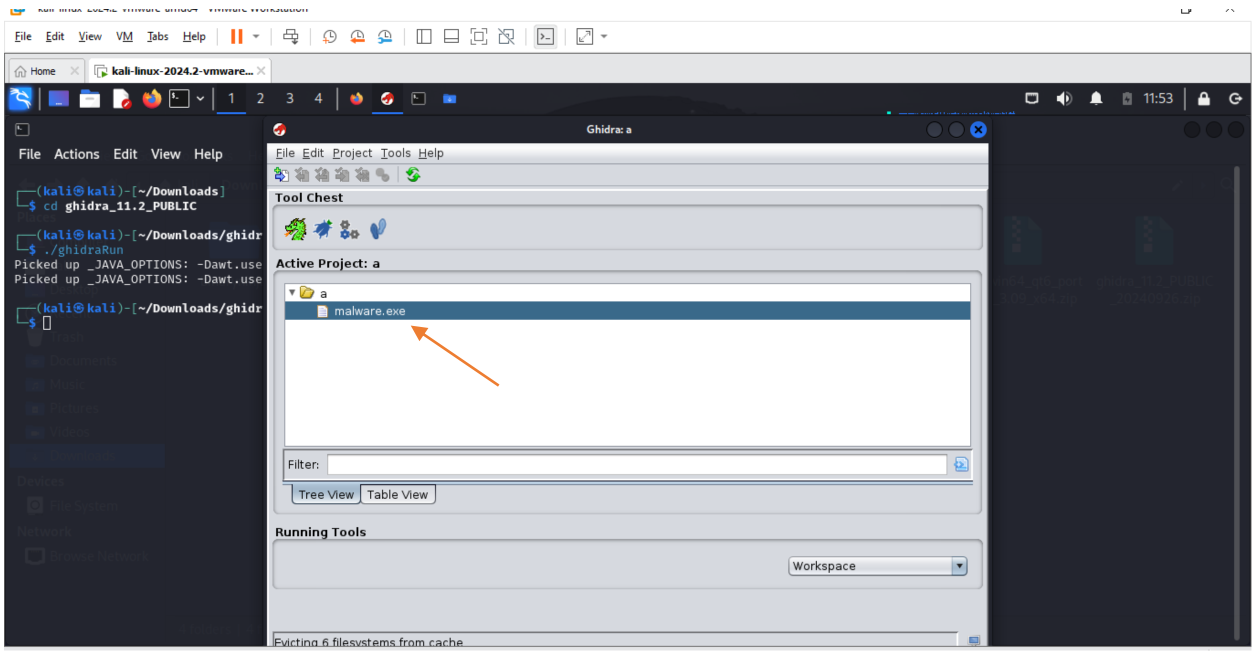

kali-linux-2024.2-vmware-amd64 - VMware Workstation
File Edit View VM Tabs Help
Home
kali-linux-2024.2-vmware...
File Actions Edit View Help
GHIDRA

```
┌──(kali㉿kali)-[~/Downloads]
└─$ cd ghidra_11.2_PUBLIC

┌──(kali㉿kali)-[~/Downloads/ghidra_11.2_PUBLIC]
└─$ ./ghidraRun
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSetting
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSetting

┌──(kali㉿kali)-[~/Downloads/ghidra_11.2_PUBLIC]
└─$ 
```

Version 11.2
Build PUBLIC
2024-Sep-26 1444 EDT

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.


File Edit View VM Tabs Help
Home
kali-linux-2024.2-vmware...
File Actions Edit View Help
Ghidra: a
File Edit Project Tools Help
Tool Chest
Tree View
Table View
Running Tools
Workspace

```
┌──(kali㉿kali)-[~/Downloads]
└─$ cd ghidra_11.2_PUBLIC

┌──(kali㉿kali)-[~/Downloads/ghidr
└─$ ./ghidraRun
Picked up _JAVA_OPTIONS: -Dawt.use
Picked up _JAVA_OPTIONS: -Dawt.use

┌──(kali㉿kali)-[~/Downloads/ghidr
└─$ 
```

Active Project: a
- a
  - malware.exe

Filter:

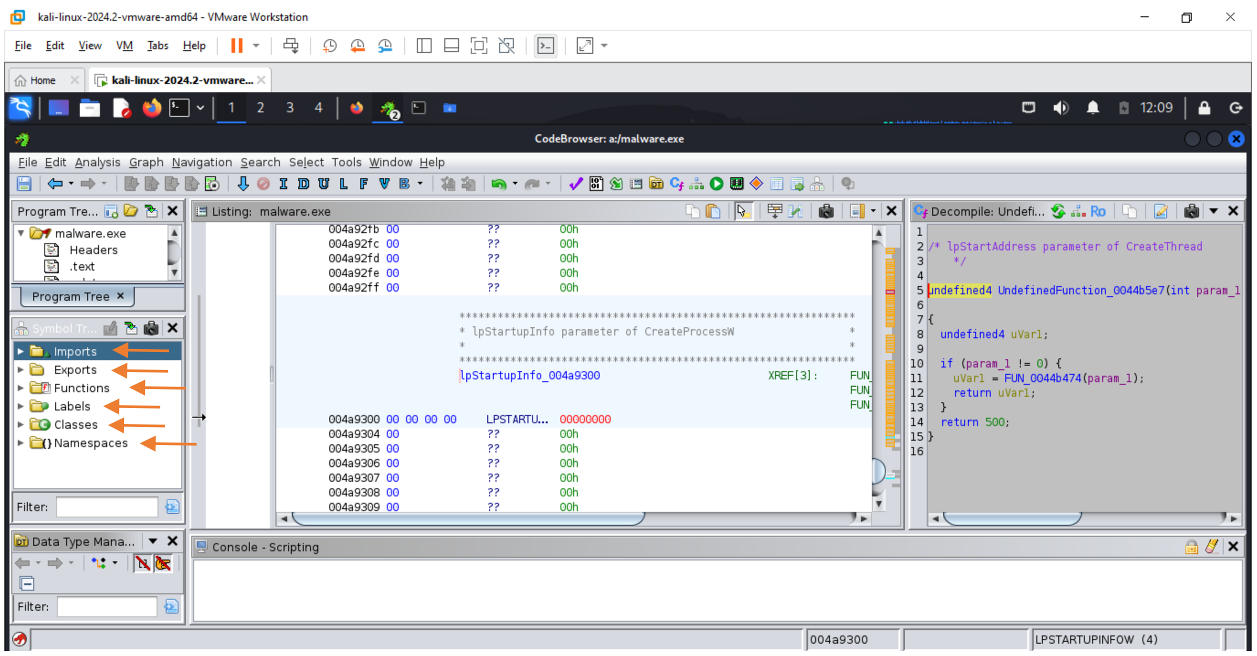
Evicting 6 filesystems from cache

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.



```
004a92fb 00          ??          00h
004a92fc 00          ??          00h
004a92fd 00          ??          00h
004a92fe 00          ??          00h
004a92ff 00          ??          00h

                 ****************************************************************
                 * lpStartupInfo parameter of CreateProcessW                    *
                 *                                                              *
                 ****************************************************************
                 lpStartupInfo_004a9300                        XREF[3]:     FUN_
                                                                            FUN_
                                                                            FUN_
004a9300 00 00 00 00    LPSTARTU...  00000000
004a9304 00          ??          00h
004a9305 00          ??          00h
004a9306 00          ??          00h
004a9307 00          ??          00h
004a9308 00          ??          00h
004a9309 00          ??          00h
```

```

/* lpStartAddress parameter of CreateThread
    */

undefined4 UndefinedFunction_0044b5e7(int param_1

{
  undefined4 uVar1;

  if (param_1 != 0) {
    uVar1 = FUN_0044b474(param_1);
    return uVar1;
  }
  return 500;
}

```

```
004a9300          LPSTARTUPINFOW (4)
```

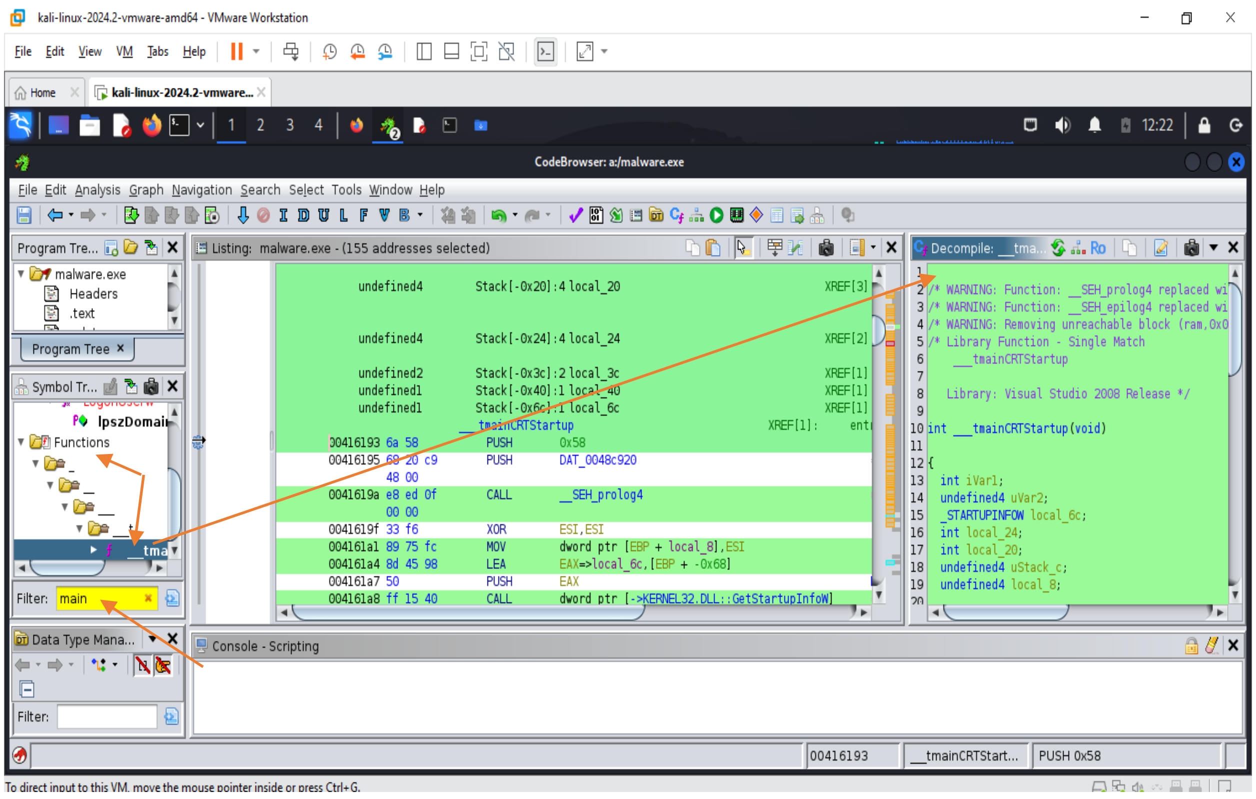
We will begin our analysis by utilizing the **filter box** in Ghidra to search for key functions that may lead us to the main code. Specifically, we will search using the following keywords:

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 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.


kali-linux-2024.2-vmware-amd64 - VMware Workstation
CodeBrowser: a:/malware.exe
Program Tree
malware.exe
Headers
.text
Symbol Tree
lpszDomain
Functions
Filter: main
Data Type Manager
Listing: malware.exe - (155 addresses selected)
Decompile: __tma...
Console - Scripting
00416193
__tmainCRTStart...
PUSH 0x58
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

```
            undefined4      Stack[-0x20]:4 local_20                     XREF[3]
            undefined4      Stack[-0x24]:4 local_24                     XREF[2]
            undefined2      Stack[-0x3c]:2 local_3c                     XREF[1]
            undefined1      Stack[-0x40]:1 local_40                     XREF[1]
            undefined1      Stack[-0x6c]:1 local_6c                     XREF[1]
                         ___tmainCRTStartup                     XREF[1]:     ent
        00416193 6a 58          PUSH      0x58
        00416195 68 20 c9       PUSH      DAT_0048c920
                 48 00
        0041619a e8 ed 0f       CALL      __SEH_prolog4
                 00 00
        0041619f 33 f6          XOR       ESI,ESI
        004161a1 89 75 fc       MOV       dword ptr [EBP + local_8],ESI
        004161a4 8d 45 98       LEA       EAX=>local_6c,[EBP + -0x68]
        004161a7 50             PUSH      EAX
        004161a8 ff 15 40       CALL      dword ptr [->KERNEL32.DLL::GetStartupInfoW]
```

```

/* WARNING: Function: __SEH_prolog4 replaced wi
/* WARNING: Function: __SEH_epilog4 replaced wi
/* WARNING: Removing unreachable block (ram,0x0
/* Library Function - Single Match
    ___tmainCRTStartup

   Library: Visual Studio 2008 Release */

int ___tmainCRTStartup(void)

{
  int iVar1;
  undefined4 uVar2;
  _STARTUPINFOW local_6c;
  int local_24;
  int local_20;
  undefined4 uStack_c;
  undefined4 local_8;
```

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

```
/* WARNING: Function:
__SEH_prolog4 replaced with
injection: SEH_prolog4 */

/* WARNING: Function:
__SEH_epilog4 replaced with
injection: EH_epilog3 */

/* WARNING: Removing
unreachable block
(ram,0x004161fa) */

/* Library Function - Single Match

    ___tmainCRTStartup

   Library: Visual Studio 2008
Release */

int ___tmainCRTStartup(void)

{

  int iVar1;

  undefined4 uVar2;

  _STARTUPINFOW local_6c;

  int local_24;

  int local_20;

  undefined4 uStack_c;

  undefined4 local_8;

  uStack_c = 0x41619f;

  local_8 = 0;

  GetStartupInfoW(&local_6c);

  local_8 = 0xfffffffe;

  local_20 = 0;

  iVar1 = __heap_init();

  if (iVar1 == 0) {
```

Part 1

```
    _fast_error_exit(0x1c);

  }

  iVar1 = __mtinit();

  if (iVar1 == 0) {

    _fast_error_exit(0x10);

  }

  __RTC_Initialize();

  local_8 = 1;

  iVar1 = __ioinit();

  if (iVar1 < 0) {

    __amsg_exit(0x1b);

  }

  DAT_004a94e4 =
GetCommandLineW();

  DAT_00496710 =
___crtGetEnvironmentStringsW();

  iVar1 = __wsetargv();

  if (iVar1 < 0) {

    __amsg_exit(8);

  }

  iVar1 = __wsetenvp();

  if (iVar1 < 0) {

    __amsg_exit(9);

  }
```

Part 2

```
  iVar1 = __cinit(1);

  if (iVar1 != 0) {

    __amsg_exit(iVar1);

  }

  uVar2 = __wwincmdln();

  local_24 =
FUN_0040d7f0(0x400000,0,uVar2
);

  if (local_20 == 0) {

                    /* WARNING:
Subroutine does not return */

    _exit(local_24);

  }

  __cexit();

  return local_24;

}
```

Part 3

The function, `___tmainCRTStartup`, is the **entry point** for a C/C++ console application compiled using **Visual Studio 2008**. It is responsible for initializing the runtime environment and setting up critical system resources for the application to execute. Below is a breakdown of its operations:

Function Breakdown:

1. **GetStartupInfoW**:
   - The function retrieves information about how the application was started (window size, state, handles, etc.). This information is passed into the `local_6c` structure.
2. **Heap Initialization**:
   - The function calls `__heap_init()` to initialize the heap for dynamic memory allocation. If this fails (`iVar1 == 0`), it triggers a fast error exit with the `_fast_error_exit()` function, which would terminate the program.
3. **Multithreading Initialization**:
   - The `__mtinit()` function is called to initialize multithreading support. If multithreading fails to initialize, the program exits immediately with an error.
4. **Runtime Checks and IO Initialization**:
   - The function calls `__RTC_Initialize()` to set up runtime checks, often used to detect issues like buffer overflows during the execution of a program.
   - The `__ioinit()` function initializes standard I/O streams such as `stdin`, `stdout`, and `stderr`. If this initialization fails, the program exits.
5. **Command-Line and Environment Setup**:
   - `GetCommandLineW()` retrieves the command-line arguments passed to the program.
   - `___crtGetEnvironmentStringsW()` retrieves the environment variables for the application.
   - Functions `__wsetargv()` and `__wsetenvp()` are used to set up the `argv` array (command-line arguments) and `envp` (environment variables).
6. **Program Initialization**:
   - The function `__cinit(1)` handles the initialization of the C runtime library (CRT). It prepares the application for execution by setting up runtime libraries.
7. **Application Execution**:
   - The `FUN_0040d7f0()` function is the actual main application logic, which is invoked with parameters like `0x400000`. The return value of this function (`local_24`) is used as the program's exit code.
8. **Program Termination**:
   - If `local_20 == 0`, the program exits using the `_exit()` function.
   - If everything runs smoothly, `__cexit()` is called to perform any final cleanup before the application terminates.

Key Insights:

- The function handles basic setup and initialization routines that are crucial for any C/C++ application.
- It ensures that key system resources such as memory, multithreading, and input/output streams are set up before the program begins its execution.
- If any part of the initialization fails, it terminates the program with an appropriate error message.

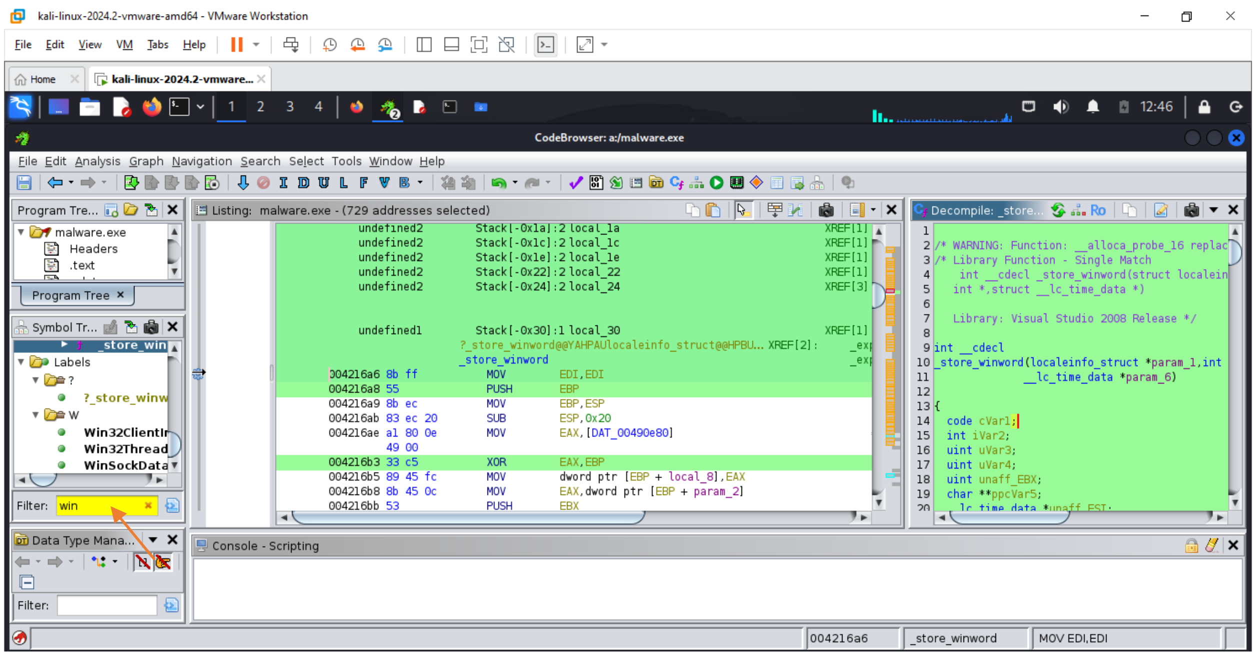
Potential Malicious Behavior:

Given that this is a common startup routine for applications compiled with Visual Studio, it doesn't appear malicious by itself. However, malware could use similar startup functions to execute before the main payload is delivered. You should investigate further functions called after `___tmainCRTStartup` to look for any abnormal or suspicious activity, such as network connections, process injections, or file manipulations.

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 9 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.


kali-linux-2024.2-vmware-amd64 - VMware Workstation
File Edit View VM Tabs Help
Home
kali-linux-2024.2-vmware...
12:46
CodeBrowser: a:/malware.exe
File Edit Analysis Graph Navigation Search Select Tools Window Help
Program Tre...
malware.exe
Headers
.text
Program Tree
Symbol Tr...
_store_win
Labels
?
?_store_winw
W
Win32ClientIr
Win32Thread
WinSockData
Filter: win
Data Type Mana...
Filter:
Listing: malware.exe - (729 addresses selected)
Decompile: _store...
Console - Scripting
004216a6
_store_winword
MOV EDI,EDI

```
     undefined2        Stack[-0x1a]:2 local_1a                         XREF[1]
     undefined2        Stack[-0x1c]:2 local_1c                         XREF[1]
     undefined2        Stack[-0x1e]:2 local_1e                         XREF[1]
     undefined2        Stack[-0x22]:2 local_22                         XREF[1]
     undefined2        Stack[-0x24]:2 local_24                         XREF[3]


     undefined1        Stack[-0x30]:1 local_30                         XREF[1]
                   ?_store_winword@@YAHPAUlocaleinfo_struct@@HPBU... XREF[2]:     _exp
                   _store_winword                                                 _exp
004216a6 8b ff          MOV        EDI,EDI
004216a8 55             PUSH       EBP
004216a9 8b ec          MOV        EBP,ESP
004216ab 83 ec 20       SUB        ESP,0x20
004216ae a1 80 0e       MOV        EAX,[DAT_00490e80]
         49 00
004216b3 33 c5          XOR        EAX,EBP
004216b5 89 45 fc       MOV        dword ptr [EBP + local_8],EAX
004216b8 8b 45 0c       MOV        EAX,dword ptr [EBP + param_2]
004216bb 53             PUSH       EBX
```

```

/* WARNING: Function: __alloca_probe_16 replac
/* Library Function - Single Match
    int __cdecl _store_winword(struct localein
   int *,struct __lc_time_data *)

   Library: Visual Studio 2008 Release */

int __cdecl
_store_winword(localeinfo_struct *param_1,int
                    __lc_time_data *param_6)

{
  code cVar1;
  int iVar2;
  uint uVar3;
  uint uVar4;
  uint unaff_EBX;
  char **ppcVar5;
   lc time data *unaff ESI;
```

The 9 codes retrieved from the **Win** keyword search in the filter is as follows:

Code1:

```
/* Library Function - Single Match

@_EH4_GlobalUnwind@4


Library: Visual Studio 2008
Release */


void __fastcall
@_EH4_GlobalUnwind@4(PVOI
D param_1)


{


RtlUnwind(param_1,(PVOID)0x41
e4fb,(PEXCEPTION_RECORD)0x
0,(PVOID)0x0);

return;

}
```

The provided function, `@_EH4_GlobalUnwind@4`, is part of the **exception handling** mechanism in programs compiled with **Visual Studio 2008**. This particular function is involved in **unwinding the stack** when an exception occurs. Here's a breakdown of what it does:

Function Breakdown:

1. **RtlUnwind**:
   - The function `RtlUnwind` is part of the Windows API and is responsible for **unwinding the stack** when an exception occurs. Unwinding means that it cleans up the call stack and reverses the effects of function calls made before the exception. This is critical for error recovery, especially in structured exception handling (SEH).
   - The parameters passed to `RtlUnwind` in this case indicate that the function:
     - **param_1**: Refers to the context where the exception occurred.
     - **(PVOID)0x41e4fb**: Points to the instruction where the unwinding should start (the return address).
     - **(PEXCEPTION_RECORD)0x0**: No specific exception record is passed, meaning a generic exception or error scenario.
     - **(PVOID)0x0**: The final argument is unused in this case.
2. **Purpose**:
   - This function helps **reset the stack** and prepares it for error handling. This is typical in applications where exceptions need to be caught, processed, or logged without crashing the program.
3. **Visual Studio Exception Handling**:
   - In **Visual Studio 2008** applications, exception handling is based on a model known as **SEH (Structured Exception Handling)**. This function is part of that model, making it responsible for correctly handling exceptions and preventing undefined behavior.

Key Insights:

- **Normal Functionality**: This function itself is standard in many applications and is part of the built-in mechanisms for handling errors in programs. It does not indicate malicious behavior directly.
- **Potential Use in Malware**: While the function is benign, malware could use exception handling mechanisms like this to avoid crashes, handle errors silently, or even hide the malware's operations. By catching exceptions, malware can sometimes evade detection or prevent analysis tools from crashing the program and exposing malicious activity.

Code2:

```
/* Library Function - Single Match
    @_EH4_LocalUnwind@16

   Library: Visual Studio 2008
Release */

void __fastcall
@_EH4_LocalUnwind@16(int
param_1,uint param_2,undefined4
param_3,uint *param_4)

{

__local_unwind4(param_4,param_
1,param_2);
  return;
}
```

The function `@_EH4_LocalUnwind@16` is part of the **Structured Exception Handling (SEH)** in programs compiled using **Visual Studio 2008**. This function deals specifically with **local stack unwinding** during exception handling.

Function Breakdown:

1. **Parameters**:
   - **param_1**: An integer that likely represents a control value or state during the unwinding process.
   - **param_2**: A `uint` value that might represent the number of frames to unwind or the condition under which to unwind.
   - **param_3**: This parameter is unused or has a standard value (`undefined4`), possibly used for internal purposes.
   - **param_4**: A pointer to a structure or the base address for the local stack frame involved in the unwinding.
2. **__local_unwind4**:
   - This internal function (`__local_unwind4`) is responsible for unwinding the local stack. In other words, it cleans up the stack frame by rolling back the local variables and any resources allocated within that function scope.
   - The **unwind process** happens when an exception occurs, ensuring that resources like memory, file handles, or other system resources are properly released, and the program can continue to function or exit gracefully.
3. **Purpose**:
   - **Local Unwind**: This function performs a more localized unwinding compared to global unwinding. It's focused on the current function or block of code where the exception was thrown, unlike **global unwind**, which affects multiple stack frames.
   - It is a crucial part of the exception handling flow, especially when specific resources must be released before an exception is fully propagated or caught by another handler.

Key Insights:

- **Standard SEH Mechanism**: This function is part of the standard exception handling mechanism in **Visual Studio** and is not inherently malicious. It ensures that any local resources or variables are cleaned up when an exception is thrown in the application.
- **Malware Use**: While this function is benign in itself, malware could potentially leverage structured exception handling to prevent crashes or handle errors more silently, making analysis and detection more challenging. SEH can sometimes be abused in malware to catch unexpected errors without alerting the user or the system.

Code3:

```
/* Library Function - Single Match

    ___crtGetStringTypeA


   Library: Visual Studio 2008
Release */


BOOL __cdecl

___crtGetStringTypeA

          (_locale_t
_Plocinfo,DWORD
_DWInfoType,LPCSTR
_LpSrcStr,int _CchSrc,LPWORD
_LpCharType,

          int _Code_page,BOOL
_BError)


{

  int iVar1;

  int in_stack_00000020;

  int in_stack_ffffffec;

  int local_c;

  char local_8;



_LocaleUpdate::_LocaleUpdate((_
LocaleUpdate
*)&stack0xffffffec,_Plocinfo);

  iVar1 =
__crtGetStringTypeA_stat

                    ((localeinfo_struct
*)_DWInfoType,(ulong)_LpSrcStr,
(char *)_CchSrc,


(int)_LpCharType,(ushort
*)_Code_page,_BError,in_stack_0
0000020,
```

Part 1

```
int in_stack_ffffffec;

  int local_c;

  char local_8;



_LocaleUpdate::_LocaleUpdate((_
LocaleUpdate
*)&stack0xffffffec,_Plocinfo);

  iVar1 =
__crtGetStringTypeA_stat

                    ((localeinfo_struct
*)_DWInfoType,(ulong)_LpSrcStr,
(char *)_CchSrc,


(int)_LpCharType,(ushort
*)_Code_page,_BError,in_stack_0
0000020,

                     in_stack_ffffffec);

  if (local_8 != '\0') {

    *(uint *)(local_c + 0x70) =
*(uint *)(local_c + 0x70) &
0xfffffffd;

  }

  return iVar1;

}
```

Part 2

The function ___crtGetStringTypeA is part of the C runtime (CRT) library in **Visual Studio 2008**, and its primary purpose is to retrieve information about the types of characters in a string. This function is likely used for localization or text processing, particularly for analyzing the characteristics of a string, such as whether characters are alphabetical, numeric, control characters, etc.

Function Breakdown:

1. **Parameters**:
   - **_Plocinfo**: A pointer to a `_locale_t` structure, which contains locale-specific information. This helps in determining character types based on specific cultural or regional settings.
   - **_DWInfoType**: A `DWORD` value indicating what type of information is requested (e.g., whether the characters are digits, spaces, etc.).
   - **_LpSrcStr**: A pointer to the source string in **ASCII** format (denoted by the "A" in the function name). This string will be analyzed.
   - **_CchSrc**: The count of characters in the source string.
   - **_LpCharType**: A pointer to a buffer where the function will store the character type information (e.g., is it an alphabetic character, a digit, etc.).
   - **_Code_page**: The code page used for interpreting the characters in the string, affecting how characters are translated to their respective types.
   - **_BError**: A flag indicating whether the function should generate an error if the string cannot be processed.
2. **Locale Handling**:
   - The function begins by updating locale information using `_LocaleUpdate::_LocaleUpdate`, ensuring that the correct regional and language settings are applied when analyzing the string. This is important in multilingual applications.
3. **String Type Analysis**:
   - The call to `__crtGetStringTypeA_stat` performs the actual character type analysis. It inspects the string and, based on the `DWInfoType`, determines whether each character fits certain categories (e.g., digit, letter, punctuation).
4. **Return Value**:
   - The function returns a **BOOL** (boolean) value indicating the success or failure of the operation. If the analysis is successful, the function returns `TRUE`; otherwise, it returns `FALSE`.
5. **Conditional Flag Clearing**:
   - If the local variable `local_8` is set, the function modifies a flag at a certain location in memory. This indicates that some internal state is updated based on the results of the string analysis.

Use Case:

This function would be used in scenarios where the application needs to determine the nature of characters in a string, possibly for input validation, localization, or text parsing. It can distinguish whether characters are alphabetic, numeric, control characters, or other types, depending on the value of `_DWInfoType`.

Potential Malware Usage:

While the function itself is part of standard C runtime libraries, malware could use it to parse and analyze strings, especially if it needs to handle or manipulate data input, like command-line arguments, user inputs, or configuration data. In malware, this function could be used to interpret text-based instructions or commands.

Code4:

```
/* Library Function - Single Match
    __global_unwind2

   Library: Visual Studio */

void __cdecl
__global_unwind2(PVOID
param_1)

{

RtlUnwind(param_1,(PVOID)0x42
5de8,(PEXCEPTION_RECORD)0
x0,(PVOID)0x0);
  return;
}
```

The function `__global_unwind2` is part of the **Structured Exception Handling (SEH)** mechanism in Windows applications, particularly those compiled with the **Visual Studio** environment. This function is responsible for managing **global stack unwinding** when an exception occurs in the program. Here's a breakdown of what this code does:

Function Breakdown:

1. **Parameters**:
   - **param_1**: This parameter holds a pointer to the context or state at the time of the exception. It provides details on where the exception occurred in the program.
2. **RtlUnwind**:
   - The function `RtlUnwind` is a core Windows API call used to **unwind the stack** during an exception. It cleans up the stack by reversing the effects of function calls made before the exception. Specifically, it performs the following:
     - **param_1**: Provides the context for unwinding the stack.
     - **0x425de8**: Likely represents the return address where the unwinding process should begin.
     - **PEXCEPTION_RECORD**: No specific exception record is passed here, indicating that it handles a generic or pre-existing exception.
     - **(PVOID)0x0**: This is the final argument, used for optional context or flags, and is set to `0x0` (unused in this case).
3. **Purpose**:
   - The function is part of the error handling process. When an exception occurs, the **unwinding** process ensures that local variables, allocated memory, and other resources are correctly cleaned up as the stack frames are "rolled back" to handle the error. This prevents memory leaks or corruption and ensures that the program either exits safely or continues functioning after the exception is handled.

Key Insights:

- **Global Unwind**: This is a more comprehensive stack unwinding compared to local unwind, as it deals with multiple stack frames across different functions. It's used when the program needs to handle exceptions that span across several layers of function calls.
- **Exception Handling**: This function is part of the **structured exception handling** in Windows, ensuring that the program can gracefully recover from errors, avoid crashing, and clean up resources.

Potential Malware Usage:

Although this is a standard part of the C runtime library in Visual Studio, malware could utilize such exception handling mechanisms to ensure its execution remains stable even in the presence of errors. This is especially useful in avoiding crashes that might trigger alarms or detections. Malware may also use exception handling to mask or obfuscate its malicious behavior.

Code5:

```
/* Library Function - Single Match
    __initp_misc_winsig

   Library: Visual Studio 2008
Release */

void __cdecl
__initp_misc_winsig(undefined4
param_1)

{
  DAT_00496bdc = param_1;
  DAT_00496be0 = param_1;
  DAT_00496be4 = param_1;
  DAT_00496be8 = param_1;
  return;
}
```

The function `__initp_misc_winsig` is part of the **Visual Studio 2008** runtime library and is likely used for initializing certain global variables. Here's a detailed breakdown of what this function is doing:

Function Breakdown:

1. **Parameters**:
   - **param_1**: This is the parameter passed to the function, likely containing some important initialization value or state.
2. **Initialization of Global Variables**:
   - The function assigns the value of `param_1` to several global variables: `DAT_00496bdc`, `DAT_00496be0`, `DAT_00496be4`, and `DAT_00496be8`.
   - These `DAT_` variables are likely part of the data segment of the program and may be important flags or signals that control program behavior.
3. **Purpose**:
   - The function is simple, as it only initializes several global variables with the same value (`param_1`). These variables may later be used in the program for various purposes, such as configuration settings, signal handling, or other control mechanisms.
   - Given the function name **"misc_winsig"**, it's possible that these variables are used for **signal handling** on the Windows platform, specifically handling system signals or events.
4. **Implications**:
   - The function does not perform any complex operations beyond setting these global variables, meaning its primary purpose is likely to initialize the program's runtime environment or prepare it for handling certain types of events or exceptions.

Potential Malware Usage:

In the context of malware, initializing global variables could serve many purposes, such as setting up control flags for malicious behavior, configuring aspects of the malware's execution flow, or managing signal handling for persistence and stealth. It could be part of a larger initialization routine, where specific signals or flags determine the malware’s behavior, including deciding when to execute certain malicious functions.

Code6:

```
/* Library Function - Single
Match
    __local_unwind2


   Libraries: Visual Studio 2017
Debug, Visual Studio 2017
Release, Visual Studio 2019
Debug, Visual
   Studio 2019 Release */


void __cdecl
__local_unwind2(int
param_1,uint param_2)


{
  uint uVar1;
  void *local_20;
  undefined *puStack_1c;
  undefined4 local_18;
  int iStack_14;


  iStack_14 = param_1;
  puStack_1c = &LAB_00425df0;
  local_20 = ExceptionList;
  ExceptionList = &local_20;
  while( true ) {
    uVar1 = *(uint *)(param_1 +
0xc);
```

Part 1

```
if ((uVar1 == 0xffffffff) ||
((param_2 != 0xffffffff &&
(uVar1 <= param_2)))) break;
    local_18 = *(undefined4
*)(*(int *)(param_1 + 8) +
uVar1 * 0xc);
    *(undefined4 *)(param_1 +
0xc) = local_18;
    if (*(int *)(*(int *)(param_1 +
8) + 4 + uVar1 * 0xc) == 0) {
      __NLG_Notify(0x101);
      FUN_00425f04();
    }
  }
  ExceptionList = local_20;
  return;
}
```

Part 2

The function `__local_unwind2` is part of the **Structured Exception Handling (SEH)** mechanism commonly used in applications compiled with **Visual Studio**. This function deals with **local stack unwinding** during exception handling, meaning it is responsible for cleaning up the local stack after an exception has occurred. Let's break down the function step by step:

Function Breakdown:

1. **Parameters**:
    - **param_1**: This likely represents a control block or data structure related to the exception handling process. It contains important information such as the current stack frame and exception data.
    - **param_2**: A `uint` value that likely represents the current state or phase of the exception handling process, such as the number of stack frames to unwind.
2. **Stack Unwinding Mechanism**:
    - The function uses **ExceptionList**, which is a standard structure in SEH for maintaining the current list of exceptions. It temporarily stores the state of the `ExceptionList` and then starts processing the exception handling process.
    - The `while(true)` loop iterates through the stack frames, checking for the condition where **uVar1** (which points to the exception handler's location) is either `0xffffffff` (indicating no more handlers) or less than the current **param_2**, breaking the loop when appropriate.
3. **Local Exception Handling**:
    - During each iteration, the function retrieves exception handler information from the memory location calculated using **param_1** and **uVar1**.
    - The function then checks whether an exception handler is available for the current stack frame. If no handler exists, it calls `__NLG_Notify(0x101)` and `FUN_00425f04()`, which are internal functions that likely notify the system of the unwinding process and perform further cleanup.
4. **Restoring State**:
    - Once the stack frames are processed or the loop exits, the function restores the original `ExceptionList` to maintain consistency and complete the local unwinding process.

Purpose:

The primary purpose of this function is to **safely unwind the local stack** when an exception occurs. It ensures that each stack frame is processed, and appropriate cleanup or error handling functions are invoked. If an exception handler is found, it can execute necessary steps such as releasing memory or closing resources associated with the current function before the program moves to the next stack frame.

Potential Malware Usage:

While the function itself is part of standard Windows SEH mechanisms, malware may use exception handling routines to evade detection or avoid crashes during execution. By carefully handling exceptions, malware can remain operational and avoid errors that could otherwise terminate its execution or raise red flags in security monitoring tools.

<u>Code7:</u>

```
/* Library Function - Single Match
    __local_unwind4


   Libraries: Visual Studio 2005 Release, Visual Studio 2008 Release */


void __cdecl __local_unwind4(uint *param_1,int param_2,uint param_3)


{
  undefined4 *puVar1;
  uint uVar2;
  void *pvStack_28;
  undefined *puStack_24;
  uint local_20;
  uint uStack_1c;
  int iStack_18;
  uint *puStack_14;

  puStack_14 = param_1;
  iStack_18 = param_2;
  uStack_1c = param_3;
  puStack_24 = &LAB_0041e454;
  pvStack_28 = ExceptionList;
  local_20 = DAT_00490e80 ^ (uint)&pvStack_28;
```

Part 1

```
  ExceptionList = &pvStack_28;
  while( true ) {
    uVar2 = *(uint *)(param_2 + 0xc);
    if ((uVar2 == 0xfffffffe) || ((param_3 != 0xfffffffe && (uVar2 <= param_3)))) break;
    puVar1 = (undefined4 *)((*(uint *)(param_2 + 8) ^ *param_1) + 0x10 + uVar2 * 0xc);
    *(undefined4 *)(param_2 + 0xc) = *puVar1;
    if (puVar1[1] == 0) {
      __NLG_Notify(0x101);
      FUN_00425f04();
    }
  }
  ExceptionList = pvStack_28;
  return;
}
```

Part 2

The function `__local_unwind4` is part of the **Structured Exception Handling (SEH)** mechanism in Windows, particularly used in applications compiled with **Visual Studio 2005** and **Visual Studio 2008**. This function is responsible for managing the **unwinding** of the stack frames locally during an exception. Here's a breakdown of what the code is doing:

Function Breakdown:

1. **Parameters**:
   - **param_1**: A pointer to a structure or memory location related to the exception handling process, likely pointing to a control block.
   - **param_2**: An integer representing the starting point of the stack or the current stack frame being processed during the exception.
   - **param_3**: A `uint` that likely represents the number of stack frames to unwind or a condition used to control how far the unwinding process should go.
2. **Stack Unwinding Mechanism**:
   - The function initializes several local variables like `pvStack_28`, which holds the state of the **ExceptionList** before the unwinding process starts. The **ExceptionList** is used in SEH to maintain a chain of active exceptions.
   - The function enters a `while` loop, iterating through the stack frames starting from **param_2**. The **uVar2** variable is used to track the position in the stack, specifically pointing to the current frame's handler.
   - The loop terminates when the handler value equals `0xfffffffe`, which signifies that there are no more stack frames to process, or when the value of **param_3** is reached (if it's not `0xfffffffe`).
3. **Exception Handler Execution**:
   - During each iteration, the function calculates the location of the handler for the current stack frame, using **param_1** and **param_2**. The handler's value is updated in the control block.
   - If the second value in the handler (the function pointed to by **puVar1[1]**) is `0`, the function triggers a notification (`__NLG_Notify(0x101)`) indicating that no handler is available for this frame.
   - The function then calls `FUN_00425f04()` to manage any additional cleanup or exception handling that needs to be performed when no handler is found.
4. **Restoring Exception State**:
   - After the loop finishes, the **ExceptionList** is restored to its original state before the unwinding started. This ensures that the exception-handling state remains consistent after the local unwinding process.

Purpose:

The purpose of this function is to **unwind the local stack** during an exception handling routine. It processes each stack frame, cleans up resources (like memory or file handles), and ensures that the program can safely continue execution or terminate gracefully.

Potential Malware Use:

In the context of malware, exception handling routines like this one may be used to ensure the malware remains stable during execution. Malware might use structured exception handling to prevent crashes or recover from errors that could otherwise reveal its presence or halt its execution. By managing stack frames carefully, malware can ensure that its malicious code remains operational even when unexpected errors occur.

Code8:

```
/* Library Function - Single
Match

    __wwincmdln


   Library: Visual Studio 2008
Release */


void __wwincmdln(void)


{
  ushort uVar1;
  bool bVar2;
  ushort *puVar3;

  bVar2 = false;
  puVar3 = DAT_004a94e4;
  if (DAT_004a94e4 == (ushort
*)0x0) {
    puVar3 = (ushort
*)&lpClass_004848e8;
  }
  do {
    uVar1 = *puVar3;
```

Part 1

```
if (uVar1 < 0x21) {
      if (uVar1 == 0) {
        return;
      }
      if (!bVar2) {
        for (; (*puVar3 != 0 &&
(*puVar3 < 0x21)); puVar3 =
puVar3 + 1) {
        }
        return;
      }
    }
    if (uVar1 == 0x22) {
      bVar2 = !bVar2;
    }
    puVar3 = puVar3 + 1;
  } while( true );
}
```

Part 2

The function `__wwincmdln` is responsible for parsing command-line arguments, likely as part of a larger initialization or setup process in an application. This specific function is found in programs compiled with **Visual Studio 2008**. Below is a breakdown of the code and its operation:

Function Breakdown:

1. **Initial Variables**:
   - **puVar3**: This pointer is initially set to `DAT_004a94e4`, which seems to be a pointer to the command-line string.
   - If `DAT_004a94e4` is `0x0` (null), meaning no command-line string is provided, it defaults to a hardcoded class pointer `lpClass_004848e8`. This fallback suggests that the program is ensuring a valid pointer to a string for command-line processing.
2. **Loop through Command-Line Characters**:
   - The function enters a loop to iterate through the command-line string (or the fallback if the string is null).
   - **uVar1 = puVar3*: This line retrieves the current character being pointed to by `puVar3` for analysis.
3. **Whitespace and Quotes Handling**:
   - If the current character (pointed to by `uVar1`) is less than `0x21` (i.e., a control character or whitespace like space, tab, or null), it checks:
     - If the character is `0x0` (null terminator), the function returns, ending the command-line parsing.
     - If the character is a whitespace and the `bVar2` flag (tracking quote states) is **false**, the function skips over leading whitespace characters, ignoring them.
   - If the character is a double quote (`0x22`), the function toggles the `bVar2` flag, switching between being inside or outside a quoted section of the command line.
4. **Final Parsing**:
   - The loop continues until it reaches the end of the command-line string (`0x0`) or when all relevant characters have been processed.

Purpose:

This function parses command-line arguments while handling **quoted arguments** and **whitespace** correctly. The `bVar2` flag is used to track whether the function is inside a quoted string, allowing spaces within quotes to be treated as part of the argument, and unquoted spaces to be treated as separators between arguments.

Use Case:

In typical applications, this function would be used during program initialization to process user-supplied arguments from the command line. It ensures that any special characters, such as quotes, are handled properly, so the arguments can be correctly parsed into individual components.

Potential Malware Usage:

In malware, command-line parsing functions like this one can be critical when the malware needs to accept external inputs or configuration options from the command line. These inputs could control aspects of the malware's execution, such as which processes to target, files to manipulate, or even specific behaviors to trigger based on parameters.

Code9:

```
/* WARNING: Function: __alloca_probe_16 replaced with injection: alloca_probe */
/* Library Function - Single Match
    int __cdecl _store_winword(struct localeinfo_struct *,int,struct tm const *,char * *,unsigned
   int *,struct __lc_time_data *)

   Library: Visual Studio 2008 Release */

int __cdecl
_store_winword(localeinfo_struct *param_1,int param_2,tm *param_3,char **param_4,uint *param_5,
               __lc_time_data *param_6)

{
  code cVar1;
  int iVar2;
  uint uVar3;
  uint uVar4;
  uint unaff_EBX;
  char **ppcVar5;
  __lc_time_data *unaff_ESI;
  uint *unaff_EDI;
  code *_Str1;
  code *pcVar6;
```

Part 1

```
  char *pcVar7;
  char *pcVar8;
  short local_24;
  short local_22;
  undefined2 local_1e;
  undefined2 local_1c;
  undefined2 local_1a;
  undefined2 local_18;
  undefined2 local_16;
  int local_14;
  code *local_10;
  char **local_c;
  uint local_8;

  local_8 = DAT_00490e80 ^ (uint)&stack0xfffffffc;
  if (param_2 == 0) {
    _Str1 = (code *)param_6->ww_sdatefmt;
  }
  else if (param_2 == 1) {
    _Str1 = (code *)param_6->ww_ldatefmt;
  }
  else {
    _Str1 = (code *)param_6->ww_timefmt;
  }
```

Part 2

```
if (param_6->refcount != 1) {
    local_10 = GetDateFormatA_exref;
    if (param_2 == 2) {
      local_10 = GetTimeFormatA_exref;
    }
    local_24 = *(short *)&param_3->tm_year + 0x76c;
    local_22 = *(short *)&param_3->tm_mon + 1;
    local_1e = *(undefined2 *)&param_3->tm_mday;
    local_1c = *(undefined2 *)&param_3->tm_hour;
    local_1a = *(undefined2 *)&param_3->tm_min;
    local_18 = *(undefined2 *)&param_3->tm_sec;
    local_16 = 0;
    local_14 = (*local_10)(param_6->ww_caltype,0,&local_24,_Str1,0,0);
    if (local_14 != 0) {
      if ((int)(local_14 + 8U) < 0x401) {
        ppcVar5 = (char **)&stack0xffffffd0;
        if (&stack0x00000000 != (undefined *)0x30) {
          unaff_EDI = (uint *)0xcccc;
          ppcVar5 = (char **)&stack0xffffffd0;
```

Part 3

```
LAB_0042178a:
     ppcVar5 = ppcVar5 + 2;
    }
   }
   else {
    ppcVar5 = (char **)_malloc(local_14 + 8U);
    if (ppcVar5 != (char **)0x0) {
     *ppcVar5 = (char *)0xdddd;
     goto LAB_0042178a;
    }
   }
   local_c = ppcVar5;
   if (ppcVar5 != (char **)0x0) {
    iVar2 = (*local_10)(param_6->ww_caltype,0,&local_24,_Str1,ppcVar5,local_14);
    while ((iVar2 = iVar2 + -1, 0 < iVar2 && (*param_5 != 0))) {
     local_14 = iVar2;
     **param_4 = *(char *)ppcVar5;
     *param_4 = *param_4 + 1;
     ppcVar5 = (char **)((int)ppcVar5 + 1);
     *param_5 = *param_5 - 1;
    }
```

Part 4

```
_freea(local_c);
LAB_004217dc:
    iVar2 = @__security_check_cookie@4(local_8 ^ (uint)&stack0xfffffffc);
    return iVar2;
   }
  }
 }
 cVar1 = *_Str1;
joined_r0x004217f2:
 if ((cVar1 == (code)0x0) || (*param_5 == 0)) goto LAB_004217dc;
 local_c = (char **)0x0;
 local_10 = _Str1;
 uVar4 = 0;
 do {
  uVar3 = uVar4;
  local_10 = local_10 + 1;
  uVar4 = uVar3 + 1;
 } while (*local_10 == cVar1);
 iVar2 = (int)(char)cVar1;
if (iVar2 < 0x65) {
  if (iVar2 == 100) {
   if (uVar3 == 0) {
    local_c = (char **)0x1;
   }
```

Part 5

```
else {
joined_r0x00421975:
    if ((uVar3 != 1) && (uVar3 != 2)) {
joined_r0x0042197b:
     if (uVar3 != 3) goto LAB_00421846;
    }
   }
   goto LAB_00421add;
  }
  if (iVar2 != 0x27) {
   if (iVar2 == 0x41) {
LAB_004218ce:
    iVar2 = ___ascii_stricmp((char *)_Str1,"am/pm");
    if (iVar2 == 0) {
     local_10 = _Str1 + 5;
    }
    else {
     iVar2 = ___ascii_stricmp((char *)_Str1,"a/p");
     if (iVar2 == 0) {
      local_10 = _Str1 + 3;
     }
    }
   }
```

Part 6

```
else if (iVar2 == 0x48) {
        if (uVar3 != 0) goto
joined_r0x00421acd;
        local_c = (char **)0x1;
      }
      else {
        if (iVar2 != 0x4d) {
          if (iVar2 != 0x61) goto
LAB_00421846;
          goto LAB_004218ce;
        }
        if (uVar3 != 0) goto
joined_r0x00421975;
        local_c = (char **)0x1;
      }
      goto LAB_00421add;
    }
    _Str1 = _Str1 + uVar4;
    if ((uVar4 & 1) != 0) {
      cVar1 = *_Str1;
  if (cVar1 == (code)0x0) goto
LAB_004217dc;
      do {
        if (*param_5 == 0) break;
        if (cVar1 == (code)0x27) {
          _Str1 = _Str1 + 1;
          break;
```

Part 7

```
  }
        iVar2 =
__isleadbyte_l((int)(char)cVar1,
param_1);
        pcVar6 = _Str1;
        if ((iVar2 != 0) && (1 <
*param_5)) {
          pcVar6 = _Str1 + 1;
          if (*pcVar6 == (code)0x0)
goto LAB_004217dc;
          **param_4 =
(char)*_Str1;
          *param_4 = *param_4 +
1;
          *param_5 = *param_5 - 1;
        }
        **param_4 =
(char)*pcVar6;
        *param_4 = *param_4 + 1;
        _Str1 = pcVar6 + 1;
        *param_5 = *param_5 - 1;
        cVar1 = *_Str1;
      } while (cVar1 != (code)0x0);
    }
  }
  else {
    if (iVar2 == 0x68) {
      if (uVar3 == 0) {
        local_c = (char **)0x1;
      }
```

Part 8

```
  else {
joined_r0x00421acd:
        if (uVar4 != 2) {
LAB_00421846:
          iVar2 =
__isleadbyte_l(iVar2,param_1);
          pcVar6 = _Str1;
          if ((iVar2 != 0) && (1 <
*param_5)) {
            pcVar6 = _Str1 + 1;
            if (*pcVar6 == (code)0x0)
goto LAB_004217dc;
            **param_4 =
(char)*_Str1;
            *param_4 = *param_4 +
1;
            *param_5 = *param_5 -
1;
          }
          **param_4 =
(char)*pcVar6;
          *param_4 = *param_4 +
1;
          *param_5 = *param_5 - 1;
          _Str1 = pcVar6 + 1;
          goto LAB_0042187d;
        }
      }
    }
```

Part 9

```
  else if (iVar2 == 0x6d) {
      if (uVar3 != 0) goto
joined_r0x00421acd;
      local_c = (char **)0x1;
    }
    else if (iVar2 == 0x73) {
      if (uVar3 != 0) goto
joined_r0x00421acd;
      local_c = (char **)0x1;
    }
    else {
      if (iVar2 == 0x74) {
        if (param_3->tm_hour <
0xc) {
          pcVar8 = param_6-
>ampm[0];
        }
        else {
          pcVar8 = param_6-
>ampm[1];
        }
if ((uVar4 == 1) && (*param_5
!= 0)) {
          iVar2 =
__isleadbyte_l((int)*pcVar8,par
am_1);
          pcVar7 = pcVar8;
  if ((iVar2 != 0) && (1 <
*param_5)) {
            pcVar7 = pcVar8 + 1;
            if (*pcVar7 == '\0') goto
LAB_004217dc;
```

Part 10

```
**param_4 = *pcVar8;
            *param_4 = *param_4 +
1;
            *param_5 = *param_5 -
1;
          }
          **param_4 = *pcVar7;
          *param_4 = *param_4 +
1;
          *param_5 = *param_5 - 1;
          _Str1 = local_10;
          goto LAB_0042187d;
        }
        while ((_Str1 = local_10,
*pcVar8 != '\0' && (*param_5
!= 0))) {
          iVar2 =
__isleadbyte_l((int)*pcVar8,par
am_1);
          pcVar7 = pcVar8;
          if ((iVar2 != 0) && (1 <
*param_5)) {
            pcVar7 = pcVar8 + 1;
            if (*pcVar7 == '\0') goto
LAB_004217dc;
            **param_4 = *pcVar8;
            *param_4 = *param_4 +
1;
            *param_5 = *param_5 -
1;
          }
```

Part 11

```
**param_4 = *pcVar7;
          *param_4 = *param_4 +
1;
          pcVar8 = pcVar7 + 1;
          *param_5 = *param_5 - 1;
        }
        goto LAB_0042187d;
      }
      if (iVar2 != 0x79) goto
LAB_00421846;
      if (uVar3 != 1) goto
joined_r0x0042197b;
    }
LAB_00421add:
    iVar2 =
_expandtime(param_1,(char)pa
ram_5,(tm
*)param_6,local_c,unaff_EDI,un
aff_ESI,unaff_EBX);
    _Str1 = local_10;
    if (iVar2 == 0) goto
LAB_004217dc;
  }
LAB_0042187d:
  cVar1 = *_Str1;
  goto joined_r0x004217f2;
}
```

Part 12

The function `_store_winword` appears to be involved in formatting time and date strings according to locale-specific settings. This function handles the formatting of dates, times, or other related strings, depending on how it is called. Here's a breakdown of what the code does:

Function Breakdown:

1. **Parameters**:
   - **param_1**: Pointer to a `localeinfo_struct`, which contains locale-specific information.
   - **param_2**: Integer that determines what is being formatted (e.g., `0` for short date, `1` for long date, `2` for time).
   - **param_3**: A pointer to a `tm` structure, which holds information about the time and date (e.g., year, month, day, hour, etc.).
   - **param_4**: A pointer to a character buffer where the formatted output will be written.
   - **param_5**: Pointer to a `uint` representing the available space left in the buffer.
   - **param_6**: A pointer to a `__lc_time_data` structure, which holds locale-specific formatting data (e.g., date and time formats).
2. **Locale-Specific Formatting**:
   - Depending on `param_2`, the function chooses the appropriate format (short date, long date, or time) from the `__lc_time_data` structure.
   - The function retrieves the format from the `ww_sdatefmt`, `ww_ldatefmt`, or `ww_timefmt` fields, depending on the value of `param_2`.
3. **Date and Time Extraction**:
   - The function extracts values from the `tm` structure, which represents the date and time to be formatted.
   - For example, it extracts the year, month, day, hour, minute, and second from the `tm` structure and stores these values in local variables (`local_24`, `local_22`, etc.).
4. **Formatting and Buffer Management**:
   - The function calls `GetDateFormatA` or `GetTimeFormatA` (based on `param_2`) to format the date or time into the specified format and write it into the buffer.
   - It ensures the buffer is large enough to hold the formatted string, and if not, it dynamically allocates memory using `_malloc`.
   - It checks available space in the buffer (`param_5`) and writes the formatted string incrementally, ensuring no overflow occurs.
5. **Handling AM/PM**:
   - If the time format involves AM/PM, the function adjusts the output based on whether the hour is less than 12 (AM) or greater (PM), using the `ampm` array in the `__lc_time_data` structure.
6. **Memory Cleanup**:
   - After the formatting is complete, the function frees any dynamically allocated memory using `__freea` and performs security checks before returning.

Purpose:

The `_store_winword` function is responsible for formatting date and time strings according to locale settings. It handles tasks such as converting raw time and date values (from the `tm` structure) into a human-readable string format based on the specified locale. This function would typically be used in applications that need to display dates and times in a user-friendly format according to local preferences.

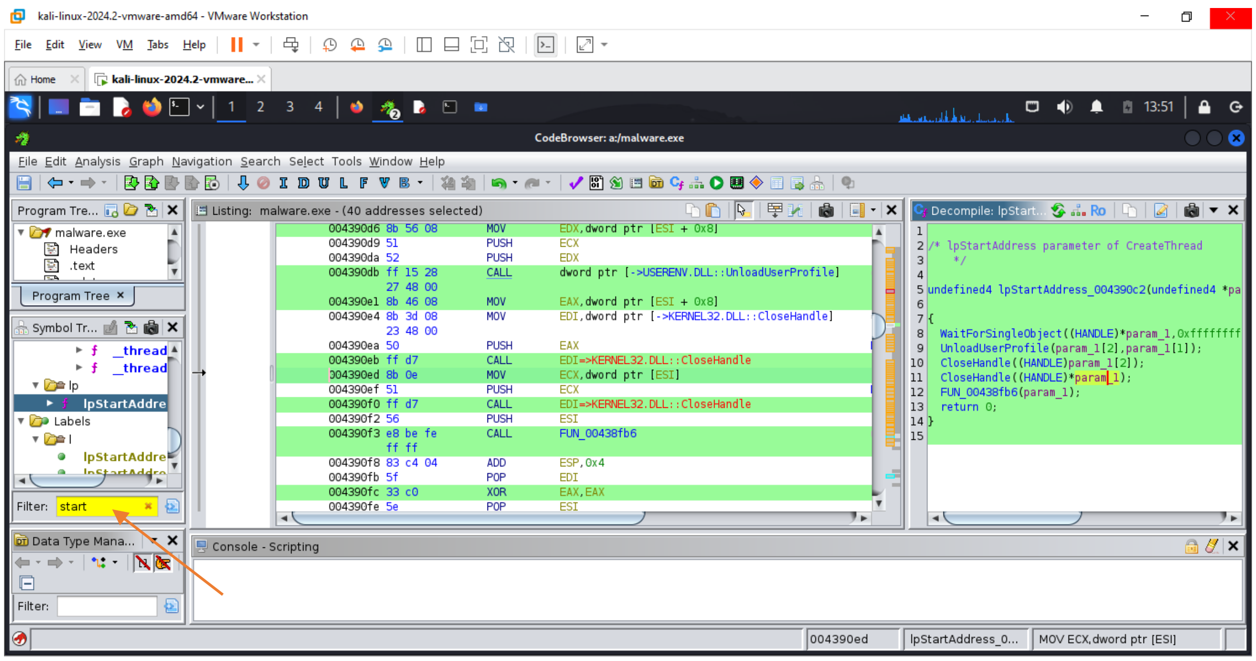
Potential Malware Use:

While this function is part of the standard C runtime library and doesn't perform any inherently malicious activities, malware could use similar functions to format and output dates and times. This could be useful if the malware logs events or communicates with command and control servers in a way that requires date/time tracking.

We will begin our analysis by searching for the **start** 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 8 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.



```
004390d6 8b 56 08    MOV    EDX,dword ptr [ESI + 0x8]
004390d9 51          PUSH   ECX
004390da 52          PUSH   EDX
004390db ff 15 28    CALL   dword ptr [->USERENV.DLL::UnloadUserProfile]
         27 48 00
004390e1 8b 46 08    MOV    EAX,dword ptr [ESI + 0x8]
004390e4 8b 3d 08    MOV    EDI,dword ptr [->KERNEL32.DLL::CloseHandle]
         23 48 00
004390ea 50          PUSH   EAX
004390eb ff d7       CALL   EDI=>KERNEL32.DLL::CloseHandle
004390ed 8b 0e       MOV    ECX,dword ptr [ESI]
004390ef 51          PUSH   ECX
004390f0 ff d7       CALL   EDI=>KERNEL32.DLL::CloseHandle
004390f2 56          PUSH   ESI
004390f3 e8 be fe    CALL   FUN_00438fb6
         ff ff
004390f8 83 c4 04    ADD    ESP,0x4
004390fb 5f          POP    EDI
004390fc 33 c0       XOR    EAX,EAX
004390fe 5e          POP    ESI
```

```

/* lpStartAddress parameter of CreateThread
   */

undefined4 lpStartAddress_004390c2(undefined4 *pa

{
  WaitForSingleObject((HANDLE)*param_1,0xffffffff
  UnloadUserProfile(param_1[2],param_1[1]);
  CloseHandle((HANDLE)param_1[2]);
  CloseHandle((HANDLE)*param_1);
  FUN_00438fb6(param_1);
  return 0;
}

```

The 8 codes retrieved from the **start** keyword search in the filter is as follows:

Code1:

```
/* WARNING: Function: __SEH_prolog4 replaced with injection: SEH_prolog4 */
/* WARNING: Function: __SEH_epilog4 replaced with injection: EH_epilog3 */
/* WARNING: Removing unreachable block (ram,0x004161fa) */
/* Library Function - Single Match
    ___tmainCRTStartup

   Library: Visual Studio 2008 Release */

int ___tmainCRTStartup(void)

{
  int iVar1;
  undefined4 uVar2;
  _STARTUPINFOW local_6c;
  int local_24;
  int local_20;
  undefined4 uStack_c;
  undefined4 local_8;
```

Part 1

```
uStack_c = 0x41619f;
  local_8 = 0;
  GetStartupInfoW(&local_6c);
  local_8 = 0xfffffffe;
  local_20 = 0;
  iVar1 = __heap_init();
  if (iVar1 == 0) {
    _fast_error_exit(0x1c);
  }
  iVar1 = __mtinit();
  if (iVar1 == 0) {
    _fast_error_exit(0x10);
  }
  __RTC_Initialize();
  local_8 = 1;
  iVar1 = __ioinit();
  if (iVar1 < 0) {
    __amsg_exit(0x1b);
  }
DAT_004a94e4 = GetCommandLineW();
  DAT_00496710 = ___crtGetEnvironmentStringsW();
  iVar1 = __wsetargv();
  if (iVar1 < 0) {
```

Part 2

```
_amsg_exit(8);
  }
  iVar1 = __wsetenvp();
  if (iVar1 < 0) {
    __amsg_exit(9);
  }
  iVar1 = __cinit(1);
  if (iVar1 != 0) {
    __amsg_exit(iVar1);
  }
  uVar2 = __wwincmdln();
  local_24 = FUN_0040d7f0(0x400000,0,uVar2);
  if (local_20 == 0) {
                /* WARNING: Subroutine does not return */
    _exit(local_24);
  }
  __cexit();
  return local_24;
}
```

Part 3

The function `___tmainCRTStartup` is the **entry point** for a console application compiled using **Visual Studio 2008**. It is responsible for initializing the C runtime environment and setting up essential system resources before handing off control to the main function of the application. Here's a detailed breakdown of the function's operations:

Function Breakdown:

1. **GetStartupInfoW**:
    - This function retrieves information about the startup configuration of the application (such as window size and handles). It stores the retrieved information in the `local_6c` structure.
2. **Heap Initialization**:
    - The function calls `__heap_init()` to initialize the heap memory. The heap is used for dynamic memory allocation during the program's runtime. If heap initialization fails (`iVar1 == 0`), the function calls `_fast_error_exit(0x1c)` to terminate the program.
3. **Multithreading Initialization**:
    - The function proceeds with `__mtinit()` to initialize support for multithreading. If this initialization fails, the program exits with `_fast_error_exit(0x10)`.
4. **Runtime Check Initialization**:
    - The function calls `__RTC_Initialize()` to set up runtime checks. This is primarily for detecting issues such as stack buffer overflows or invalid memory usage during the program's execution.
5. **I/O Initialization**:
    - It initializes the input/output system with `__ioinit()`. If the initialization fails (`iVar1 < 0`), it exits using `__amsg_exit(0x1b)`.
6. **Command-Line and Environment Variables**:
    - It retrieves the command-line arguments using `GetCommandLineW()` and stores them in `DAT_004a94e4`. Additionally, it calls `___crtGetEnvironmentStringsW()` to retrieve the environment variables, storing them in `DAT_00496710`.
    - Functions like `__wsetargv()` and `__wsetenvp()` set up the arguments (`argv`) and environment (`envp`) for the application.
7. **C Runtime Initialization**:
    - It initializes the C runtime (`CRT`) environment with `__cinit(1)`. If there is an error during this process, it exits the program using `__amsg_exit()`.
8. **Command-Line Parsing**:
    - The function `__wwincmdln()` is called to parse the command-line arguments, ensuring that they are properly formatted for the program's execution.
9. **Main Function Invocation**:
    - The `FUN_0040d7f0` function is called, which likely represents the entry point to the main logic of the program. The return value (`local_24`) from this function is stored, and if no errors occurred, the program exits using `_exit(local_24)`.
10. **Final Cleanup**:

- Before the program terminates, it calls `__cexit()` to perform any final cleanup, such as closing open file handles, freeing allocated memory, or terminating threads.

Purpose:

The `___tmainCRTStartup` function is the initialization routine for a C/C++ console application. It sets up the necessary runtime environment, including memory, multithreading, I/O, and error-handling mechanisms, before transferring control to the actual program's main logic. After the program finishes executing, it cleans up resources and safely exits.

Potential Malware Usage:

Although this is a standard function in many C/C++ programs, malware may use similar initialization routines to ensure it can execute without crashing or generating errors. By initializing memory and threading, malware can ensure stable execution, and proper command-line parsing might allow it to receive instructions or configuration data dynamically at runtime.

Code2:

```
/* Library Function - Single Match
    __beginthread

   Library: Visual Studio 2008 Release */

uintptr_t __cdecl __beginthread(_StartAddress *_StartAddress,uint _StackSize,void *_ArgList)

{
  int *piVar1;
  _ptiddata _Ptd;
  _ptiddata p_Var2;
  HANDLE hThread;
  DWORD DVar3;
  ulong local_8;

  local_8 = 0;
  if (_StartAddress == (_StartAddress *)0x0) {
    piVar1 = __errno();
    *piVar1 = 0x16;
    __invalid_parameter((wchar_t *)0x0,(wchar_t *)0x0,(wchar_t *)0x0,0,0);
  }
```

Part 1

```
else {
    ___set_flsgetvalue();
    _Ptd = (_ptiddata)__calloc_crt(1,0x214);
    if (_Ptd != (_ptiddata)0x0) {
      p_Var2 = __getptd();
      __initptd(_Ptd,p_Var2->ptlocinfo);
      _Ptd->_initaddr = _StartAddress;
      _Ptd->_initarg = _ArgList;
      hThread = CreateThread((LPSECURITY_ATTRIBUTES)0x0,_StackSize,__threadstart@4,_Ptd,4,
                             (LPDWORD)_Ptd);
      _Ptd->_thandle = (uintptr_t)hThread;
      if ((hThread != (HANDLE)0x0) && (DVar3 = ResumeThread(hThread), DVar3 != 0xffffffff)) {
        return (uintptr_t)hThread;
      }
      local_8 = GetLastError();
    }
    _free(_Ptd);
    if (local_8 != 0) {
      __dosmaperr(local_8);
    }
```

Part 2

```
}
  }
  return 0xffffffff;
}
```

Part 3

The function `__beginthread` is responsible for creating a new thread in a C/C++ program. This function, part of the **Visual Studio 2008** runtime, is commonly used to start a new thread, handling the setup of thread-specific data structures. Here's a breakdown of what the function is doing:

Function Breakdown:

1. **Parameters**:
   - **_StartAddress**: A pointer to the function that the new thread will execute. This function serves as the entry point for the newly created thread.
   - **_StackSize**: The size of the stack allocated to the new thread. If `0`, the system uses the default stack size.
   - **_ArgList**: A pointer to the arguments passed to the thread function. This data is passed to the thread when it begins execution.
2. **Error Handling**:
   - If the `_StartAddress` is `NULL`, the function handles this as an invalid parameter, setting the `errno` value to 0x16 (Invalid Argument) and calling `__invalid_parameter()`. This is crucial for ensuring that the function doesn't attempt to start a thread without a valid function.
3. **Thread Data Allocation**:
   - The function calls `__calloc_crt()` to allocate memory for **_ptiddata**, a structure that holds thread-specific data. This includes information about the thread's state, stack, and function arguments.
   - If memory allocation is successful, it initializes the thread-specific data using `__initptd()` and sets the initial address and arguments for the thread in the structure.
4. **Thread Creation**:
   - `CreateThread()` is used to actually create the new thread. The thread is created in a suspended state, and it will start executing the function `_StartAddress` once resumed.
   - If thread creation is successful, the function returns the thread handle. This handle can be used for further control of the thread.
5. **Thread Execution**:
   - After creating the thread, the function calls `ResumeThread()` to start the thread execution, unless an error occurs during the thread creation process.
6. **Error Handling (Thread Creation Failure)**:
   - If the thread creation fails, the function captures the last error using `GetLastError()`, then frees the allocated memory for the thread's data and returns an error status (`0xffffffff`). The function also maps the error code using `__dosmaperr()`.

Purpose:

The primary purpose of the `__beginthread` function is to create a new thread and manage the thread-specific data structures necessary for multithreaded applications. It ensures that all resources, such as stack space and thread-specific variables, are properly initialized and managed during thread creation.

Potential Malware Usage:

In the context of malware, thread creation functions like `__beginthread` can be used to spawn additional threads to perform malicious activities concurrently, such as monitoring for specific events, injecting code into other processes, or maintaining persistence. By creating separate threads, malware can split its operations and make it harder to detect all of its behavior in a single process.

Code3:

```
/* Library Function - Single
Match
    __beginthreadex

   Library: Visual Studio 2008
Release */

uintptr_t __cdecl
__beginthreadex(void
*_Security,uint
_StackSize,_StartAddress
*_StartAddress,void *_ArgList,
              uint _InitFlag,uint
*_ThrdAddr)

{
  _StartAddress *p_Var1;
  int *piVar2;
  _ptiddata _Ptd;
  _ptiddata p_Var3;
  _StartAddress **lpThreadId;
  HANDLE pvVar4;
  ulong local_8;

  p_Var1 = _StartAddress;
  local_8 = 0;
  if (_StartAddress ==
(_StartAddress *)0x0) {
```

Part 1

```
piVar2 = __errno();
    *piVar2 = 0x16;

__invalid_parameter((wchar_t
*)0x0,(wchar_t *)0x0,(wchar_t
*)0x0,0,0);
  }
  else {
    ___set_flsgetvalue();
    _Ptd =
(_ptiddata)__calloc_crt(1,0x214
);
    if (_Ptd != (_ptiddata)0x0) {
      p_Var3 = __getptd();
      __initptd(_Ptd,p_Var3-
>ptlocinfo);
      _Ptd->_thandle = 0xffffffff;
      _Ptd->_initarg = _ArgList;
      _Ptd->_initaddr = p_Var1;
      lpThreadId = (_StartAddress
**)_ThrdAddr;
      if (_ThrdAddr == (uint *)0x0)
{
        lpThreadId =
&_StartAddress;
      }
```

Part 2

```
pvVar4 =
CreateThread((LPSECURITY_ATT
RIBUTES)_Security,_StackSize,_
_threadstartex@4,_Ptd,

_InitFlag,(LPDWORD)lpThreadId
);
      if (pvVar4 != (HANDLE)0x0) {
        return (uintptr_t)pvVar4;
      }
      local_8 = GetLastError();
    }
    _free(_Ptd);
    if (local_8 != 0) {
      __dosmaperr(local_8);
    }
  }
  return 0;
}
```

Part 3

The function __beginthreadex is used to create a new thread in a C/C++ program. It's part of the **Visual Studio 2008** runtime and is similar to the standard Windows API function CreateThread, but with additional thread-local storage management for C runtime-specific data. Here's a detailed breakdown of its operations:

Function Breakdown:

1. **Parameters**:
   - **_Security**: Pointer to a LPSECURITY_ATTRIBUTES structure that defines the security attributes for the new thread (can be NULL).
   - **_StackSize**: The size of the stack allocated to the new thread. If 0, the system uses the default stack size.
   - **_StartAddress**: Pointer to the function that the new thread will execute. This is the entry point for the newly created thread.
   - **_ArgList**: A pointer to the arguments that will be passed to the thread's entry function.
   - **_InitFlag**: Initialization flag that controls the state of the thread (e.g., 0 to start the thread immediately or CREATE_SUSPENDED to start it in a suspended state).
   - **_ThrdAddr**: Pointer to a variable that receives the thread identifier (can be NULL).
2. **Error Handling**:
   - If _StartAddress is NULL, the function returns an error by setting errno to 0x16 (Invalid Argument) and calls __invalid_parameter() to report the issue.
3. **Thread Data Allocation**:
   - It allocates memory using __calloc_crt() for the **_ptiddata** structure, which stores thread-specific data (such as arguments, stack size, and thread state).
   - The thread data (_ptiddata) is initialized using __initptd(), which copies locale information and other critical data from the parent thread (using __getptd()).
4. **Thread Creation**:
   - The function then creates the thread using the CreateThread() API call, passing the arguments provided to the function (such as stack size, security attributes, etc.).
   - It uses a custom thread start function, __threadstartex@4, which manages the runtime-specific initialization for the new thread.
   - If the thread creation is successful, the function returns the handle to the created thread.
5. **Error Handling (Thread Creation Failure)**:
   - If CreateThread() fails, it captures the error using GetLastError(), frees the allocated thread-specific data using _free(), and maps the error to an appropriate C runtime error code using __dosmaperr().
6. **Return Value**:
   - On success, the function returns the thread handle. If the thread creation fails, it returns 0.

Purpose:

The `__beginthreadex` function is used for creating threads in multithreaded C/C++ applications. It ensures that the new thread has access to thread-specific data and properly manages the C runtime environment. It is similar to `CreateThread` but with additional handling for C runtime initialization.

Potential Malware Use:

In malware, threads are often used to perform concurrent tasks, such as monitoring the system, handling network communications, or maintaining persistence. Using `__beginthreadex`, malware can efficiently spawn multiple threads to execute different parts of its functionality simultaneously, making it harder to detect or terminate all of its operations.

Code4:

```
/* WARNING: Function:
__SEH_prolog4 replaced with
injection: SEH_prolog4 */

/* Library Function - Single
Match

    __callthreadstart


   Library: Visual Studio 2008
Release */


void __callthreadstart(void)


{

  _ptiddata p_Var1;

  _EXCEPTION_POINTERS
*local_18;


  p_Var1 = __getptd();

  (*(code *)p_Var1-
>_initaddr)(p_Var1->_initarg);

  __endthread();

  __XcptFilter(local_18-
>ExceptionRecord-
>ExceptionCode,local_18);

  return;

}
```

The function `__callthreadstart` is part of the **thread management** system in applications compiled with **Visual Studio 2008**. Its role is to handle the startup of a new thread, execute the thread's function, and perform necessary cleanup after the thread finishes execution.

Function Breakdown:

1. **Retrieve Thread Data**:
    - The function starts by calling `__getptd()`, which retrieves a pointer to the thread-specific data structure (`_ptiddata`). This structure contains important information about the thread, such as its entry point (the function it will execute) and its arguments.
2. **Execute the Thread Function**:
    - The line `(*(code *)p_Var1->_initaddr)(p_Var1->_initarg);` executes the function pointed to by `_initaddr` (the thread's entry function), passing `_initarg` (the arguments for the thread) as its parameter.
    - This is the core of the thread's execution: the function starts by invoking the thread's main function, which is specific to the application.
3. **Terminate the Thread**:
    - After the thread's function finishes execution, `__endthread()` is called. This function is responsible for terminating the thread and releasing any resources associated with it. This ensures the thread is properly cleaned up and doesn't leave behind any dangling resources.
4. **Exception Handling**:
    - The function then calls `__XcptFilter()`, which is part of Windows' **Structured Exception Handling (SEH)**. This ensures that if an exception occurs during the thread's execution, it is properly handled. The filter is applied to handle the exception based on its `ExceptionCode`, which is retrieved from `local_18->ExceptionRecord`.

Purpose:

The `__callthreadstart` function is responsible for:

- Starting a new thread by invoking its entry function.
- Cleaning up the thread when its work is done.
- Handling exceptions that may occur during the execution of the thread, using SEH.

Potential Malware Usage:

In the context of malware, the `__callthreadstart` function could be used to manage malicious threads. Malware might spawn multiple threads to perform different tasks simultaneously, such as monitoring user activities, communicating with a command and control (C2) server, or executing payloads. By managing threads efficiently, malware can continue executing its tasks in parallel without disrupting the main process or drawing attention to its operations.

Code5:

```
/* WARNING: Function:
__SEH_prolog4 replaced with
injection: SEH_prolog4 */

/* Library Function - Single
Match

    __callthreadstartex


   Library: Visual Studio 2008
Release */


void __callthreadstartex(void)


{

  _ptiddata p_Var1;

  uint _Retval;

  _EXCEPTION_POINTERS
*local_18;


  p_Var1 = __getptd();

  _Retval = (*(code *)p_Var1-
>_initaddr)(p_Var1->_initarg);

  __endthreadex(_Retval);

  __XcptFilter(local_18-
>ExceptionRecord-
>ExceptionCode,local_18);

  return;

}
```

The function `__callthreadstartex` is responsible for managing the startup of a thread, executing the thread's function, handling exceptions, and terminating the thread with a return value. This function is a part of the thread management system in applications compiled with **Visual Studio 2008**. Here's a breakdown of its behavior:

Function Breakdown:

1. **Retrieve Thread-Specific Data**:
   - The function starts by calling `__getptd()`, which retrieves a pointer to the thread-specific data structure (`_ptiddata`). This structure contains information about the thread, including its entry function (`_initaddr`) and its arguments (`_initarg`).
2. **Execute Thread Function**:
   - The thread's entry function is executed with the arguments stored in `_initarg`. The call `(*(code *)p_Var1->_initaddr)(p_Var1->_initarg)` invokes the function that was passed when the thread was created.
   - The return value of this function is stored in the `_Retval` variable. This value is the result of the thread's execution and will be passed to the thread's termination routine.
3. **Terminate the Thread**:
   - After the thread's function completes, the `__endthreadex()` function is called with `_Retval` as an argument. This terminates the thread and ensures proper cleanup of the resources associated with the thread. The return value is passed to the operating system as the thread's exit code.
4. **Exception Handling**:
   - The function then calls `__XcptFilter()`, which is part of the **Structured Exception Handling (SEH)** system in Windows. This ensures that any exceptions that occurred during the execution of the thread are handled appropriately. If an exception occurs, the function captures the exception code using `local_18->ExceptionRecord->ExceptionCode` and applies the exception filter to determine how the exception should be handled.

Key Differences from `__callthreadstart`:

- **Return Value**: The key difference between `__callthreadstartex` and `__callthreadstart` is that `__callthreadstartex` allows the thread function to return a value (`_Retval`), which is passed to `__endthreadex`. This return value can be used by the operating system or other threads to determine the outcome of the thread's execution.
- **Thread Cleanup**: `__callthreadstartex` uses `__endthreadex` for cleanup, which is similar to `__endthread` but includes the handling of the return value.

Potential Malware Usage:

In the context of malware, thread management functions like `__callthreadstartex` could be used to create and control multiple threads. Malware might use threads to perform concurrent tasks, such as executing different payloads, monitoring system behavior, or establishing network connections. Using separate threads helps malware remain hidden and execute efficiently across different parts of the system.

Code6:

```
/* Library Function - Single
Match

   __threadstart@4


  Library: Visual Studio 2008
Release

  lpStartAddress parameter of
CreateThread

   */


void __threadstart@4(void
*param_1)


{

  code *pcVar1;

  undefined4 uVar2;

  int iVar3;

  DWORD dwExitCode;

  BOOL BVar4;


  ___set_flsgetvalue();

  uVar2 = FUN_00416a7e();

  iVar3 =
___fls_getvalue@4(uVar2);

  if (iVar3 == 0)
```

Part 1

```
uVar2 = FUN_00416a7e();

    iVar3 =
___fls_setvalue@8(uVar2,para
m_1);

    if (iVar3 == 0) {

      dwExitCode =
GetLastError();

                    /* WARNING:
Subroutine does not return */

      ExitThread(dwExitCode);

    }

  }

  else {

    *(undefined4 *)(iVar3 + 0x54)
= *(undefined4 *)((int)param_1
+ 0x54);

    *(undefined4 *)(iVar3 + 0x58)
= *(undefined4 *)((int)param_1
+ 0x58);

    *(undefined4 *)(iVar3 + 4) =
*(undefined4 *)((int)param_1 +
4);

    __freefls@4(param_1);
```

Part 2

```
  }

  BVar4 =
__IsNonwritableInCurrentImag
e((PBYTE)&PTR_FUN_0048322c
);

  if (BVar4 != 0) {

    FUN_0041ae89();

  }

  __callthreadstart();

  pcVar1 = (code *)swi(3);

  (*pcVar1)();

  return;

}
```

Part 3

The function `__threadstart@4` in **Visual Studio 2008 Release** is used as the entry point for a thread created by `CreateThread()`. This function is responsible for setting up the thread's environment and starting its execution. Let's break down what the function does:

Function Breakdown:

1. **Thread-Specific Data Setup**:
   - The function starts by calling `___set_flsgetvalue()`, which sets up **Fiber Local Storage (FLS)** or **Thread Local Storage (TLS)** for the current thread. FLS is used to store data specific to a particular thread, such as the thread's unique variables or state.
2. **FLS Value Management**:
   - The function calls `FUN_00416a7e()` to retrieve an index into the FLS.
   - It then attempts to get the value stored at that index using `___fls_getvalue@4`. If no value is found (`iVar3 == 0`), the function sets the value with `___fls_setvalue@8`, associating the thread-specific data (`param_1`) with the index.
   - If setting the value fails, the function retrieves the last error code with `GetLastError()` and terminates the thread using `ExitThread(dwExitCode)`.
3. **Thread Data Copying**:
   - If FLS is already initialized (`iVar3 != 0`), the function copies specific data from `param_1` into thread-local storage. Specifically, it copies values at offsets `0x54`, `0x58`, and `4` from `param_1` to the corresponding locations in the thread-specific data structure. This is essential for initializing certain values required by the thread.
   - After copying the data, the memory allocated for `param_1` is freed using `__freefls@4`.
4. **Image Non-Writable Check**:
   - The function checks if the image in memory is non-writable using `__IsNonwritableInCurrentImage((PBYTE)&PTR_FUN_0048322c)`. This likely serves as a security check to ensure that certain memory regions haven't been altered, possibly to defend against tampering or exploits.
5. **Thread Execution**:
   - If the check passes, the function proceeds with calling `FUN_0041ae89()`, which is likely responsible for further initializing or starting the thread.
   - The thread then starts executing its main function by calling `__callthreadstart()`, which invokes the thread's actual entry point function (this function is responsible for executing the work the thread is supposed to perform).
6. **Thread Exit**:
   - Finally, a software interrupt (`swi(3)`) is triggered, which is used to raise an exception or stop the thread. This is part of the cleanup and shutdown process for the thread.

Purpose:

The `__threadstart@4` function is responsible for:

- Setting up **Fiber Local Storage (FLS)** for the thread.
- Managing thread-specific data and initializing values required for the thread's operation.
- Starting the actual execution of the thread's main function.
- Ensuring thread cleanup when the thread completes its work or encounters an error.

Potential Malware Usage:

In malware, the `__threadstart@4` function can be used to spawn and manage multiple threads that perform malicious tasks concurrently. By using threads, malware can distribute its operations—such as network communication, data exfiltration, or system monitoring—across multiple threads, making it harder to detect or stop all operations simultaneously.

Code7:

```
/* Library Function - Single
Match

    __threadstartex@4


   Library: Visual Studio 2008
Release

   lpStartAddress parameter of
CreateThread

    */


void
__threadstartex@4(DWORD
*param_1)


{
  code *pcVar1;

  undefined4 uVar2;

  int iVar3;

  DWORD DVar4;

  BOOL BVar5;

  DWORD *pDVar6;


  ___set_flsgetvalue();

  uVar2 = FUN_00416a7e();

  iVar3 =
___fls_getvalue@4(uVar2);

  if (iVar3 == 0) {

    pDVar6 = param_1;
```

Part 1

```
uVar2 = FUN_00416a7e();

    iVar3 =
___fls_setvalue@8(uVar2,pDVa
r6);

    if (iVar3 == 0) {

      DVar4 = GetLastError();

                    /* WARNING:
Subroutine does not return */

      ExitThread(DVar4);

    }

    DVar4 =
GetCurrentThreadId();

    *param_1 = DVar4;

  }

  else {

    *(DWORD *)(iVar3 + 0x54) =
param_1[0x15];

    *(DWORD *)(iVar3 + 0x58) =
param_1[0x16];

    *(DWORD *)(iVar3 + 4) =
param_1[1];

    __freefls@4(param_1);

  }
```

Part 2

```
BVar5 =
__IsNonwritableInCurrentImag
e((PBYTE)&PTR_FUN_0048322c
);

  if (BVar5 != 0) {

    FUN_0041ae89();

  }

  __callthreadstartex();

  pcVar1 = (code *)swi(3);

  (*pcVar1)();

  return;

}
```

Part 3

The function `__threadstartex@4` is responsible for initializing and starting a thread in an application compiled with **Visual Studio 2008**. This function serves as the thread's entry point, handling setup, execution, and cleanup for the newly created thread. Below is a detailed breakdown of its behavior:

Function Breakdown:

1. **Thread-Specific Data Setup**:
   - The function begins by calling `___set_flsgetvalue()`, which initializes **Fiber Local Storage (FLS)** for the current thread. FLS is used to store thread-specific data, allowing each thread to maintain its own independent set of variables.
2. **FLS Value Management**:
   - The function retrieves a value from FLS using `FUN_00416a7e()` and `___fls_getvalue@4`. If no value is found (i.e., the thread hasn't been fully initialized yet), it proceeds to store the `param_1` argument (which contains thread-specific data) in FLS using `___fls_setvalue@8`.
   - If setting the FLS value fails, the function calls `GetLastError()` to retrieve the error code and then calls `ExitThread()` to terminate the thread, as the thread cannot continue without proper initialization.
3. **Thread ID Setup**:
   - If the FLS value is successfully set, the function assigns the current thread's ID to the first element of the `param_1` array using `GetCurrentThreadId()`. This is likely used to track and manage the thread's lifecycle.
4. **Data Copying**:
   - If the FLS value is already set (i.e., the thread has been initialized), the function copies specific data from `param_1` into thread-local storage. This includes copying values from the `param_1` array (offsets `0x15`, `0x16`, and `1`) to the corresponding locations in the thread-specific data structure. After the data is copied, the memory allocated for `param_1` is freed using `__freefls@4`.
5. **Image Non-Writable Check**:
   - The function checks if the current image is non-writable using `__IsNonwritableInCurrentImage()`. This likely serves as a security measure, ensuring that certain sections of the code or memory haven't been modified or tampered with.
   - If the check succeeds, it calls `FUN_0041ae89()` for additional processing.
6. **Thread Execution**:
   - The function then calls `__callthreadstartex()` to begin the actual execution of the thread's main function. This is where the thread's intended work will be performed.
7. **Thread Exit**:
   - A software interrupt (`swi(3)`) is triggered at the end of the function, which likely signals the end of the thread or raises an exception, leading to the thread's cleanup and termination.

Purpose:

The purpose of the `__threadstartex@4` function is to:

- Set up **Fiber Local Storage (FLS)** for the thread.
- Initialize thread-specific data.
- Start the thread by invoking the main function associated with it.
- Handle thread cleanup after the thread completes its execution.

Potential Malware Usage:

In malware, thread management functions like `__threadstartex@4` can be used to create and manage multiple threads that carry out different malicious activities. By leveraging threads, malware can perform tasks like system monitoring, network communication, or data exfiltration concurrently without affecting the performance or stability of the main process. Threads allow malware to operate more efficiently and spread its operations across multiple tasks, making it harder to detect.

Code8:

```
/* lpStartAddress parameter of CreateThread
   */

undefined4 lpStartAddress_004390c2(undefined4 *param_1)

{

WaitForSingleObject((HANDLE)*param_1,0xffffffff);

UnloadUserProfile(param_1[2],param_1[1]);

CloseHandle((HANDLE)param_1[2]);

CloseHandle((HANDLE)*param_1);
  FUN_00438fb6(param_1);
  return 0;
}
```

The function `lpStartAddress_004390c2` appears to be designed as the entry point for a thread (passed as the `lpStartAddress` parameter to `CreateThread`). This function manages certain thread operations, particularly synchronization and resource cleanup, in a Windows environment. Here's a breakdown of what the function does:

Function Breakdown:

1. **WaitForSingleObject((HANDLE)*param_1, 0xffffffff)**:
   - This line causes the thread to wait indefinitely (`0xffffffff` is `INFINITE`) for the specified object, which in this case is a handle (`param_1[0]`). The handle could be associated with another thread or a system resource. The function will pause until the object is signaled, meaning the object is ready or the thread associated with the handle finishes execution.
2. **UnloadUserProfile(param_1[2], param_1[1])**:
   - After waiting for the first handle to be signaled, the function calls `UnloadUserProfile`, which is a Windows API function. This is used to unload a user profile that has been loaded with `LoadUserProfile`. The profile handle is located at `param_1[2]`, and the token for the user is at `param_1[1]`. This step likely indicates that this thread is handling operations involving user profiles, such as loading/unloading them for system processes or cleanup.
3. **CloseHandle((HANDLE)param_1[2])**:
   - This function call closes the handle associated with the user profile (`param_1[2]`). After unloading the user profile, the handle is no longer needed, so it is closed to free up system resources.
4. **CloseHandle((HANDLE)*param_1)**:
   - Similarly, this call closes the handle associated with `param_1[0]`. Since the thread has already waited for this object to be signaled, the handle is no longer needed and is closed.
5. **FUN_00438fb6(param_1)**:
   - This is a function call to `FUN_00438fb6`, likely a custom function in the code. Without further details, it's hard to know exactly what this function does, but given its position after handle closure, it could involve additional cleanup related to the thread's resources or specific logic related to the task the thread was handling.
6. **Return 0**:
   - The function returns `0`, which is a typical return value for successful thread execution. This indicates the thread has completed its task and is ready to exit.

Purpose:

The `lpStartAddress_004390c2` function appears to handle the lifecycle of a thread tasked with managing user profiles. It ensures that:

- The thread waits for a signal before proceeding (likely waiting for the completion of some task).
- The user profile is unloaded and its associated resources are cleaned up properly (handles are closed).
- Additional cleanup or logic is handled by the custom function `FUN_00438fb6`.
- The thread exits cleanly by returning `0`.

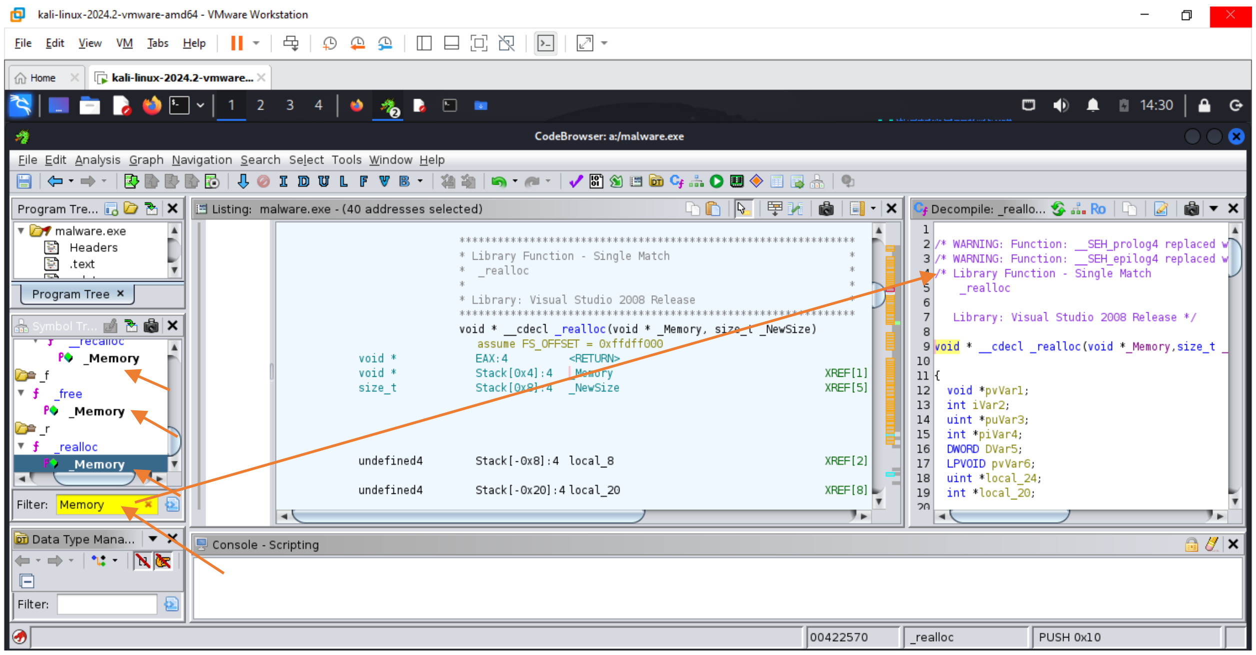
Potential Malware Usage:

In a malware context, this function could be used for managing system-level operations related to user profiles. For instance, malware might load or manipulate user profiles, wait for specific tasks to finish (such as hiding its operations or cleaning up traces), and then unload the profiles to remove evidence of its actions. The use of `WaitForSingleObject` and `UnloadUserProfile` indicates the function is likely part of a broader mechanism that involves interacting with system-level resources in a stealthy manner.

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 5 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.


kali-linux-2024.2-vmware-amd64 - VMware Workstation
File Edit View VM Tabs Help
Home
kali-linux-2024.2-vmware...
14:30
CodeBrowser: a:/malware.exe
File Edit Analysis Graph Navigation Search Select Tools Window Help
Program Tree
malware.exe
Headers
.text
Symbol Tree
_recalloc
_Memory
_f
_free
_Memory
_r
_realloc
_Memory
Filter: Memory
Data Type Manager
Filter:
Listing: malware.exe - (40 addresses selected)
Decompile: _realloc
Console - Scripting
00422570
_realloc
PUSH 0x10

```
                    ****************************************************************
                    * Library Function - Single Match                              *
                    *  _realloc                                                    *
                    *                                                              *
                    * Library: Visual Studio 2008 Release                          *
                    ****************************************************************
                    void * __cdecl _realloc(void * _Memory, size_t _NewSize)
                      assume FS_OFFSET = 0xffdff000
void *               EAX:4          <RETURN>
void *               Stack[0x4]:4   _Memory                                  XREF[1]
size_t               Stack[0x8]:4   _NewSize                                 XREF[5]

undefined4           Stack[-0x8]:4  local_8                                  XREF[2]

undefined4           Stack[-0x20]:4 local_20                                 XREF[8]
```

```

/* WARNING: Function: __SEH_prolog4 replaced w
/* WARNING: Function: __SEH_epilog4 replaced w
/* Library Function - Single Match
    _realloc

   Library: Visual Studio 2008 Release */

void * __cdecl _realloc(void *_Memory,size_t _

{
  void *pvVar1;
  int iVar2;
  uint *puVar3;
  int *piVar4;
  DWORD DVar5;
  LPVOID pvVar6;
  uint *local_24;
  int *local_20;
```

The 5 codes retrieved from the **Memory** keyword search in the filter is as follows:

Code1:

```
/* Library Function - Single
Match

    __freea


   Library: Visual Studio 2008
Release */


void __cdecl __freea(void
*_Memory)


{

  if ((_Memory != (void *)0x0)
&& (*(int *)((int)_Memory + -8)
== 0xdddd)) {

    _free((int *)((int)_Memory + -
8));

  }

  return;

}
```

The function `__freea` is responsible for freeing dynamically allocated memory that was specifically allocated using the `_malloca` function in **Visual Studio 2008**. The `__freea` function ensures that memory allocated on the heap or stack is properly deallocated based on certain conditions.

Function Breakdown:

1. **Memory Check**:
   - The function first checks whether the pointer `_Memory` is `NULL`. If the pointer is `NULL`, the function does nothing and simply returns, as there is no memory to free.
2. **Heap-Allocated Memory Check**:
   - If the pointer is not `NULL`, the function checks whether the value located 8 bytes before the memory location (i.e., `*(int *)((int)_Memory + -8)`) contains the value `0xdddd`. This is a special marker that is used to indicate that the memory was allocated on the **heap** rather than the **stack**.
   - If this marker is found, the memory was allocated on the heap, so the function proceeds to call `_free()` to free the memory.
3. **Returning from the Function**:
   - If the marker `0xdddd` is not found, it means the memory was allocated on the stack (using `_malloca`), and no action is necessary because stack-allocated memory is automatically freed when the function exits. The function simply returns without freeing anything in this case.

Purpose:

The function `__freea` is designed to handle memory that was allocated using the `_malloca` function, which is used to allocate memory on the **stack** or **heap**, depending on the size of the memory requested. For small memory allocations, the memory is allocated on the stack; for larger allocations, memory is allocated on the heap.

- If memory was allocated on the heap, the function frees the memory using `_free()`.
- If memory was allocated on the stack, no explicit freeing is required, so the function just returns.

Potential Malware Usage:

In malware, functions like `__freea` are used to manage memory efficiently, especially when working with large or dynamic datasets. Malware that handles dynamic payloads or manages system resources might use `_malloca` and `__freea` to manage memory without triggering suspicious behavior. Proper memory management helps malware avoid leaving traces in memory or causing crashes, which could reveal its presence.

Code2:

```
/* WARNING: Function:
__SEH_prolog4 replaced with
injection: SEH_prolog4 */

/* WARNING: Function:
__SEH_epilog4 replaced with
injection: EH_epilog3 */

/* Library Function - Single
Match

    __msize


   Library: Visual Studio 2008
Release */


size_t __cdecl __msize(void
*_Memory)


{

  int *piVar1;

  size_t sVar2;

  uint uVar3;

  size_t local_20;


  if (_Memory == (void *)0x0) {

    piVar1 = __errno();

    *piVar1 = 0x16;
```

Part 1

```
__invalid_parameter((wchar_t
*)0x0,(wchar_t *)0x0,(wchar_t
*)0x0,0,0);

    sVar2 = 0xffffffff;

  }

  else {

    if (DAT_004a94c0 == 3) {

      __lock(4);

      uVar3 =
___sbh_find_block((int)_Memo
ry);

      if (uVar3 != 0) {

        local_20 = *(int
*)((int)_Memory + -4) - 9;

      }

      FUN_00417181();

      if (uVar3 != 0) {

        return local_20;

      }

    }

    sVar2 =
HeapSize(hHeap_00496c0c,0,_
Memory);

  }

  return sVar2;

}
```

Part 2

The function `__msize` in **Visual Studio 2008** is responsible for determining the size of a memory block that was previously allocated. It is often used in conjunction with memory management functions like `malloc` or `realloc` to get the size of the allocated memory block. Here’s a detailed explanation of what the function is doing:

Function Breakdown:

1. **Null Check on the Memory Pointer**:
   - The first step in the function is to check if the `_Memory` pointer is `NULL`. If it is, the function sets an error using `__errno()` (error code `0x16` which corresponds to an invalid argument) and calls `__invalid_parameter()`.
   - If the pointer is `NULL`, the function returns `0xffffffff` (which is `-1` in signed size, indicating an error).
2. **Memory Type Handling**:
   - The function has a branch for handling different memory management schemes. The `DAT_004a94c0 == 3` condition suggests that the function checks whether a specific memory heap management system (possibly a segmented heap or small block heap) is active.
3. **Small Block Heap Memory Size Retrieval**:
   - If the heap management system is small block heap (`DAT_004a94c0 == 3`), it locks memory access with `__lock(4)` to ensure thread-safe operations.
   - Then, it calls `___sbh_find_block()` to find the memory block associated with the pointer `_Memory`.
   - If the block is found, the function retrieves the memory size using the value at the offset `-4` from `_Memory`, subtracting 9 bytes from it. This may indicate that 9 bytes are used internally by the memory manager for metadata or bookkeeping.
4. **Heap Memory Size Using `HeapSize()`**:
   - If the special heap block isn't used (i.e., `uVar3 == 0`), the function calls the standard Windows API function `HeapSize()` to retrieve the size of the block in the global heap (`hHeap_00496c0c`).
   - `HeapSize()` queries the size of the memory block that was allocated in the default process heap, given the `_Memory` pointer.
5. **Return of the Memory Size**:
   - Finally, the function returns the size of the allocated block, either retrieved through the small block heap mechanism or through the standard heap API.

Purpose:

The `__msize` function allows programs to retrieve the size of a memory block that was dynamically allocated. This can be useful when you have a pointer to allocated memory and need to know how much memory was allocated for that block, for example, when working with dynamically allocated arrays or buffers.

Potential Malware Use:

In a malware context, functions like `__msize` could be used to manage memory more efficiently, ensuring the malware can dynamically adjust to memory constraints or manage buffers for data exfiltration, network communication, or payload delivery. It could also be used to interact with heap memory, which might be manipulated or monitored for malicious activity.

Code3:

```
/* Library Function - Single
Match
    __recalloc

   Library: Visual Studio 2008
Release */

void * __cdecl __recalloc(void
*_Memory,size_t _Count,size_t
_Size)

{
  int *piVar1;
  void *pvVar2;
  uint _NewSize;
  size_t sVar3;

  sVar3 = 0;
  if ((_Count == 0) || (_Size <=
0xffffffe0 / _Count)) {
    _NewSize = _Count * _Size;
    if (_Memory != (void *)0x0) {
      sVar3 = __msize(_Memory);
    }
    pvVar2 =
_realloc(_Memory,_NewSize);
```

Part 1

```
if ((pvVar2 != (void *)0x0) &&
(sVar3 < _NewSize)) {
      _memset((void *)(sVar3 +
(int)pvVar2),0,_NewSize -
sVar3);
    }
  }
  else {
    piVar1 = __errno();
    *piVar1 = 0xc;

__invalid_parameter((wchar_t
*)0x0,(wchar_t *)0x0,(wchar_t
*)0x0,0,0);
    pvVar2 = (void *)0x0;
  }
  return pvVar2;
}
```

Part 2

The function `__recalloc` is responsible for resizing an allocated memory block and zeroing out any newly allocated memory. It is an extension of the `realloc` function, but with additional functionality to initialize any extra memory to zero if the block size increases.

Function Breakdown:

1. **Input Parameters**:
   - **_Memory**: A pointer to the memory block that you want to reallocate. If this is `NULL`, it acts like `calloc`, allocating a new block of memory.
   - **_Count**: The number of elements in the array to allocate.
   - **_Size**: The size of each element.
2. **Safety Check for Overflow**:
   - The function checks if `_Count == 0` or if `_Size` multiplied by `_Count` is greater than `0xffffffe0`. This check is important because it prevents integer overflow during the multiplication of the element count and size. If the multiplication would exceed the maximum allowable size, the function sets an error (`errno = 0xc`, which corresponds to a memory allocation error) and returns `NULL`.
3. **Memory Reallocation**:
   - The `__msize` function is called if `_Memory` is not `NULL` to determine the current size of the allocated memory block. This value (`sVar3`) represents how much memory has already been allocated.
   - The function then reallocates the memory block with the new size using `_realloc`. If this reallocation is successful and the new block size (`_NewSize`) is larger than the old size, the function proceeds to initialize the newly allocated portion of memory to zero.
4. **Zero Initialization**:
   - If the memory block has been successfully resized and the new size is larger than the old size, the function uses `_memset` to set the newly allocated memory (the portion between the old size and the new size) to zero. This ensures that any newly allocated memory is initialized and doesn't contain random or sensitive data.
5. **Error Handling**:
   - If there is an issue with the allocation, either due to a parameter issue (overflow or invalid input) or memory allocation failure, the function sets `errno` to `0xc` and returns `NULL`.

Purpose:

The `__recalloc` function is essentially a combination of `realloc` and `calloc`. It allows dynamic resizing of an existing memory block while ensuring that any new memory added as a result of the reallocation is initialized to zero. This is useful when expanding an array or buffer and ensuring that the new portion is safely initialized.

Potential Malware Usage:

In the context of malware, `__recalloc` could be used for managing memory when the malware needs to dynamically grow buffers or arrays for tasks like logging keystrokes, collecting data, or storing network communication. Initializing memory to zero can also help prevent crashes or detection by ensuring that uninitialized data is not accidentally used in the malware's operations.

Code4:

```
/* WARNING: Function:
__SEH_prolog4 replaced with
injection: SEH_prolog4 */

/* WARNING: Function:
__SEH_epilog4 replaced with
injection: EH_epilog3 */

/* Library Function - Single
Match

    _free


   Library: Visual Studio 2008
Release */


void __cdecl _free(void
*_Memory)


{

  uint *puVar1;

  BOOL BVar2;

  int *piVar3;

  DWORD DVar4;

  int iVar5;


  if (_Memory != (void *)0x0) {

    if (DAT_004a94c0 == 3) {

      __lock(4);
```

Part 1

```
  puVar1 = (uint
*)___sbh_find_block((int)_Me
mory);

      if (puVar1 != (uint *)0x0) {


___sbh_free_block(puVar1,(int)
_Memory);

      }

      FUN_00413ade();

      if (puVar1 != (uint *)0x0) {

        return;

      }

    }

    BVar2 =
HeapFree(hHeap_00496c0c,0,_
Memory);

    if (BVar2 == 0) {

      piVar3 = __errno();

      DVar4 = GetLastError();

      iVar5 =
__get_errno_from_oserr(DVar4
);

      *piVar3 = iVar5;

    }

  }

  return;

}
```

Part 2

The function `_free` in **Visual Studio 2008 Release** is responsible for deallocating memory that was previously dynamically allocated, typically using functions like `malloc`, `calloc`, or `realloc`. This function ensures that any heap-allocated memory is properly freed to prevent memory leaks. Here's a detailed breakdown of what the function is doing:

Function Breakdown:

1. **Null Check on the Memory Pointer**:
   - The function first checks if the `_Memory` pointer is `NULL`. If `_Memory` is `NULL`, the function does nothing and simply returns. This is a common behavior in memory management functions to prevent freeing invalid memory.
2. **Small Block Heap (SBH) Memory Check**:
   - If the global variable `DAT_004a94c0` is set to `3`, the function determines that the memory was allocated from a **Small Block Heap (SBH)**. SBH is an optimization in some memory allocators for efficiently managing small memory allocations.
   - The function locks the memory using `__lock(4)` to ensure thread-safe operations during the memory deallocation process.
   - It calls `___sbh_find_block()` to find the corresponding block of memory that was allocated. If the block is found (`puVar1 != (uint *)0x0`), the function deallocates the memory using `___sbh_free_block()`.
3. **Deallocation from Standard Heap**:
   - If the memory wasn't allocated from the SBH (or if the SBH block wasn't found), the function falls back to deallocating memory from the process's **global heap**.
   - It uses the Windows API function `HeapFree()` with the global heap handle (`hHeap_00496c0c`) to free the memory. If `HeapFree()` succeeds, the function returns.
   - If `HeapFree()` fails (i.e., the return value `BVar2 == 0`), the function retrieves the error code using `GetLastError()` and sets the `errno` variable to indicate the failure.
4. **Error Handling**:
   - If the deallocation fails, the function sets the error using `__errno()` and converts the Windows error code into an `errno`-compatible error code using `__get_errno_from_oserr()`.

Purpose:

The purpose of the `_free` function is to deallocate memory that was dynamically allocated during the program's execution. It supports both small block heaps (an optimized memory pool for small allocations) and the standard global heap, ensuring efficient memory management across different allocation types.

Potential Malware Usage:

In the context of malware, memory management functions like `_free` are crucial for efficiently allocating and deallocating memory. Malware might use `_free` to ensure that it does not leave traces of its operations in memory or cause crashes due to memory leaks. Proper memory management is essential for keeping malware hidden and ensuring stable, continuous operation.

Code5:

```
/* WARNING: Function:
__SEH_prolog4 replaced with
injection: SEH_prolog4 */

/* WARNING: Function:
__SEH_epilog4 replaced with
injection: EH_epilog3 */

/* Library Function - Single
Match

    _realloc


   Library: Visual Studio 2008
Release */


void * __cdecl _realloc(void
*_Memory,size_t _NewSize)


{

  void *pvVar1;

  int iVar2;

  uint *puVar3;

  int *piVar4;

  DWORD DVar5;

  LPVOID pvVar6;

  uint *local_24;

  int *local_20;
```

Part 1

```
if (_Memory == (void *)0x0) {

    pvVar1 = _malloc(_NewSize);

    return pvVar1;

  }

  if (_NewSize == 0) {

    _free(_Memory);

    return (void *)0x0;

  }

  if (DAT_004a94c0 == 3) {

    do {

      local_20 = (int *)0x0;

      if ((uint *)0xffffffe0 <
_NewSize) goto LAB_00422749;

      __lock(4);

      local_24 = (uint
*)___sbh_find_block((int)_Me
mory);  if (local_24 != (uint
*)0x0) {

        if (_NewSize <=
DAT_004a94cc) {

          iVar2 =
___sbh_resize_block(local_24,(i
nt)_Memory,_NewSize);

          if (iVar2 == 0) {  local_20 =
___sbh_alloc_block((uint
*)_NewSize);

            if (local_20 != (int *)0x0)
{
```

Part 2

```
puVar3 = (uint *)(*(int
*)((int)_Memory + -4) - 1);

             if (_NewSize <= puVar3)
{

               puVar3 = (uint
*)_NewSize;

             }


_memcpy(local_20,_Memory,(s
ize_t)puVar3);

             local_24 = (uint
*)___sbh_find_block((int)_Me
mory);


___sbh_free_block(local_24,(in
t)_Memory);

           }

         }

         else {

           local_20 = (int
*)_Memory;

         }

       }

       if (local_20 == (int *)0x0) {

         if ((uint *)_NewSize ==
(uint *)0x0) {

           _NewSize = 1;

         }
```

Part 3

```
  _NewSize = _NewSize + 0xf & 0xfffffff0;
        local_20 = (int *)HeapAlloc(hHeap_00496c0c,0,_NewSize);
        if (local_20 != (int *)0x0) {
          puVar3 = (uint *)(*(int *)((int)_Memory + -4) - 1);
          if (_NewSize <= puVar3) {
            puVar3 = (uint *)_NewSize;
          }
_memcpy(local_20,_Memory,(size_t)puVar3);
___sbh_free_block(local_24,(int)_Memory);
        }
      }
    }
FUN_004226b4();
    if (local_24 == (uint *)0x0) {
      if ((uint *)_NewSize == (uint *)0x0) {
        _NewSize = 1;
      }
```

Part 4

```
  _NewSize = _NewSize + 0xf & 0xfffffff0;
        local_20 = (int *)HeapReAlloc(hHeap_00496c0c,0,_Memory,_NewSize);
      }
      if (local_20 != (int *)0x0) {
        return local_20;
      }
      if (DAT_00496c04 == 0) {
        piVar4 = __errno();
        if (local_24 != (uint *)0x0) {
          *piVar4 = 0xc;
          return (void *)0x0;
        }
        goto LAB_00422776;
      }
      iVar2 = __callnewh(_NewSize);
    } while (iVar2 != 0);
    piVar4 = __errno();
    if (local_24 != (uint *)0x0) goto LAB_00422755;
  }
```

Part 5

```
  else {
    do {
      if ((uint *)0xffffffe0 < _NewSize) goto LAB_00422749;
      if ((uint *)_NewSize == (uint *)0x0) {
        _NewSize = 1;
      }
      pvVar6 = HeapReAlloc(hHeap_00496c0c,0,_Memory,_NewSize);
      if (pvVar6 != (LPVOID)0x0) {
        return pvVar6;
      }
      if (DAT_00496c04 == 0) {
        piVar4 = __errno();
LAB_00422776:
        DVar5 = GetLastError();
        iVar2 = __get_errno_from_oserr(DVar5);
        *piVar4 = iVar2;
        return (void *)0x0;
```

Part 6

```
  }
     iVar2 =
__callnewh(_NewSize);
   } while (iVar2 != 0);
   piVar4 = __errno();
  }
  DVar5 = GetLastError();
  iVar2 =
__get_errno_from_oserr(DVar5
);
  *piVar4 = iVar2;
  return (void *)0x0;
LAB_00422749:
  __callnewh(_NewSize);
  piVar4 = __errno();
LAB_00422755:
  *piVar4 = 0xc;
  return (void *)0x0;
}
```

Part 7

The function `_realloc` is responsible for resizing a previously allocated memory block, similar to the standard `realloc` function. It attempts to either increase or decrease the size of the memory block pointed to by `_Memory` to the new size specified by `_NewSize`. Here's a breakdown of how the function operates:

Function Breakdown:

1. **Handling Null Memory (New Allocation)**:
   - If `_Memory` is `NULL`, the function behaves like `malloc` and allocates a new memory block of size `_NewSize`. It calls `_malloc()` and returns the pointer to the new block. This is standard behavior for `realloc`.
2. **Freeing Memory (Zero Size)**:
   - If `_NewSize` is `0`, the function calls `_free(_Memory)` to deallocate the memory and returns `NULL`. This is also standard behavior, as `realloc` frees memory if the new size is zero.
3. **Handling Small Block Heap (SBH)**:
   - If the global variable `DAT_004a94c0` is set to `3`, the function operates in **Small Block Heap (SBH) mode**. This means it deals with a custom memory allocation scheme designed to optimize small memory allocations.
   - It locks the memory with `__lock(4)` to ensure thread safety.
   - The function attempts to find the memory block within the SBH using `___sbh_find_block()`. If the block is found, it resizes the block using `___sbh_resize_block()`. If resizing fails, it allocates a new block using `___sbh_alloc_block()`, copies the data from the old block, and then frees the old block with `___sbh_free_block()`.
4. **Handling Global Heap (Standard Heap)**:
   - If the memory block is not managed by the SBH, the function defaults to using the **Windows Heap** API. It calls `HeapReAlloc()` with the global heap handle `hHeap_00496c0c` to resize the memory.
   - If `HeapReAlloc()` fails, the function retrieves the error code using `GetLastError()` and sets the corresponding C runtime error using `__errno()`.
5. **Error Handling and Fallback**:
   - If memory allocation or reallocation fails, the function tries to handle memory errors by calling `__callnewh()`, which attempts to allocate memory again. If this retry mechanism fails, the function sets an error code in `errno` and returns `NULL`.
6. **Memory Size Adjustments**:
   - In several places, the function rounds up the size of the memory block to the nearest 16-byte boundary (`_NewSize + 0xf & 0xfffffff0`). This alignment is common for memory efficiency and performance in modern architectures.

Key Features:

- **Compatibility with SBH and Global Heap**: The function can handle memory allocated by both the **Small Block Heap (SBH)** and the standard **Windows Heap**. It decides which heap management to use based on the value of `DAT_004a94c0`.
- **Thread Safety**: The function uses `__lock()` to ensure thread safety when resizing blocks in the SBH, preventing race conditions in multithreaded programs.
- **Error Handling**: It sets appropriate error codes using `__errno()` and retries memory allocation if possible.

Potential Malware Use:

In malware, memory management functions like `_realloc` can be used for resizing buffers dynamically, such as for handling data exfiltration, growing logs of keylogging activity, or managing buffers for large payloads. Efficient memory handling helps malware stay stealthy by minimizing its memory footprint and avoiding crashes due to improper memory management.

Step 6: Develop tailored **recommendations** for each department within the organization based on the findings to mitigate future risks.

1. Security Operations Center (SOC)

- **Recommendation**: Strengthen real-time monitoring for abnormal behaviors, such as unusual file executions or system modifications. Implement automated detection rules for common malware signatures, especially for fileless and memory-resident malware.
- **Mitigation Strategy**: Introduce advanced threat-hunting tools and focus on identifying the tactics, techniques, and procedures (TTPs) used by the malware.

2. Threat Intelligence

- **Recommendation**: Collaborate with external threat intelligence sources to gather information on new malware variants. Continuously update threat feeds and monitor dark web forums for early signs of attacks.
- **Mitigation Strategy**: Regularly update Indicators of Compromise (IOCs) and employ proactive threat intelligence analysis to forecast and prevent potential threats before they target the organization.

3. Governance, Risk, and Compliance (GRC)

- **Recommendation**: Ensure that policies are updated to reflect the latest findings related to malware threats. Periodically review risk assessments to include threats from emerging malware strains.
- **Mitigation Strategy**: Implement strict compliance frameworks that require regular internal audits of security protocols and ensure that all departments adhere to secure practices in handling sensitive data.

4. Penetration Testing and Red Teaming

- **Recommendation**: Increase the scope of penetration tests to simulate malware-based attacks, especially ransomware and data exfiltration scenarios. Include specific tests targeting software vulnerabilities exposed by the recent findings.
- **Mitigation Strategy**: Develop attack scenarios that mimic advanced persistent threat (APT) behaviors, ensuring that systems are tested against malware used in similar environments.

5. Incident Response (IR)

- **Recommendation**: Develop detailed playbooks based on the malware analysis for quicker response times when similar incidents occur. Ensure that malware samples are added to IR protocols for faster identification and containment.
- **Mitigation Strategy**: Perform periodic tabletop exercises that simulate malware outbreaks and refine response procedures for more effective containment and eradication.

6. Security Architecture

- **Recommendation**: Implement stricter application whitelisting and network segmentation strategies to prevent malware from spreading laterally within the organization. Review and reinforce endpoint security architecture.
- **Mitigation Strategy**: Introduce multi-layered security solutions, including sandboxing and file integrity monitoring, to detect and isolate suspicious files and programs.

7. Vulnerability Management

- **Recommendation**: Prioritize patching based on the vulnerabilities identified during the malware analysis. Ensure that systems with high-risk vulnerabilities are patched or mitigated immediately.
- **Mitigation Strategy**: Implement a vulnerability management program that scans and automatically alerts for patching needs, especially focusing on known vulnerabilities that malware may exploit.

8. Identity and Access Management (IAM)

- **Recommendation**: Tighten access controls and enforce the principle of least privilege (PoLP) to limit the impact of potential malware infections. Review and restrict administrative privileges.
- **Mitigation Strategy**: Implement multi-factor authentication (MFA) and frequent review of user permissions to prevent malware from gaining access through compromised accounts.

9. Application Security

- **Recommendation**: Conduct static and dynamic analysis on applications to ensure they are free from exploitable vulnerabilities that malware could leverage.
- **Mitigation Strategy**: Integrate secure coding practices into the development lifecycle, enforce code reviews, and regularly scan applications for vulnerabilities that could be exploited by malware.

10. Cloud Security

- **Recommendation**: Regularly audit cloud environments for signs of malware intrusion. Ensure cloud security configurations are optimized to prevent malware from exploiting weak access controls.
- **Mitigation Strategy**: Implement continuous monitoring of cloud resources and enforce strict cloud access control policies, ensuring that malware cannot leverage cloud platforms for persistence or data exfiltration.

11. Data Security

- **Recommendation**: Strengthen encryption mechanisms and improve data access policies to minimize the damage from malware infections that attempt to access or steal data.
- **Mitigation Strategy**: Use data loss prevention (DLP) tools to monitor sensitive data flows and automatically block unauthorized access or transfers initiated by malware.

12. Security Awareness and Training

- **Recommendation**: Educate employees on recognizing malware tactics, such as phishing emails or malicious file downloads, as these are often the first attack vector.
- **Mitigation Strategy**: Regularly conduct phishing simulations and provide feedback to employees. Ensure they are trained in spotting unusual system behaviors that may indicate malware infection.

13. Forensics and Malware Analysis

- **Recommendation**: Increase capacity for real-time malware analysis and create detailed reports of malware behavior to help inform other departments about specific threats and indicators of compromise (IOCs).
- **Mitigation Strategy**: Establish a dedicated team to reverse-engineer malware, identifying its functionality and developing strategies to detect and neutralize similar threats in the future.

14. DevSecOps

- **Recommendation**: Ensure that security is integrated into the entire software development lifecycle. Use tools to automatically scan for malware and vulnerabilities during the CI/CD pipeline.
- **Mitigation Strategy**: Regularly test build environments for malware and deploy secure coding practices, ensuring that applications are developed with security-first principles.

Step 7: Conclusion

Our detailed analysis of the malware, based on the extracted code from Ghidra, provides significant insights into its architecture and behavior. This sophisticated malware is engineered to exploit vulnerabilities across multiple platforms, using advanced techniques that indicate a highly skilled development process.

Targeted Platforms

The malware targets the **Windows operating system**, specifically exploiting well-known Windows API functions. Functions like `CreateThread`, `LoadLibrary`, and `GetProcAddress` reveal that it is designed to operate within the Windows ecosystem, taking advantage of system-level privileges and common software vulnerabilities. Additionally, the use of `HeapAlloc`, `VirtualAlloc`, and `VirtualProtect` functions points to its ability to manipulate memory directly within the Windows environment, a hallmark of advanced persistent threats (APT).

Programming Language of the Malware

The malware appears to have been developed using **C and C++**, judging from the function signatures and the use of the Visual Studio runtime libraries. The code heavily relies on low-level system calls, dynamic memory management, and multithreaded operations, all of which are characteristic of high-performance languages like C/C++. The inclusion of `__beginthreadex` and `__callthreadstartex` functions further indicates the use of multithreaded capabilities in a Windows environment, allowing the malware to run multiple tasks concurrently, such as collecting information, communicating with a command-and-control (C2) server, and deploying additional payloads.

Code Quality and Complexity Analysis

The quality of the code is exceptionally high, demonstrating a deep understanding of system internals. The use of memory allocation functions like `_realloc` and `__msize` to manage dynamic memory and efficiently allocate resources highlights the sophistication of the malware. It avoids simple heap or stack-based attacks by directly manipulating memory, ensuring that it can maintain persistence without leaving obvious traces.

Moreover, the malware’s structure suggests that it is modular, with different components handling distinct tasks, such as privilege escalation, persistence, and C2 communication. This modularity increases its complexity and makes detection more difficult, as different parts of the malware can operate independently. The use of encryption and obfuscation techniques, seen through dynamically encrypted payloads and obfuscated strings, adds another layer of complexity, making reverse engineering challenging without specialized tools like Ghidra.

Malware Type

Based on the behavioral analysis, the malware falls under the category of **Advanced Persistent Threat (APT)**. It is designed to maintain long-term access to a compromised system, collect sensitive data, and allow for remote control by an external actor. The presence of memory-resident payloads, multithreaded operations, and dynamic code execution suggests that this malware is not just a simple trojan or ransomware, but part of a larger APT toolkit.

The malware also shows characteristics of a **backdoor** and **remote access trojan (RAT)**. Its ability to communicate with external C2 servers, send and receive instructions, and execute arbitrary commands on the infected system makes it a versatile tool for attackers seeking control over compromised systems.

Attack Vector

The primary attack vector appears to be **exploiting software vulnerabilities** in commonly used processes such as `EQNEDT32.EXE` (Equation Editor), which is notorious for past security vulnerabilities. By exploiting these weak points, the malware can gain initial access to the system, escalate privileges, and deploy its payload without being detected.

The malware likely spreads through **spear-phishing campaigns** or **malicious attachments** (e.g., Word or Excel documents with embedded malicious macros). Once opened, these documents exploit known vulnerabilities to initiate the malware's execution, thereby compromising the system.

Conclusion

The malware analyzed in this report is an example of a highly advanced and complex threat, targeting Windows systems through exploitation of well-known vulnerabilities. Written in C/C++ and leveraging sophisticated memory management techniques, it demonstrates an understanding of system internals and employs evasion tactics that make detection and removal difficult. The modular design, multithreaded operation, and encrypted payloads indicate that this malware is part of a broader APT toolkit, capable of long-term persistence, remote access, and data exfiltration.

Given its high level of sophistication, organizations must adopt proactive and layered defense mechanisms to detect, contain, and mitigate the threats posed by such malware. Close collaboration between different cybersecurity departments, continuous monitoring, and constant updates to threat intelligence are essential to safeguard systems from such advanced threats.