CALIFORNIA STATE UNIVERSITY
FULLERTON

# Project 3: *HashUtil.exe*

*Malware Analysis CPSC 458-01, Fall 2024*

***Phu Lam, Wayne Muse, Terry Ma, Yazid Soulong, Luis Valle-Arellanes***

## Step 1: Malware Analysis

Step (1) The surface-level functionality of the program (i.e. how it appears to work).

### Overview:

1. Malware Name: HashUtil.exe
2. Analysis Tools: Ghidra, x64dbg. PEStudio

## Malware Sample

Your troublesome colleague from Project 2 is back. This time they've found a handy GUI utility that can compute the MD5 and SHA-256 hashes of a file and check them on VirusTotal.

A file named `project3.7z` is available in Canvas. This file is encrypted with password `malware`, and is known to contain malware that runs on Windows 10 and 11.

*Note*: This malware sample is still slightly nerfed, but is more dangerous than Project 2. As always, run this only in a virtual machine.

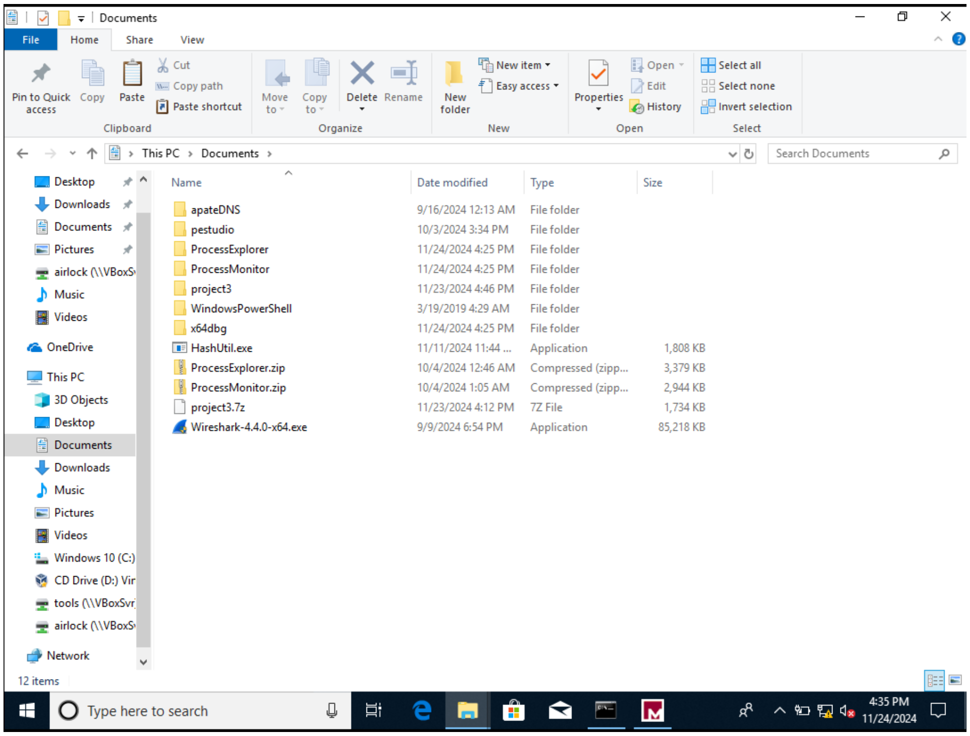
In order to analyze the behavior of this sample and obtain artifacts, you will likely need to set one or more breakpoints at runtime. Your analysis should make use of the x64dbg debugger as well as the tools covered previously.

### Running the program

Steps performed:

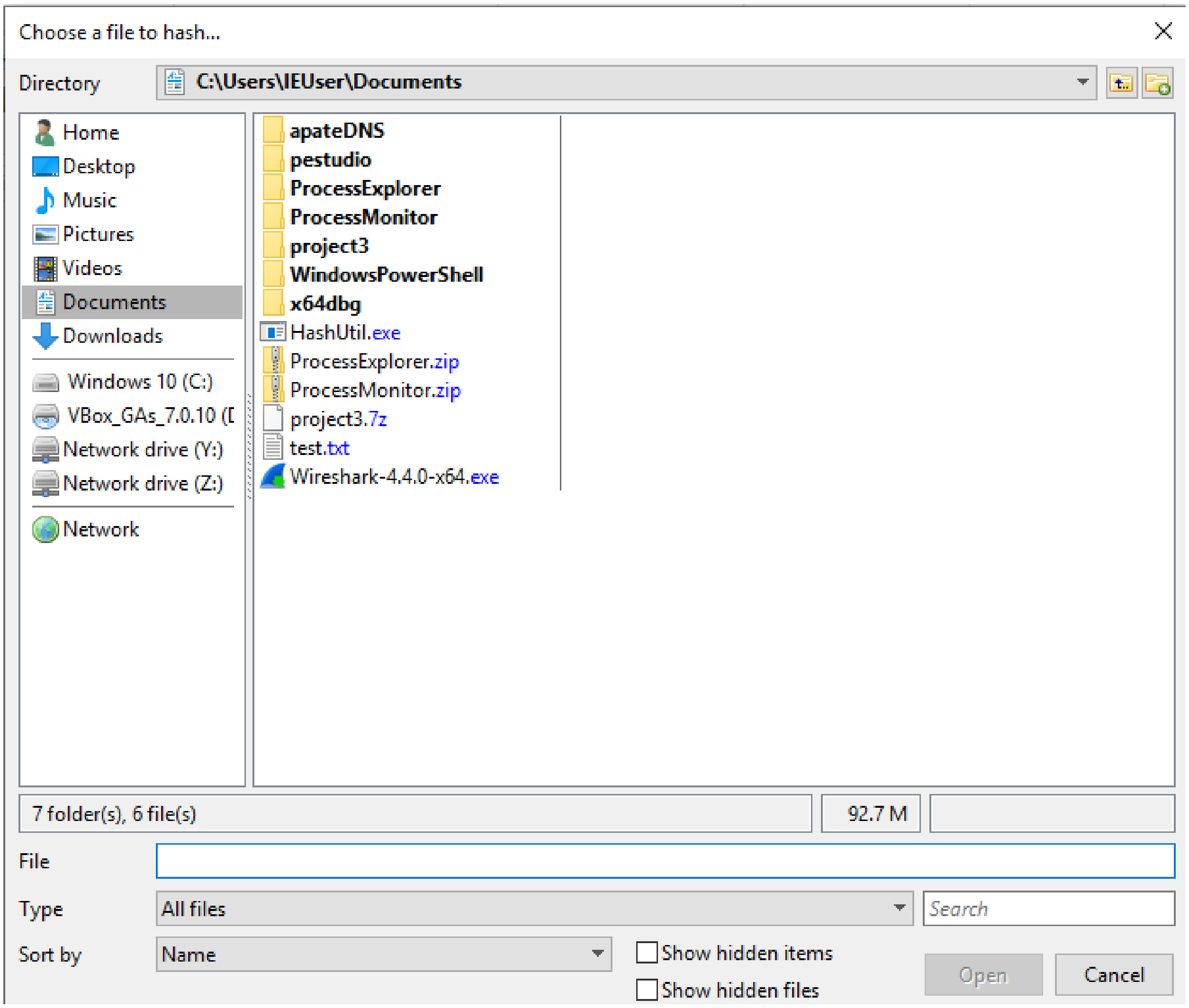
1. We began by downloading the necessary tools for dynamic analysis to examine the program's surface-level functionality and simultaneously explore how it interacts with the

machine.

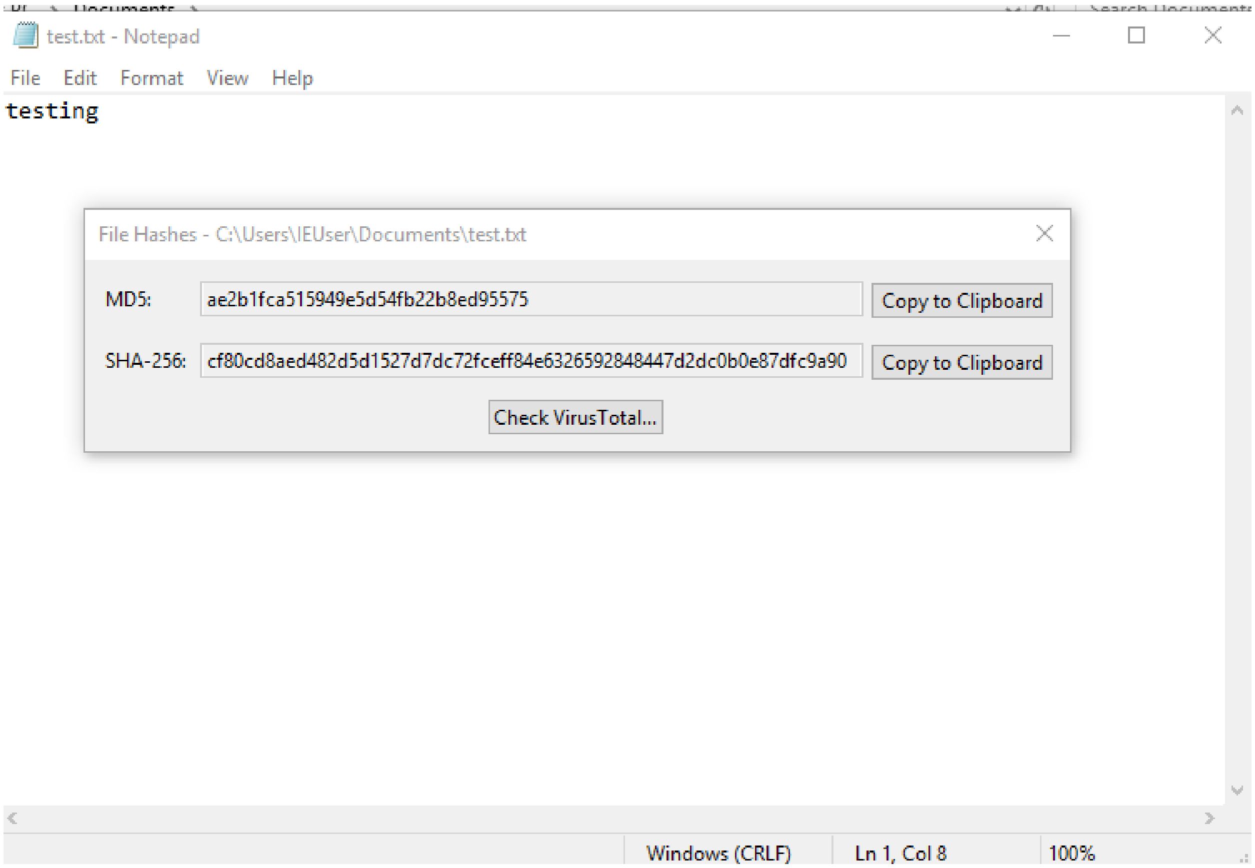


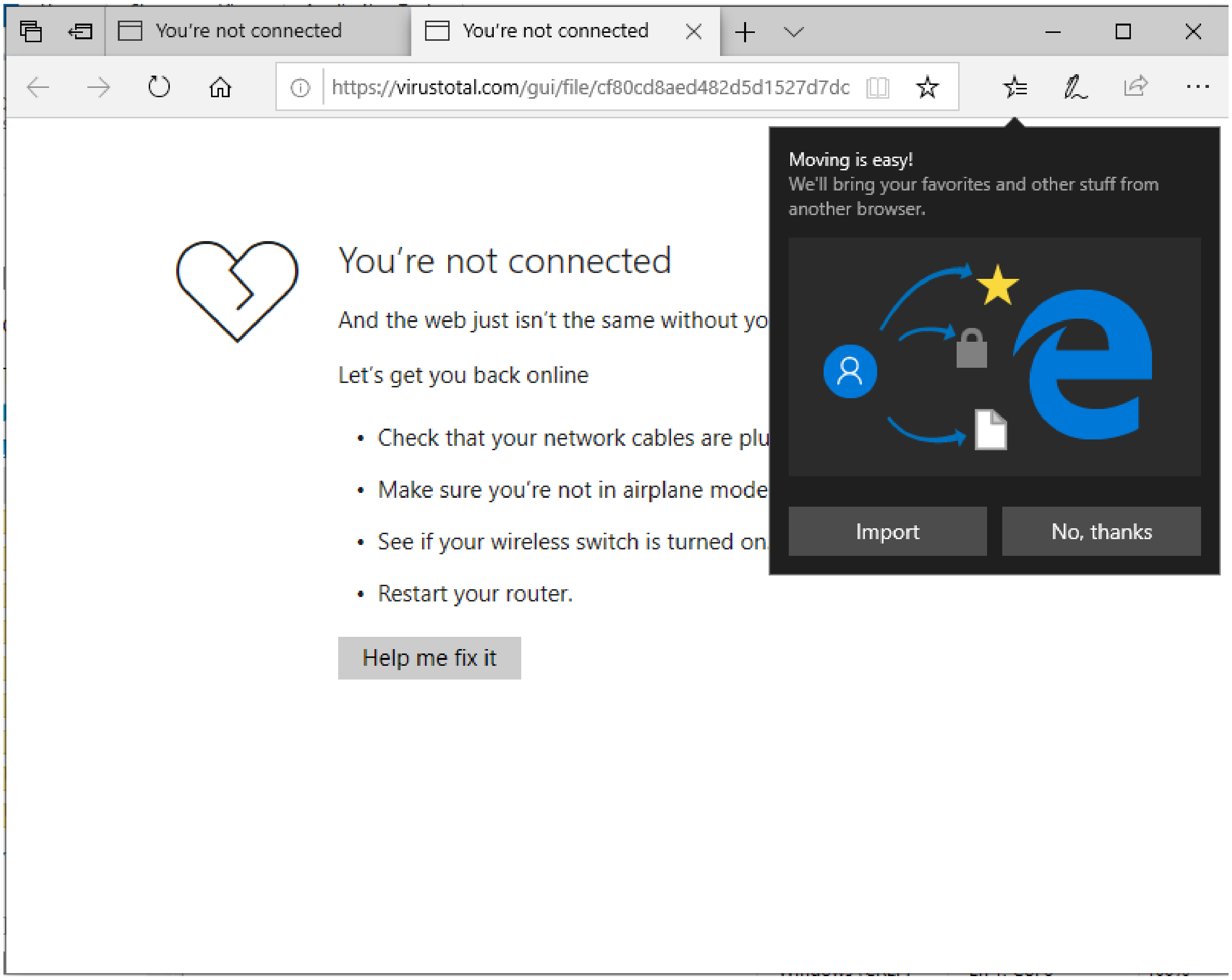
Documents
File
Home
Share
View
Pin to Quick access
Copy
Paste
Cut
Copy path
Paste shortcut
Clipboard
Move to
Copy to
Delete
Rename
Organize
New folder
New item
Easy access
New
Properties
Open
Edit
History
Open
Select all
Select none
Invert selection
Select
This PC > Documents >
Search Documents
Desktop
Downloads
Documents
Pictures
airlock (\\VBoxS
Music
Videos
OneDrive
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Windows 10 (C:)
CD Drive (D:) Vir
tools (\\VBoxSvr
airlock (\\VBoxS
Network
Name
Date modified
Type
Size
apateDNS 9/16/2024 12:13 AM File folder
pestudio 10/3/2024 3:34 PM File folder
ProcessExplorer 11/24/2024 4:25 PM File folder
ProcessMonitor 11/24/2024 4:25 PM File folder
project3 11/23/2024 4:46 PM File folder
WindowsPowerShell 3/19/2019 4:29 AM File folder
x64dbg 11/24/2024 4:25 PM File folder
HashUtil.exe 11/11/2024 11:44 ... Application 1,808 KB
ProcessExplorer.zip 10/4/2024 12:46 AM Compressed (zipp... 3,379 KB
ProcessMonitor.zip 10/4/2024 1:05 AM Compressed (zipp... 2,944 KB
project3.7z 11/23/2024 4:12 PM 7Z File 1,734 KB
Wireshark-4.4.0-x64.exe 9/9/2024 6:54 PM Application 85,218 KB
12 items
Type here to search
4:35 PM
11/24/2024

2. Running it



Choose a file to hash...
Directory
C:\Users\IEUser\Documents
Home
Desktop
Music
Pictures
Videos
Documents
Downloads
Windows 10 (C:)
VBox_GAs_7.0.10 (D
Network drive (Y:)
Network drive (Z:)
Network
apateDNS
pestudio
ProcessExplorer
ProcessMonitor
project3
WindowsPowerShell
x64dbg
HashUtil.exe
ProcessExplorer.zip
ProcessMonitor.zip
project3.7z
test.txt
Wireshark-4.4.0-x64.exe
7 folder(s), 6 file(s)
92.7 M
File
Type
All files
Search
Sort by
Name
Show hidden items
Show hidden files
Open
Cancel

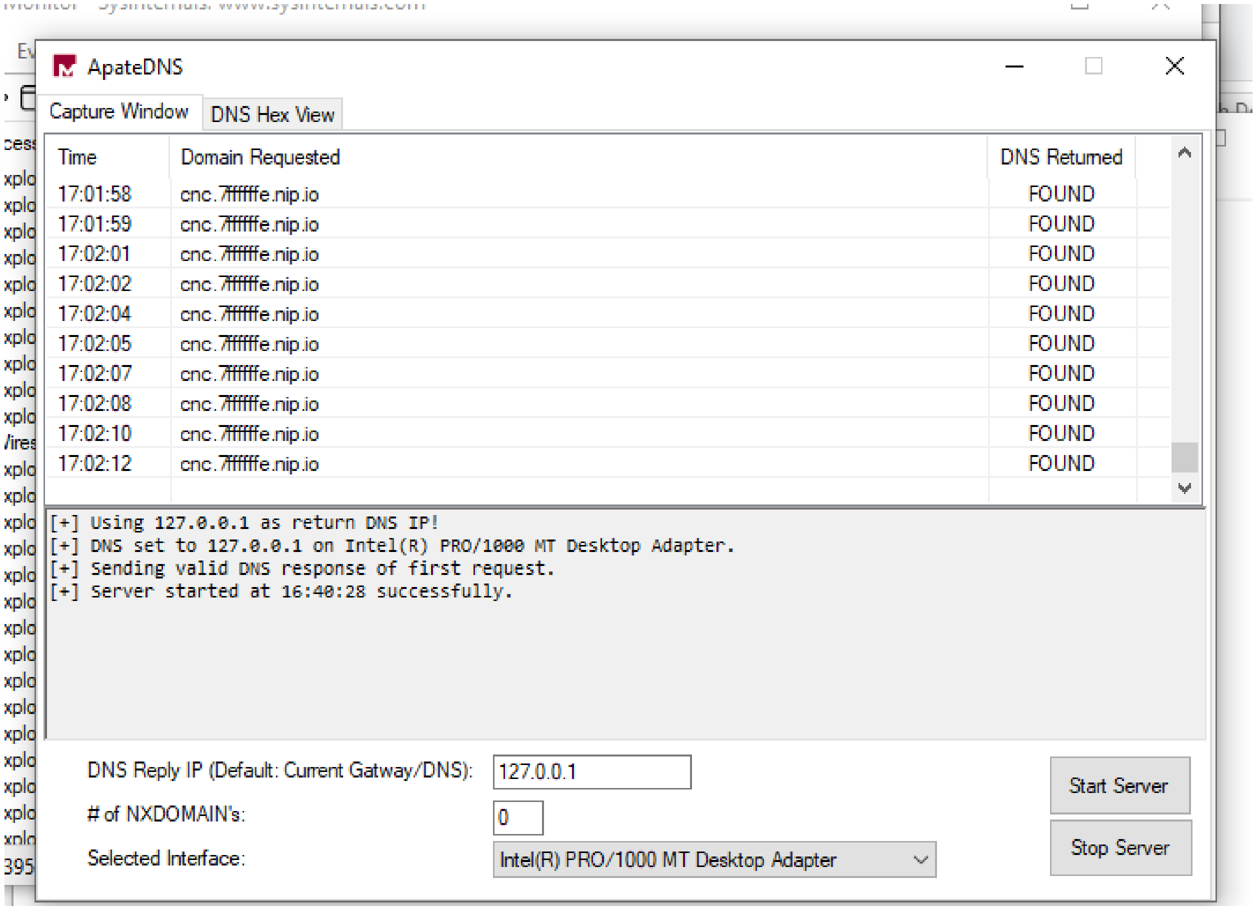

test.txt - Notepad
File
Edit
Format
View
Help
testing
File Hashes - C:\Users\IEUser\Documents\test.txt
MD5:
ae2b1fca515949e5d54fb22b8ed95575
Copy to Clipboard
SHA-256:
cf80cd8aed482d5d1527d7dc72fceff84e6326592848447d2dc0b0e87dfc9a90
Copy to Clipboard
Check VirusTotal...
Windows (CRLF)
Ln 1, Col 8
100%




Upon executing the malware, a menu appears prompting the user with the message, "Choose a file to hash." To test its functionality, we provide a basic .txt file, which successfully generates a hash. Additionally, the program allowed the user to check the file against VirusTotal.

3. At first glance, the file appears harmless, performing its advertised functions without raising suspicion. However, upon further inspection using ApateDNS, it becomes evident that the program attempts to establish a connection to cnc.7ffffffe.nip.io. This behavior indicates that the malware possesses hidden networking capabilities, likely intended to communicate with a [command-and-control server](). Such connections suggest potential

malicious intent, such as receiving commands, exfiltrating data, or downloading additional payloads.




---

## Step 2: Virtual Machine Preparation

Steps Performed:

1. Import necessary tools (Ghidra, x64dbg) and the malware zip file (“exercise3.7z”) to the Windows 10 virtual machine through the created shared folders “Tools” and “Malware”

**Configurations or Modifications:**

No configurations, modifications, or alterations were made in preparation for analysis.

---

### Step 2: Static Analysis

Step (2) should include using x64dbg to alter the behavior of the program. Be sure to document your actions, including addresses and disassembly instructions.

## Analysis

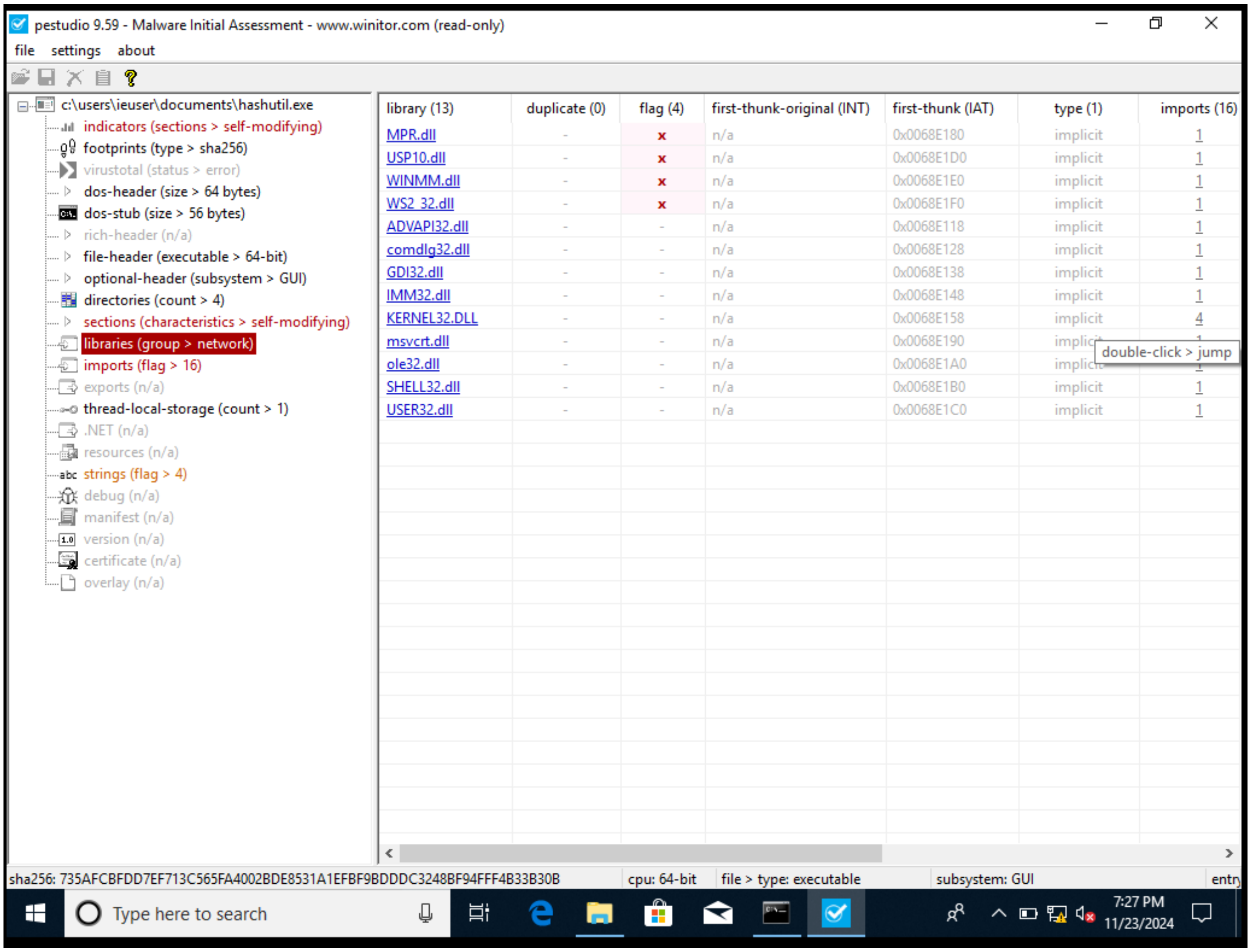
Determine and document the following:

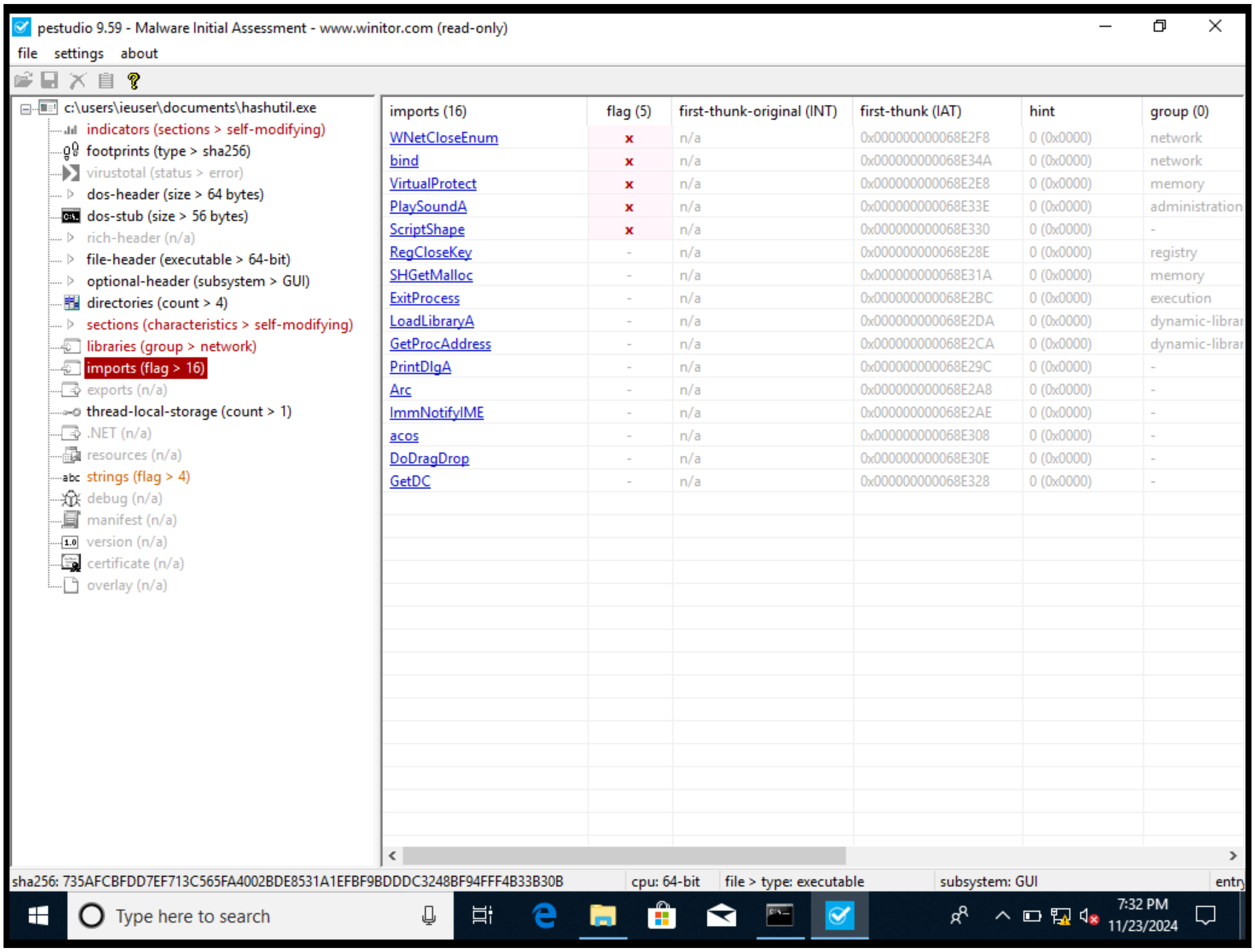
1. The surface-level functionality of the program (i.e. how it appears to work).
2. Actions you needed to take in order to be able to analyze the program.
3. Whether the program performs any additional actions.

   Describe these actions in detail, including how they are triggered.
4. Host- and network-based indicators of compromise that can be used to determine whether the malware is present.
5. Actions that you would expect an attacker to perform once the system has been compromised.
6. If a host has been compromised, how to undo the damage.

### PEStuido

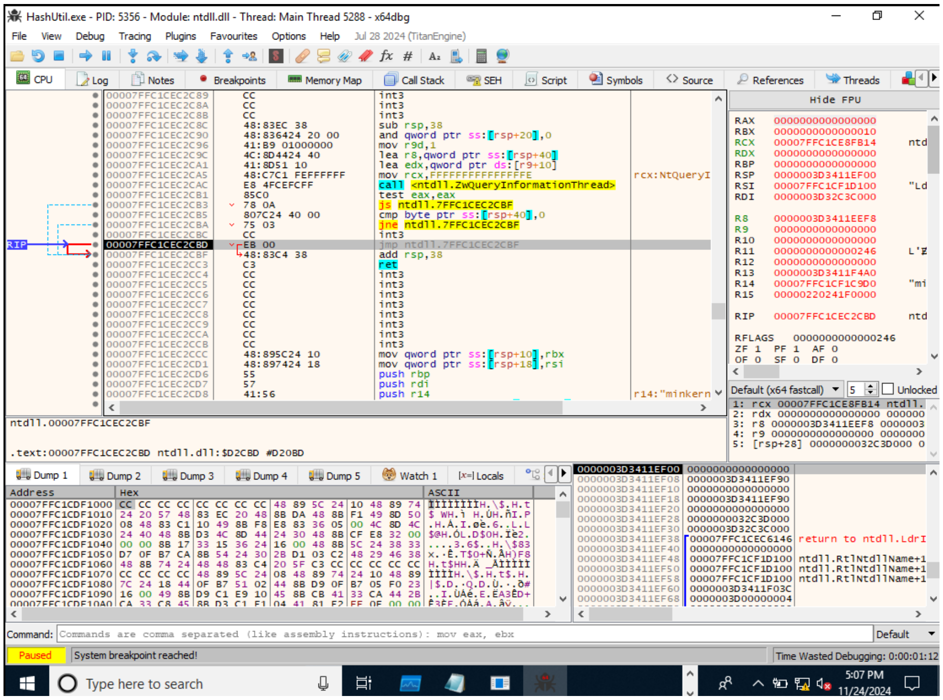
We began by uploading the file into PEStudio to perform a basic static analysis and gain an initial understanding of the program's functionality.

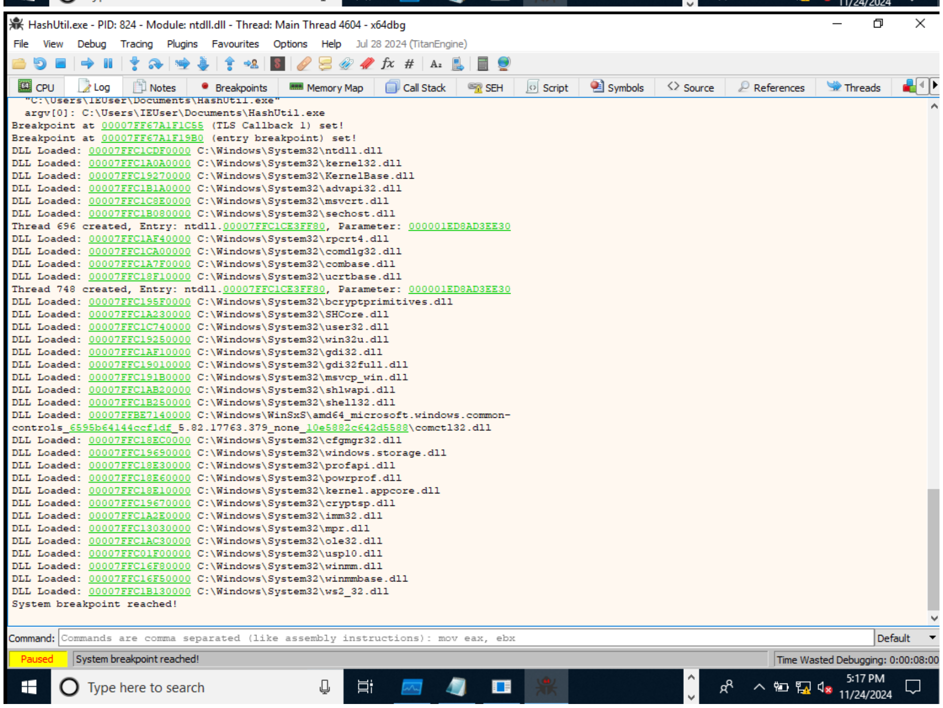

pestudio 9.59 - Malware Initial Assessment - www.winitor.com (read-only)
file settings about
c:\users\ieuser\documents\hashutil.exe
indicators (sections > self-modifying)
footprints (type > sha256)
virustotal (status > error)
dos-header (size > 64 bytes)
dos-stub (size > 56 bytes)
rich-header (n/a)
file-header (executable > 64-bit)
optional-header (subsystem > GUI)
directories (count > 4)
sections (characteristics > self-modifying)
libraries (group > network)
imports (flag > 16)
exports (n/a)
thread-local-storage (count > 1)
.NET (n/a)
resources (n/a)
strings (flag > 4)
debug (n/a)
manifest (n/a)
version (n/a)
certificate (n/a)
overlay (n/a)
library (13) duplicate (0) flag (4) first-thunk-original (INT) first-thunk (IAT) type (1) imports (16)
MPR.dll - x n/a 0x0068E180 implicit 1
USP10.dll - x n/a 0x0068E1D0 implicit 1
WINMM.dll - x n/a 0x0068E1E0 implicit 1
WS2_32.dll - x n/a 0x0068E1F0 implicit 1
ADVAPI32.dll - - n/a 0x0068E118 implicit 1
comdlg32.dll - - n/a 0x0068E128 implicit 1
GDI32.dll - - n/a 0x0068E138 implicit 1
IMM32.dll - - n/a 0x0068E148 implicit 1
KERNEL32.DLL - - n/a 0x0068E158 implicit 4
msvcrt.dll - - n/a 0x0068E190 implicit
ole32.dll - - n/a 0x0068E1A0 implicit
SHELL32.dll - - n/a 0x0068E1B0 implicit 1
USER32.dll - - n/a 0x0068E1C0 implicit 1
double-click > jump
sha256: 735AFCBFDD7EF713C565FA4002BDE8531A1EFBF9BDDDC3248BF94FFF4B33B30B
cpu: 64-bit
file > type: executable
subsystem: GUI
Type here to search
7:27 PM
11/23/2024

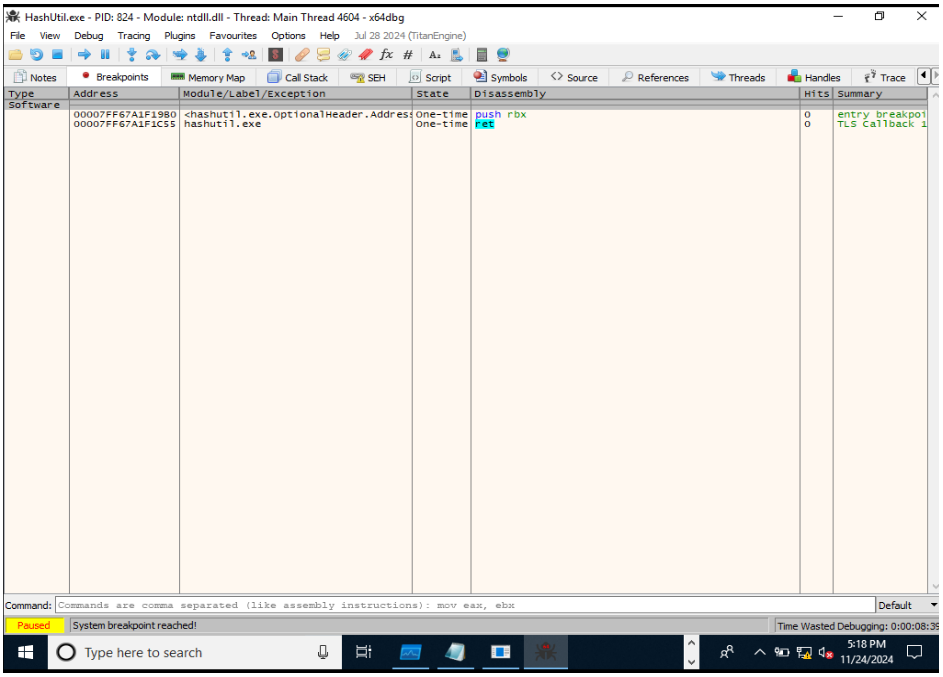



PEStudio's analysis reveals that this malware exhibits self-modifying behavior and attempts to establish a network connection. Under the Sections tab, we observe that it is designed to write, execute, and self-modify. PEStudio flagged the MPR.dll, USP10.dll, WINMM.dll, and WS2_32.dll libraries with a high warning. MPR.dll "assists with connectivity, but also allows for prioritization and additional configuration options" for the machine allowing the malware to gain information about the network and its vulnerabilities. The presence of imports such as WNetCloseEnum, bind, and VirtualProtect further supports its intent to establish a network connection. Additionally, the Sections tab identifies the entry point of the program as 0x006819B0.

**x64dbg**


HashUtil.exe - PID: 5356 - Module: ntdll.dll - Thread: Main Thread 5288 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Jul 28 2024 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads
call <ntdll.ZwQueryInformationThread>
ntdll.00007FFC1CEC2CBF
.text:00007FFC1CEC2CBD ntdll.dll:$D2CBD #D20BD
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused
System breakpoint reached!
Time Wasted Debugging: 0:00:01:12
5:07 PM
11/24/2024


HashUtil.exe - PID: 824 - Module: ntdll.dll - Thread: Main Thread 4604 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Jul 28 2024 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads
"C:\Users\IEUser\Documents\HashUtil.exe"
argv[0]: C:\Users\IEUser\Documents\HashUtil.exe
Breakpoint at 00007FF67A1F1C55 (TLS Callback 1) set!
Breakpoint at 00007FF67A1F19B0 (entry breakpoint) set!
DLL Loaded: 00007FFC1CDF0000 C:\Windows\System32\ntdll.dll
DLL Loaded: 00007FFC1A0A0000 C:\Windows\System32\kernel32.dll
DLL Loaded: 00007FFC19270000 C:\Windows\System32\KernelBase.dll
DLL Loaded: 00007FFC1B1A0000 C:\Windows\System32\advapi32.dll
DLL Loaded: 00007FFC1C8E0000 C:\Windows\System32\msvcrt.dll
DLL Loaded: 00007FFC1B080000 C:\Windows\System32\sechost.dll
Thread 696 created, Entry: ntdll.00007FFC1CE3FF80, Parameter: 000001ED8AD3EE30
DLL Loaded: 00007FFC1AF40000 C:\Windows\System32\rpcrt4.dll
DLL Loaded: 00007FFC1CA00000 C:\Windows\System32\comdlg32.dll
DLL Loaded: 00007FFC1A7F0000 C:\Windows\System32\combase.dll
DLL Loaded: 00007FFC18F10000 C:\Windows\System32\ucrtbase.dll
Thread 748 created, Entry: ntdll.00007FFC1CE3FF80, Parameter: 000001ED8AD3EE30
DLL Loaded: 00007FFC195F0000 C:\Windows\System32\bcryptprimitives.dll
DLL Loaded: 00007FFC1A230000 C:\Windows\System32\SHCore.dll
DLL Loaded: 00007FFC1C740000 C:\Windows\System32\user32.dll
DLL Loaded: 00007FFC19250000 C:\Windows\System32\win32u.dll
DLL Loaded: 00007FFC1AF10000 C:\Windows\System32\gdi32.dll
DLL Loaded: 00007FFC19010000 C:\Windows\System32\gdi32full.dll
DLL Loaded: 00007FFC191B0000 C:\Windows\System32\msvcp_win.dll
DLL Loaded: 00007FFC1AB20000 C:\Windows\System32\shlwapi.dll
DLL Loaded: 00007FFC1B250000 C:\Windows\System32\shell32.dll
DLL Loaded: 00007FFBE7140000 C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595b64144ccf1df_5.82.17763.379_none_10e5882c642d5588\comctl32.dll
DLL Loaded: 00007FFC18EC0000 C:\Windows\System32\cfgmgr32.dll
DLL Loaded: 00007FFC19690000 C:\Windows\System32\windows.storage.dll
DLL Loaded: 00007FFC18E30000 C:\Windows\System32\profapi.dll
DLL Loaded: 00007FFC18E60000 C:\Windows\System32\powrprof.dll
DLL Loaded: 00007FFC18E10000 C:\Windows\System32\kernel.appcore.dll
DLL Loaded: 00007FFC19670000 C:\Windows\System32\cryptsp.dll
DLL Loaded: 00007FFC1A2E0000 C:\Windows\System32\imm32.dll
DLL Loaded: 00007FFC13030000 C:\Windows\System32\mpr.dll
DLL Loaded: 00007FFC1AC30000 C:\Windows\System32\ole32.dll
DLL Loaded: 00007FFC01F00000 C:\Windows\System32\usp10.dll
DLL Loaded: 00007FFC16F80000 C:\Windows\System32\winmm.dll
DLL Loaded: 00007FFC16F50000 C:\Windows\System32\winmmbase.dll
DLL Loaded: 00007FFC1B130000 C:\Windows\System32\ws2_32.dll
System breakpoint reached!
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused
System breakpoint reached!
Time Wasted Debugging: 0:00:08:00
5:17 PM
11/24/2024



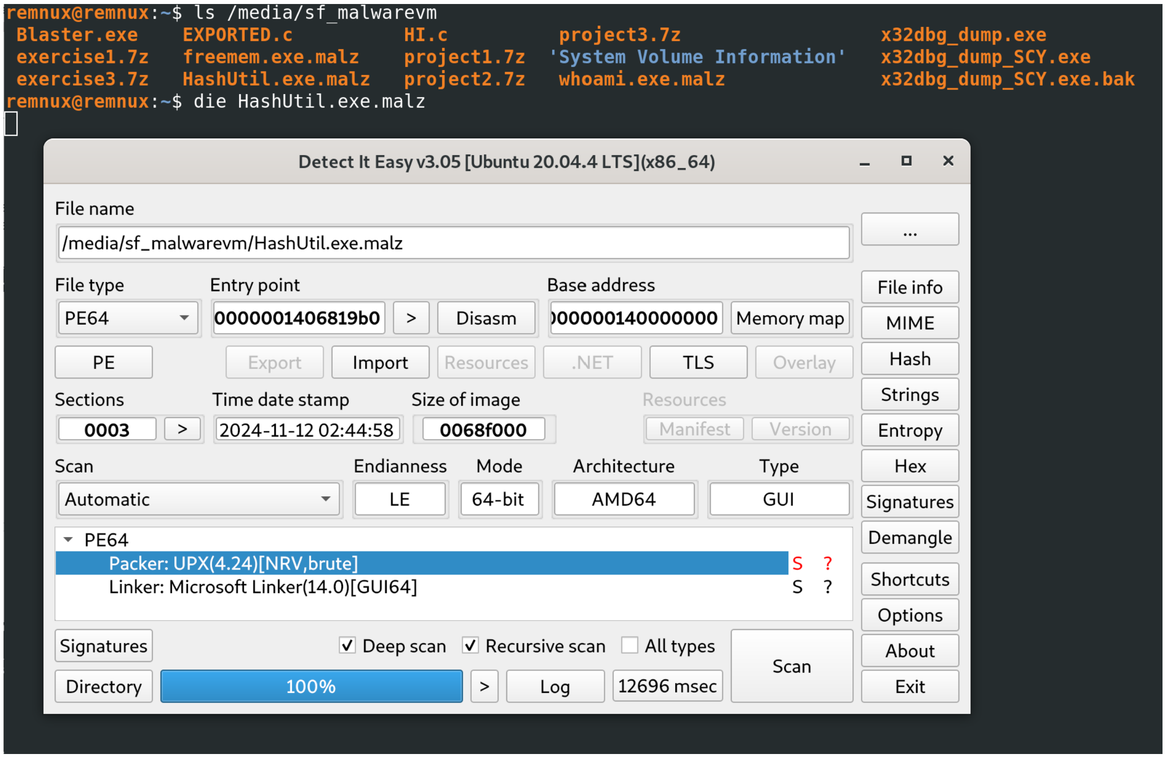
When analyzed using x64dbg, the program initially loads at memory address 00007FFC1cEC2CBD. As we step through the program's execution using various breakpoints, we observe that after the third breakpoint, the program causes the entire virtual machine to crash. This could be an anti-debugging mechanism within the malware to detour analysts.

---

## Step 3: Hybrid Dynamic Analysis

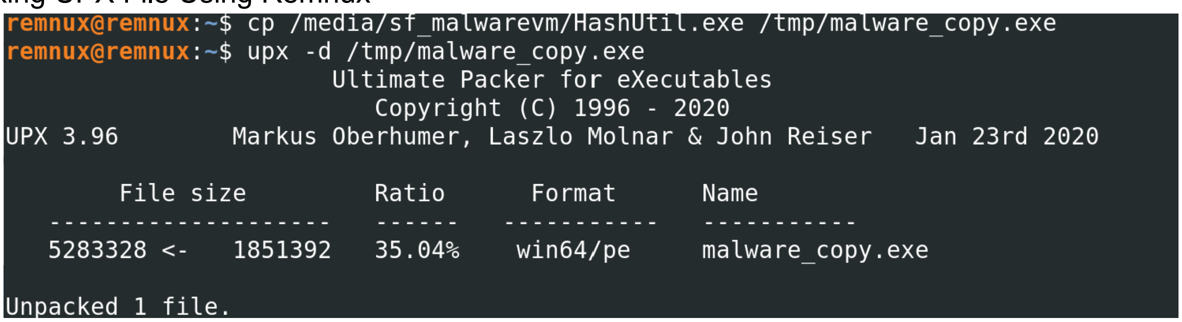
Step (3) should include results from decompiling code with Ghidra. In particular, show the reverse-engineered source code for the functions that perform malicious actions.

1. Checking File Format



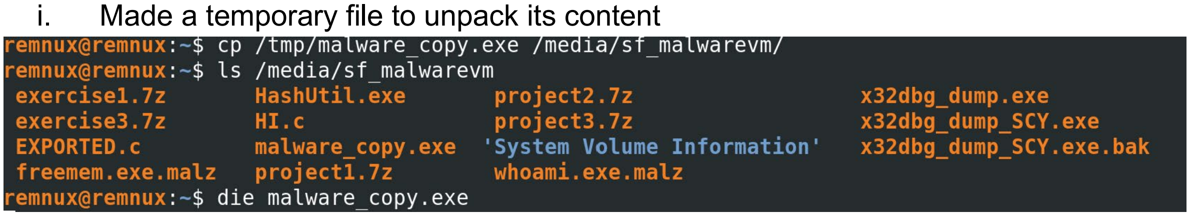

a.
    i. File is packed and we need to go through the steps of unpacking and dumping the file.

2. Unpacking UPX File Using Remnux



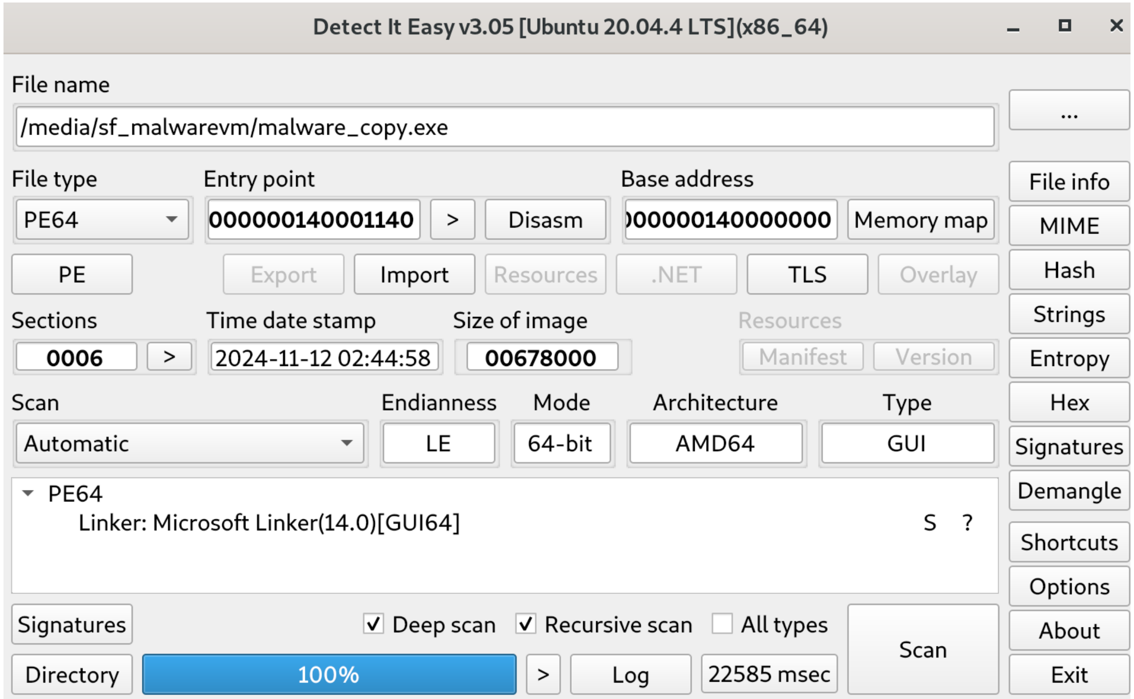
```
remnux@remnux:~$ cp /media/sf_malwarevm/HashUtil.exe /tmp/malware_copy.exe
remnux@remnux:~$ upx -d /tmp/malware_copy.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96        Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   5283328 <-   1851392   35.04%    win64/pe     malware_copy.exe

Unpacked 1 file.
```

a.
    i. Made a temporary file to unpack its content



```
remnux@remnux:~$ cp /tmp/malware_copy.exe /media/sf_malwarevm/
remnux@remnux:~$ ls /media/sf_malwarevm
 exercise1.7z       HashUtil.exe       project2.7z                  x32dbg_dump.exe
 exercise3.7z       HI.c               project3.7z                  x32dbg_dump_SCY.exe
 EXPORTED.c         malware_copy.exe  'System Volume Information'   x32dbg_dump_SCY.exe.bak
 freemem.exe.malz   project1.7z        whoami.exe.malz
remnux@remnux:~$ die malware_copy.exe
```

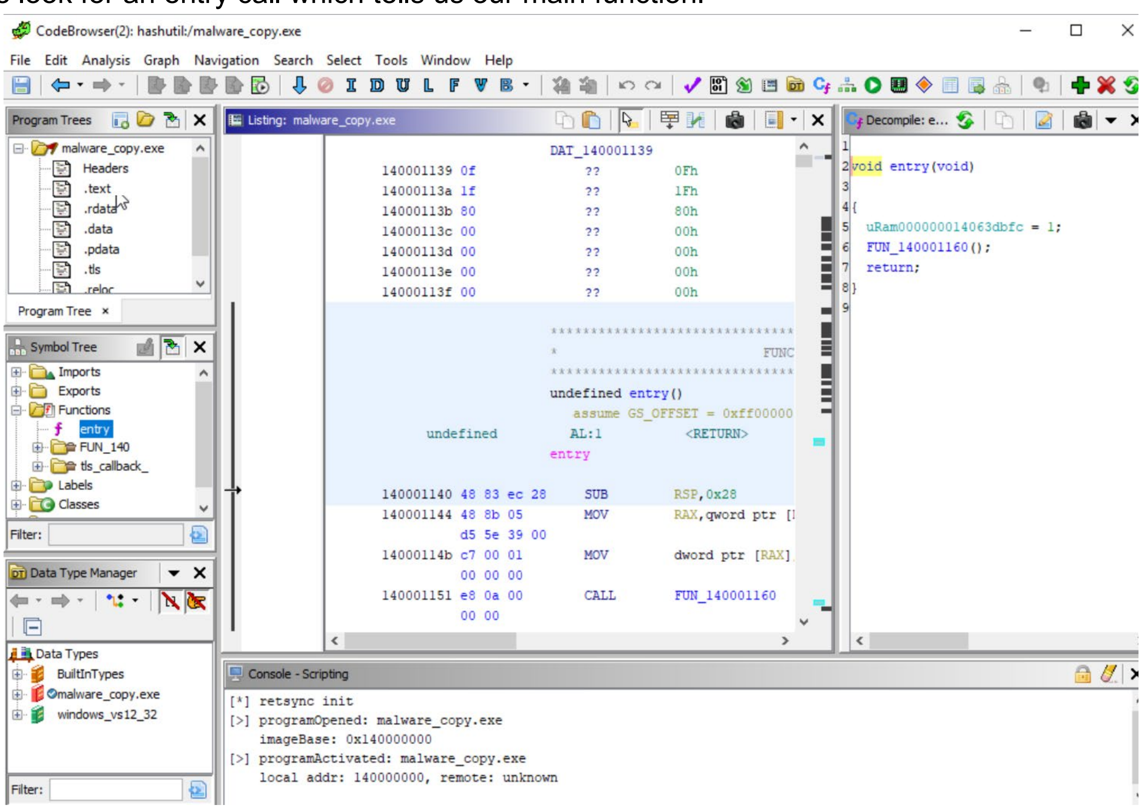
b.
    i. Copied the temporary file back into my locked malware folder

c. 


i. Unpacked file of the HashUtil.exe

## Ghidra

1. First we look for an entry call which tells us our main function.

a. 

2. Name entry main()

a.

```
void main(void)

{
  DAT_14063dbfc = 1;
  FUN_140001160();
  return;
}
```

3. Check FUN_140001160

a.

Decompile: FUN_140001160 - (malware_copy.exe)

```
  auStack_48 = ZEXT816(0);
  auStack_58 = ZEXT816(0);
  auStack_68 = ZEXT816(0);
  auStack_78 = ZEXT816(0);
  auStack_88 = ZEXT816(0);
  auStack_98 = ZEXT816(0);
  uStack_38 = 0;
  if (DAT_14063dbfc != 0) {
    GetStartupInfoA(auStack_98);
  }
  pvVar4 = StackBase;
  LOCK();
  bVar12 = DAT_14063dbd8 == 0;
  DAT_14063dbd8 = DAT_14063dbd8 ^ (ulonglong)bVar12 * (DAT_14063dbd8 ^
  pvVar2 = (void *)(!bVar12 * DAT_14063dbd8);
  while ((!bVar12 && (pvVar4 != pvVar2))) {
    Sleep(1000);
    LOCK();
    bVar12 = DAT_14063dbd8 == 0;
    DAT_14063dbd8 = DAT_14063dbd8 ^ (ulonglong)bVar12 * (DAT_14063dbd8
    pvVar2 = (void *)(!bVar12 * DAT_14063dbd8);
  }
  if (DAT_14063dbd0 == 1) {
    _amsg_exit(0x1f);
```

i. The decompiled malware (maare_copy.ExE) exhibits anti-debugging behavior and possible obfuscation techniques. It initializes stack variables, retrieves startup information via GetStartupInfoA, and employs a looping mechanism with LOCK() and Sleep(1000) to potentially detect or thwart debugging efforts. Conditional logic with XOR operations on

DAT_14063dbd8 and the use of _amsg_exit(0x1f) indicate obfuscation and environment-dependent execution. These patterns suggest the malware is designed to evade detection while adapting its behavior based on the host environment, warranting further dynamic analysis to uncover its purpose and functionality.

4. Fun_1402716b0

    a.
```
undefined8 uStack_80;
int iStack_74;
short *psStack_70;
longlong lStack_68;
uint uStack_5c;

FUN_140319950(0xff);
FUN_14032bf40(0x2bab3);
uVar6 = GetCommandLineW();
ppsVar7 = (short **)CommandLineToArgvW(uVar6,&iStack_74);
if (ppsVar7 != (short **)0x0) {
  psVar8 = *ppsVar7;
  if (psVar8 == (short *)0x0) {
    iVar14 = 0;
  }
  else {
    iVar14 = -1;
    psVar9 = psVar8;
```

        i. The code snippet demonstrates a program's initialization phase, where it invokes specific functions (FUN_140319950 and FUN_14032bf40) with defined parameters, likely as part of its setup or configuration. It retrieves the command-line arguments using GetCommandLineW and processes them with CommandLineToArgvW, storing the result in ppsVar7. A conditional check ensures the returned pointer is valid, with subsequent logic depending on whether psVar8 points to a null value. The iVar14 variable indicates a decision-making outcome, where 0 suggests successful handling and -1 implies a failure or an error state. This segment hints at command-line parameter parsing with error-checking mechanisms.

b.
```
        }
        if (uStack_80._6_1_ != '\0') {
          FUN_140314750(&puStack_88);
        }
        lVar10 = lVar10 + 1;
      } while (lVar10 < iStack_74);
    }
    LocalFree(ppsVar7);
  }
  psVar8 = (short *)GetEnvironmentStringsW();
  psStack_70 = psVar8;
  if (*psVar8 != 0) {
    do {
      uVar13 = 0;
```

c.
```
          piVarl = (int *)((longlong)ppppuStack_98 + -4);
          *piVarl = *piVarl + -1;
          if (*piVarl == 0) {
            FUN_1403130f0((longlong)ppppuStack_98 + -4);
          }
        }
      }
    } while (*psVar8 != 0);
  }
  FreeEnvironmentStringsW(psStack_70);
  FUN_140270d90();
  return;
```

i. The snippet illustrates a loop-driven process where the program iterates through command-line arguments or environment strings, performing cleanup and resource management tasks. It checks a condition (uStack_80._6_1_ != '\0'), invokes FUN_140314750, and increments iVar10 until the counter matches iStack_74. Post-loop, it releases allocated memory (LocalFree) and retrieves environment strings using GetEnvironmentStringsW, storing them in psStack_70. Within another loop, it processes these strings, managing a decrement operation on a counter (*piVarl), and invokes FUN_1403130f0 when the counter reaches zero. After the strings are fully handled, it calls FreeEnvironmentStringsW to release the memory and concludes with a cleanup function

(FUN_140270d90). This flow highlights structured memory management and environmental interaction within the program.

5. Fun_14014efe0

a.
```
if (iVar4 == 0) {
  uVar5 = FUN_1400f66e0(&DAT_1405a52d8);
  EnterCriticalSection(uVar5);
}
FUN_1402839b0(&LAB_14014efa0);
DAT_1405a6718 = GetCurrentThreadId();
auStack_28 = ZEXT816(0);
auStack_38 = ZEXT816(0);
pwStack_18 = (wchar_t *)0x0;
auStack_58._4_12_ = SUB1612(ZEXT816(0) >> 0x20,0);
auStack_58 = CONCAT124(auStack_58._4_12_,0xb);
auStack_58 = CONCAT88(0x14014f4d0,auStack_58._0_8_);
auStack_48 = ZEXT816(param_1) << 0x40;
DAT_1405a6690 = param_1;
uVar5 = LoadCursorA(0,0x7f00);
auStack_38 = CONCAT88(uVar5,auStack_38._0_8_);
auStack_28 = CONCAT88(auStack_28._8_8_,6);
pwStack_18 = L"UPP-CLASS-W";
RegisterClassW(auStack_58);
auStack_58._0_4_ = 0x2000b;
pwStack_18 = L"UPP-CLASS-DS-W";
RegisterClassW(auStack_58);
auStack_58._0_4_ = 0x80b;
```

i. The provided code segment appears to set up a multithreaded environment while configuring window classes in a Windows application. If iVar4 equals zero, a function (FUN_1400f66e0) is called to retrieve a critical section pointer, which is subsequently locked using EnterCriticalSection. The thread ID is stored in DAT_1405a6718, and stack variables are initialized with extended integer (ZEXT816) operations for consistent handling of large values. It prepares window class structures (auStack_58) with parameters, such as cursor types (via LoadCursorA) and unique class names (L"UPP-CLASS-W", L"UPP-CLASS-DS-W"). These classes are registered using RegisterClassW, enabling their use in GUI element creation. The focus on thread safety, parameter configuration, and resource setup indicates the foundation of a graphical or multi-threaded application.

```
pwStack_18 = L"UPP-CLASS-SB-A";
RegisterClassA(auStack_58);
auStack_58 = CONCAT124(auStack_58._4_12_,0x2080b);
pwStack_18 = L"UPP-CLASS-SB-DS-A";
RegisterClassA(auStack_58);
auStack_28 = ZEXT816(0);
auStack_38 = ZEXT816(0);
auStack_48 = ZEXT816(param_1) << 0x40;
pwStack_18 = L"UPP-TIMER";
auStack_58 = ZEXT816(0x14014eab0) << 0x40;
RegisterClassA(auStack_58);
FUN_1401398f0();
DAT_1405a6710 =
     CreateWindowExA(0,"UPP-TIMER",&DAT_1403cfe30,0,0x80000000,0x800
                     ZEXT816(0),param_1,0);
SetTimer(DAT_1405a6710,1,10,0);
if (DAT_1405a6770 == '\0') {
  iVar4 = FUN_140369514(&DAT_1405a6770);
  if (iVar4 != 0) {
    FUN_14028f670(&DAT_1405a6738);
    _DAT_1405a6760 = ZEXT816(0);
    _DAT_1405a6750 = ZEXT816(0);
    FUN_140001440(&LAB_140150150);
```

b.

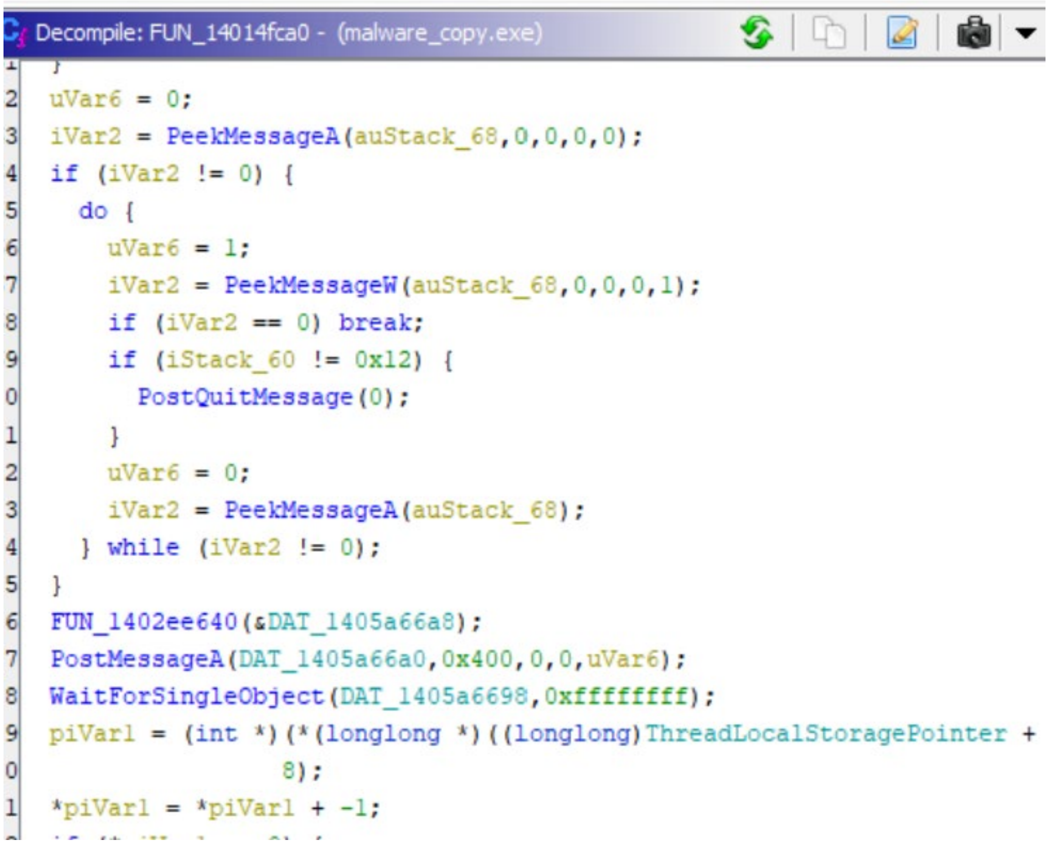
i. The code snippet is part of a Windows application that sets up several window classes and initializes a timer-based mechanism. RegisterClassA is used to register ASCII window classes such as "UPP-CLASS-SB-DS-A" and "UPP-TIMER", where class properties are configured in the auStack_58 structure. After registering the timer class, a window is created using CreateWindowExA with the "UPP-TIMER" class, associating it with DAT_1405a6710. A timer is then started on this window using SetTimer, with a 10-millisecond interval. The subsequent logic checks a global flag (DAT_1405a6770) to execute additional initialization routines, such as FUN_140369514 and FUN_14028f670, which likely set up specific application states. The use of helper functions like FUN_140001440 suggests further configuration or event handling related to the application's GUI or processing loop. This setup forms the backbone of a timer-driven or event-based application.

c.

```
*(int *)(*(longlong *)((longlong)ThreadLocalStoragePointer + (ulongl
     iVar3;
if (iVar3 == 0) {
  uVar5 = FUN_1400f66e0(&DAT_1405a52d8);
  LeaveCriticalSection(uVar5);
  iVar3 = *(int *)(*(longlong *)((longlong)ThreadLocalStoragePointer
                  + 8);
}
iVar4 = iVar3 + 1;
*(int *)(*(longlong *)((longlong)ThreadLocalStoragePointer + (ulongl
     iVar4;
if (iVar3 == 0) {
  uVar5 = FUN_1400f66e0(&DAT_1405a52d8);
  EnterCriticalSection(uVar5);
  iVar4 = *(int *)(*(longlong *)((longlong)ThreadLocalStoragePointer
                  + 8);
}
*(int *)(*(longlong *)((longlong)ThreadLocalStoragePointer + (ulongl
     iVar4 + -1;
if (iVar4 + -1 == 0) {
  uVar5 = FUN_1400f66e0(&DAT_1405a52d8);
  LeaveCriticalSection(uVar5);
```

   i. The provided code snippet seems to be part of a multithreading or synchronization mechanism, likely in a Windows environment, utilizing critical sections for thread safety. The code checks and modifies a value stored at a memory location relative to ThreadLocalStoragePointer. If iVar3 is zero, it enters a critical section using EnterCriticalSection, performs an operation, and then increments or decrements a value (likely a reference counter or state flag). After modifying the value, it checks whether it should leave the critical section by comparing iVar4 (the updated value) with zero. If it decrements the counter to zero, it exits the critical section using LeaveCriticalSection. The function FUN_1400f66e0 is likely fetching or manipulating a handle for the critical section object. This type of code structure is common in situations where thread synchronization is necessary to manage access to shared resources.

6. Fun_14014fca0

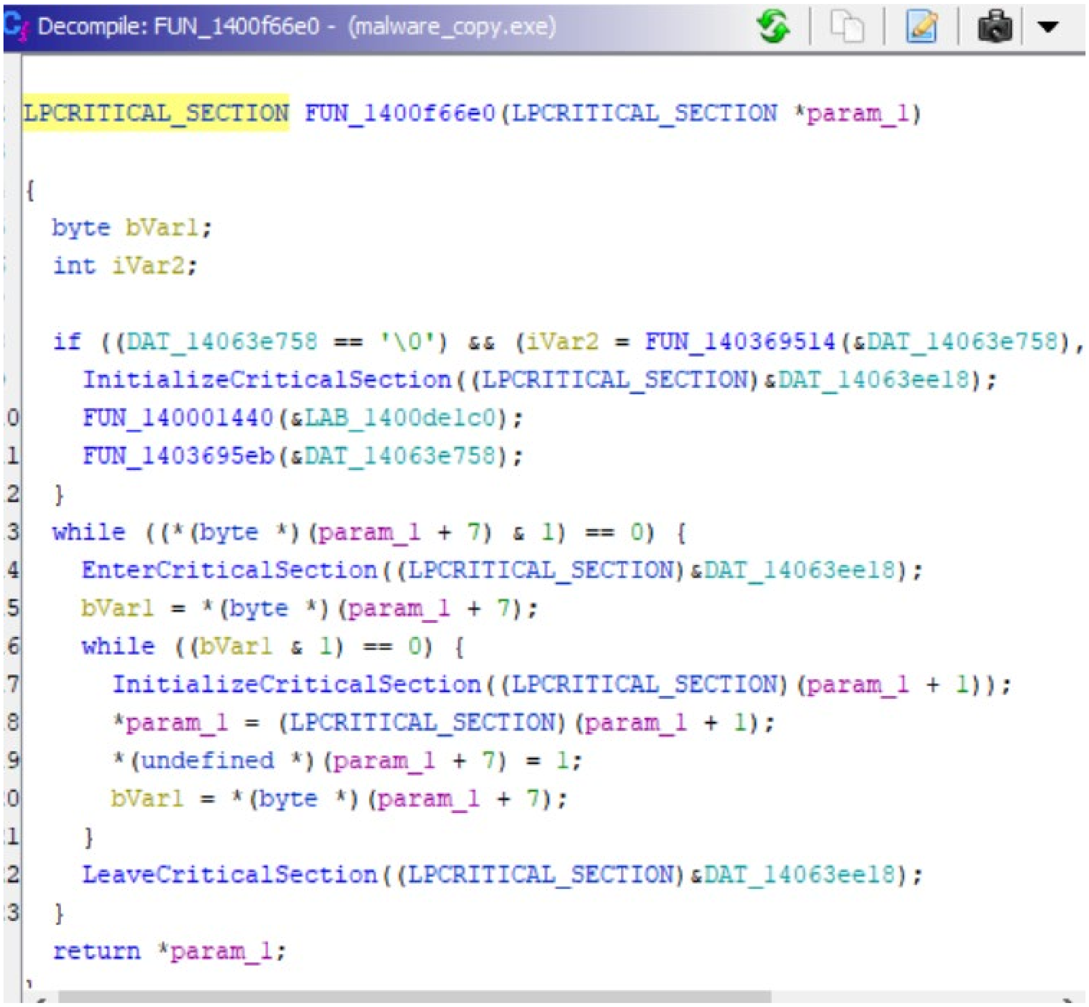
a. 

Decompile: FUN_14014fca0 - (malware_copy.exe)

```
}
uVar6 = 0;
iVar2 = PeekMessageA(auStack_68,0,0,0,0);
if (iVar2 != 0) {
  do {
    uVar6 = 1;
    iVar2 = PeekMessageW(auStack_68,0,0,0,1);
    if (iVar2 == 0) break;
    if (iStack_60 != 0x12) {
      PostQuitMessage(0);
    }
    uVar6 = 0;
    iVar2 = PeekMessageA(auStack_68);
  } while (iVar2 != 0);
}
FUN_1402ee640(&DAT_1405a66a8);
PostMessageA(DAT_1405a66a0,0x400,0,0,uVar6);
WaitForSingleObject(DAT_1405a6698,0xffffffff);
piVar1 = (int *)(*(longlong *)((longlong)ThreadLocalStoragePointer +
                 8);
*piVar1 = *piVar1 + -1;
```

   i. The provided decompiled code snippet is part of a larger function, likely from a Windows application, which interacts with the message loop and performs various operations related to thread synchronization and message posting. The code starts by calling PeekMessageA to check if there are any messages in the message queue. If a message is found, it enters a loop, where it checks the message type using PeekMessageW, processes it, and may post a quit message with PostQuitMessage if the message type is not 0x12. The variable uVar6 is used to track the status of the message loop, potentially indicating whether the loop should continue running or exit. After processing the messages, the function calls FUN_1402ee640 (likely another function within the malware) and posts a message with PostMessageA to the window identified by DAT_1405a66a0. It then waits for a signal using WaitForSingleObject on DAT_1405a6698, possibly waiting for a specific event or thread to finish. Finally, the code accesses a thread-local storage pointer and decrements a value, indicating the thread might be tracking some reference count or state. This type of functionality suggests that the malware is manipulating the message queue for its process while managing thread synchronization.

7. Fun_1440f66e0

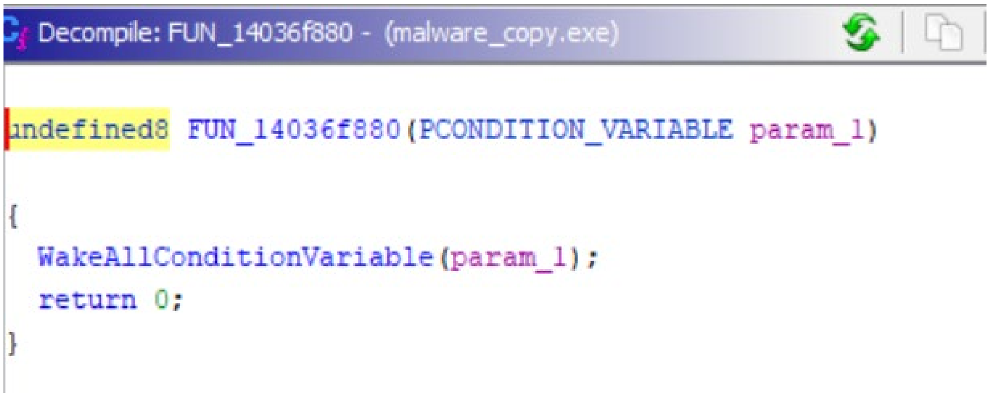
a. 

Decompile: FUN_1400f66e0 - (malware_copy.exe)

```
LPCRITICAL_SECTION FUN_1400f66e0(LPCRITICAL_SECTION *param_1)

{
  byte bVar1;
  int iVar2;

  if ((DAT_14063e758 == '\0') && (iVar2 = FUN_140369514(&DAT_14063e758),
    InitializeCriticalSection((LPCRITICAL_SECTION)&DAT_14063ee18);
    FUN_140001440(&LAB_1400de1c0);
    FUN_1403695eb(&DAT_14063e758);
  }
  while ((*(byte *)(param_1 + 7) & 1) == 0) {
    EnterCriticalSection((LPCRITICAL_SECTION)&DAT_14063ee18);
    bVar1 = *(byte *)(param_1 + 7);
    while ((bVar1 & 1) == 0) {
      InitializeCriticalSection((LPCRITICAL_SECTION)(param_1 + 1));
      *param_1 = (LPCRITICAL_SECTION)(param_1 + 1);
      *(undefined *)(param_1 + 7) = 1;
      bVar1 = *(byte *)(param_1 + 7);
    }
    LeaveCriticalSection((LPCRITICAL_SECTION)&DAT_14063ee18);
  }
  return *param_1;
}
```

i. The decompiled code represents a function that initializes and manages critical sections for thread synchronization in a malware program. It first checks if a global variable DAT_14063e758 is zero and, if so, initializes a critical section. The function then enters a loop, checking specific byte flags within the passed param_1 (the critical section pointer) to determine if further initialization is needed. If the condition isn't met, it enters a different critical section, sets flags, and ensures the critical section is properly initialized before leaving the section. After the necessary steps, it returns the updated critical section pointer, ensuring that access to shared resources is synchronized to avoid conflicts among threads, which is typical in malware for controlling resources while executing malicious activities.

8. Fun_14036f880

a. 

```
Decompile: FUN_14036f880 - (malware_copy.exe)

undefined8 FUN_14036f880(PCONDITION_VARIABLE param_1)

{
  WakeAllConditionVariable(param_1);
  return 0;
}
```

   i. The decompiled function FUN_14036f880 simply calls WakeAllConditionVariable on the provided param_1, which is a pointer to a condition variable. This function releases all threads that are currently waiting on the specified condition variable, effectively signaling them to continue execution. After waking the waiting threads, the function returns 0, indicating successful execution. This is likely part of a synchronization mechanism in the malware to coordinate multiple threads in its execution flow.

9. Fun_14036f840

a. 

```
Decompile: FUN_14036f840 - (malware_copy.exe)

undefined8 FUN_14036f840(PSRWLOCK param_1)

{
  ReleaseSRWLockExclusive(param_1);
  return 0;
}
```

   i. The decompiled function FUN_14036f840 takes a pointer to a SRWLOCK (Spin-Release-Write Lock) param_1 and releases it using the ReleaseSRWLockExclusive function. After successfully releasing the lock, the function then returns 0. This is likely part of a synchronization mechanism within the malware to control access to shared resources, allowing multiple threads to safely read and write data in a concurrent environment.

10. Fun _1403695eb

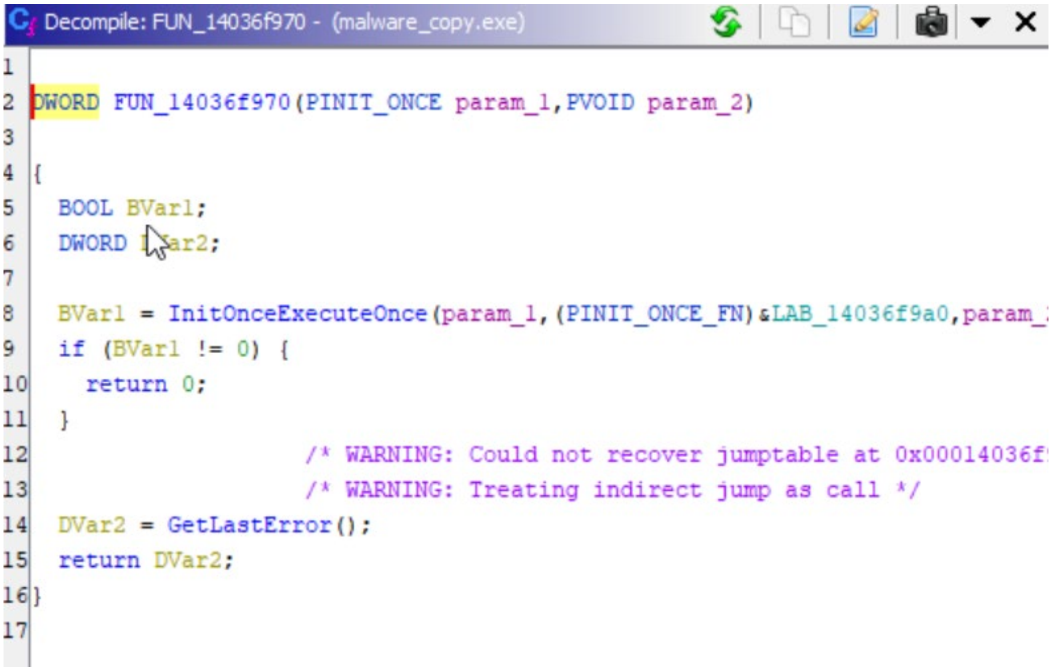
a.

Decompile: FUN_1403695eb - (malware_copy.exe)

```
  if (iVar4 == 0) {
    bVarl = param_1[1];
    param_1[1] = 1;
    iVar4 = FUN_14036f840(&DAT_14063e958);
    if (iVar4 == 0) {
      if ((bVarl & 4) == 0) {
        return;
      }
      iVar4 = FUN_14036f880(&DAT_14063e960);
      if (iVar4 == 0) {
        return;
      }
    }
    else {
      FUN_14036fbc0("%s failed to release mutex","__cxa_guard_release");
    }
    pcVar5 = "%s failed to broadcast";
  }
  else {
    pcVar5 = "%s failed to acquire mutex";
  }
  uVar3 = FUN_14036fbc0(pcVar5,"__cxa_guard_release");
  FUN_140002840(uVar3);
  pcVar2 = (code *)swi(3);
  (*pcVar2)();
```

i. The decompiled function FUN_1403695eb handles a sequence of operations related to mutex acquisition and condition broadcasting. Initially, it retrieves a value from param_1[1], sets param_1[1] to 1, and then attempts to acquire a mutex by calling FUN_140361840 with a reference to DAT_14063e958. If the mutex acquisition fails (i.e., iVar4 != 0), it checks if a specific condition bit (4) is set in bVarl. If this condition is not met, the function returns. If the condition is met, it attempts to release a mutex by calling FUN_14036f880. If this fails as well, an error message is logged. The function also includes error handling for mutex-related failures, using FUN_14036fbc0 to log failure messages for acquiring or releasing mutexes, followed by a call to FUN_140002840. If further error handling is required, a system-level interrupt (swi(3)) is triggered, likely leading to a crash or specific exception handling mechanism.

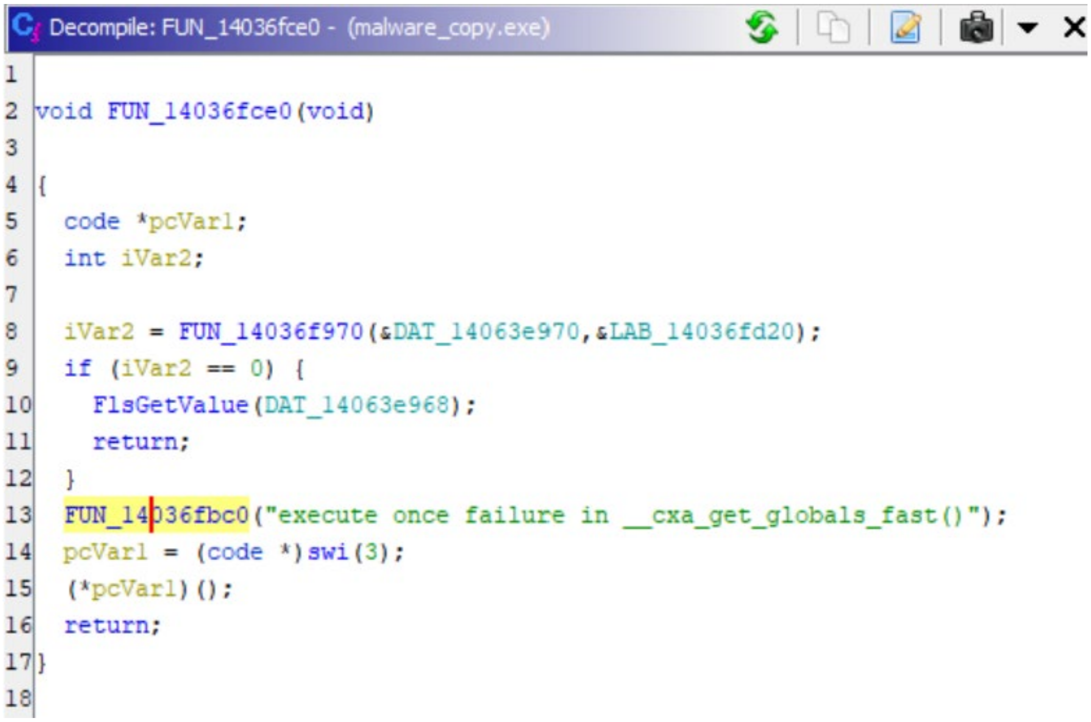
11. Fun_14036f970

a. 

```
Decompile: FUN_14036f970 - (malware_copy.exe)

DWORD FUN_14036f970(PINIT_ONCE param_1,PVOID param_2)

{
  BOOL BVar1;
  DWORD DVar2;

  BVar1 = InitOnceExecuteOnce(param_1,(PINIT_ONCE_FN)&LAB_14036f9a0,param_
  if (BVar1 != 0) {
    return 0;
  }
                    /* WARNING: Could not recover jumptable at 0x00014036f
                    /* WARNING: Treating indirect jump as call */
  DVar2 = GetLastError();
  return DVar2;
}
```

i. The decompiled function FUN_14036f970 uses the InitOnceExecuteOnce function to execute a specific initialization routine (LAB_14036f9a0) only once. The function receives param_1 (a pointer to an INIT_ONCE structure) and param_2 (a parameter passed to the initialization function). It first checks if the initialization was successful by evaluating the BVarl boolean, which stores the return value of InitOnceExecuteOnce. If the initialization is successful (BVarl != 0), the function returns 0. If an error occurs during the initialization, the function retrieves the last error code using GetLastError() and returns this error code as DVar2, signaling the failure of the initialization process.

12. Fun_14036fce0




a.
   i. The decompiled function FUN_14036fce0 first calls FUN_140361970 with two parameters: &DAT_14063e970 and &LAB_14036fd20. If this function returns 0, it proceeds to call FlsGetValue with the global variable DAT_14063e968 and then returns. If the call to FUN_140361970 does not return 0, indicating a failure, the function calls FUN_14036fbc0 to log an error message ("execute once failure in _cxa_get_globals_fast ()"). Following this, it performs a system call using swi(3) to invoke a function ((*pcVar1)()), likely related to error handling or cleanup, and then returns.

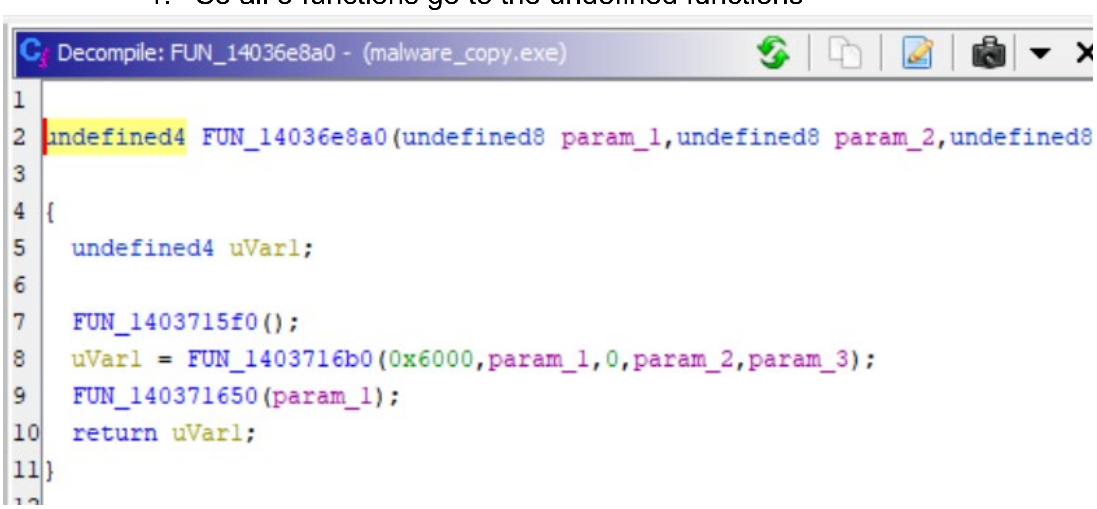
13. I keep coming back to these two pages wherever functions I traverse to these two pages keep popping up which is very suspicious

a. 

Decompile: FUN_14036fbc0 - (malware_copy.exe)

```
void FUN_14036fbc0(undefined8 param_1,undefined8 param_2,undefined8 param_

{
  undefined *puVar1;
  undefined8 uVar2;
  undefined8 local_res10;
  undefined8 local_res18;
  undefined8 local_res20;

  puVar1 = PTR_FUN_1404d55d0;
  local_res10 = param_2;
  local_res18 = param_3;
  local_res20 = param_4;
  uVar2 = (*(code *)PTR_FUN_1404d55d0)(2);
  FUN_14036d970(uVar2,"libc++abi: ");
  uVar2 = (*(code *)puVar1)(2);
  thunk_FUN_14036e8a0(uVar2,param_1,&local_res10);
  uVar2 = (*(code *)puVar1)(2);
  FUN_14036d970(uVar2,&DAT_1403fe60c);
                    /* WARNING: Subroutine does not return */
  abort();
}
```

   i. The first Fun_14036d970 with libc++abi turns me the undefined4 fun_1403638a0 function
   ii. The second thunk_fun also turns to the same 3 functions
   iii. The third function goes to the same 3 functions as well
      1. So all 3 functions go to the undefined functions

b. 

Decompile: FUN_14036e8a0 - (malware_copy.exe)

```
undefined4 FUN_14036e8a0(undefined8 param_1,undefined8 param_2,undefined8

{
  undefined4 uVar1;

  FUN_1403715f0();
  uVar1 = FUN_1403716b0(0x6000,param_1,0,param_2,param_3);
  FUN_140371650(param_1);
  return uVar1;
}
```

   i. The first fun function 1403615f0 goes to this page

1. Decompile: FUN_14036f590 - (malware_copy.exe)

```

FILE * FUN_14036f590(uint param_1)

{
  FILE *pFVar1;

  pFVar1 = __iob_func();
  return pFVar1 + param_1;
}

```

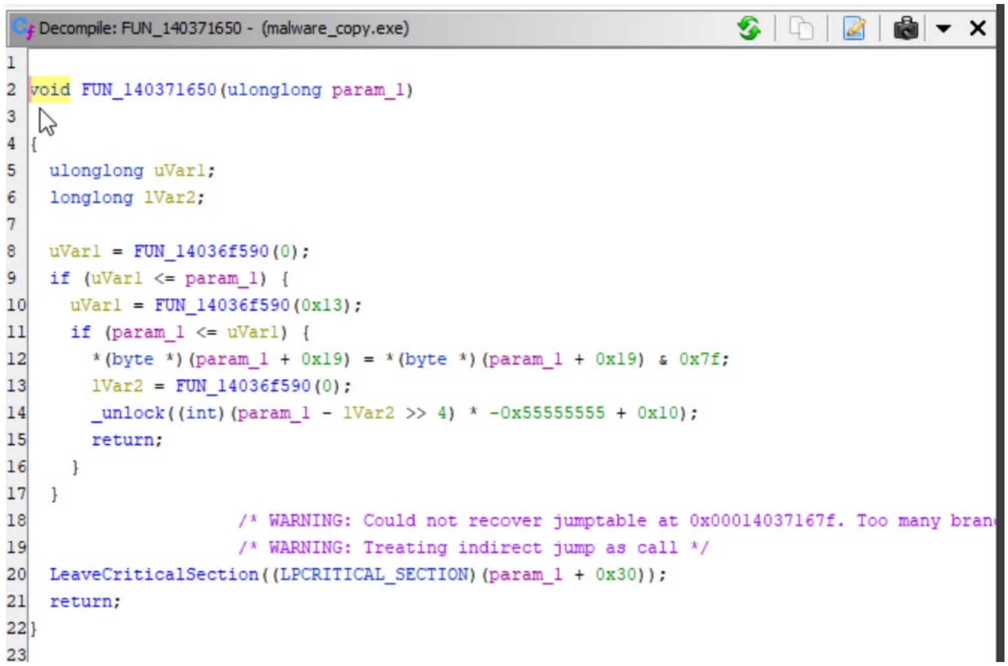
ii. The second fun function with 4 parameters

1. Decompile: FUN_1403716b0 - (malware_copy.exe)

```
  local_128 = param_2;
  local_120 = param_1;
LAB_14037175f:
  do {
    pcVar15 = param_4 + 1;
    cVar1 = *param_4;
    if (cVar1 == 0) {
      return local_104;
    }
    param_4 = pcVar15;
    if (cVar1 != '%') {
      if (((local_120 & 0x4000) != 0) || (local_104 < local_100)) {
        if ((local_120 & 0x2000) != 0) {
          fputc((int)cVar1,local_128);
          local_104 = local_104 + 1;
          goto LAB_14037175f;
        }
        *(char *)((longlong)&local_128->_ptr + (longlong)local_104) = cVar1;
      }
      local_104 = local_104 + 1;
      goto LAB_14037175f;
    }
    local_11c = (undefined  [8])0xffffffffffffffff;
    cVar1 = *pcVar15;
```

   a. We see a bunch of goto LAB_14037175f

iii. The third function

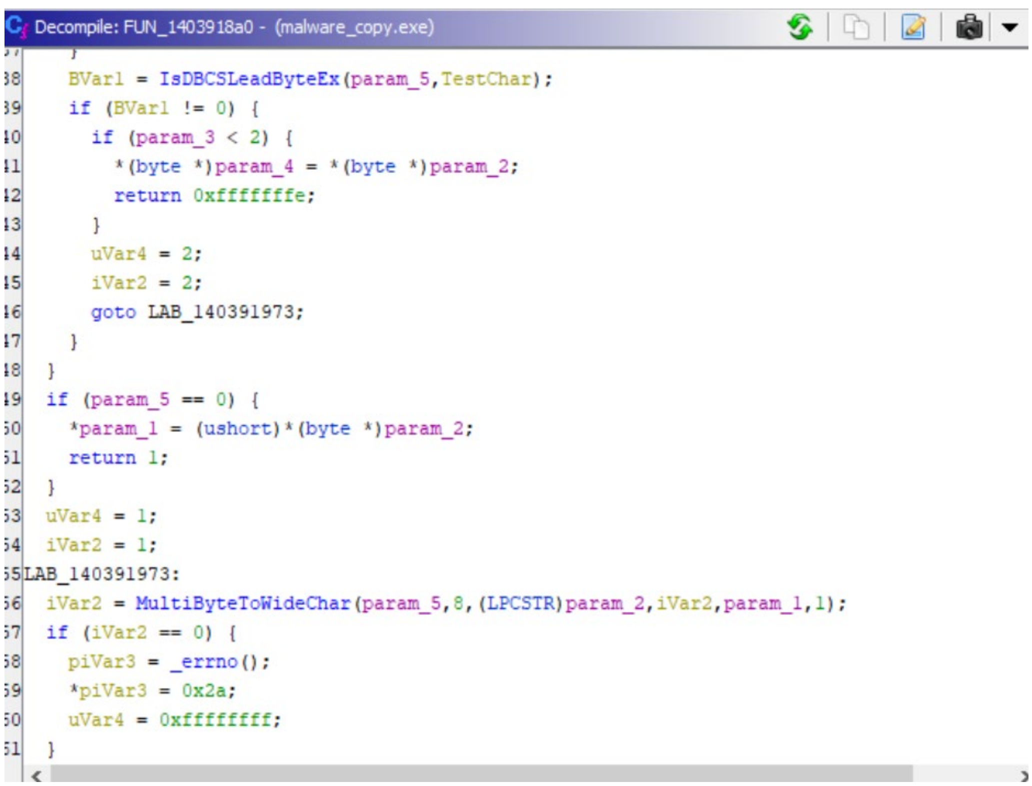


Decompile: FUN_140371650 - (malware_copy.exe)

```

void FUN_140371650(ulonglong param_1)

{
  ulonglong uVar1;
  longlong lVar2;

  uVar1 = FUN_14036f590(0);
  if (uVar1 <= param_1) {
    uVar1 = FUN_14036f590(0x13);
    if (param_1 <= uVar1) {
      *(byte *)(param_1 + 0x19) = *(byte *)(param_1 + 0x19) & 0x7f;
      lVar2 = FUN_14036f590(0);
      _unlock((int)(param_1 - lVar2 >> 4) * -0x55555555 + 0x10);
      return;
    }
  }
                    /* WARNING: Could not recover jumptable at 0x00014037167f. Too many bran
                    /* WARNING: Treating indirect jump as call */
  LeaveCriticalSection((LPCRITICAL_SECTION)(param_1 + 0x30));
  return;
}

```

1.
    a. Goes to leave critical selection

14. Fun_1403918a0



Decompile: FUN_1403918a0 - (malware_copy.exe)

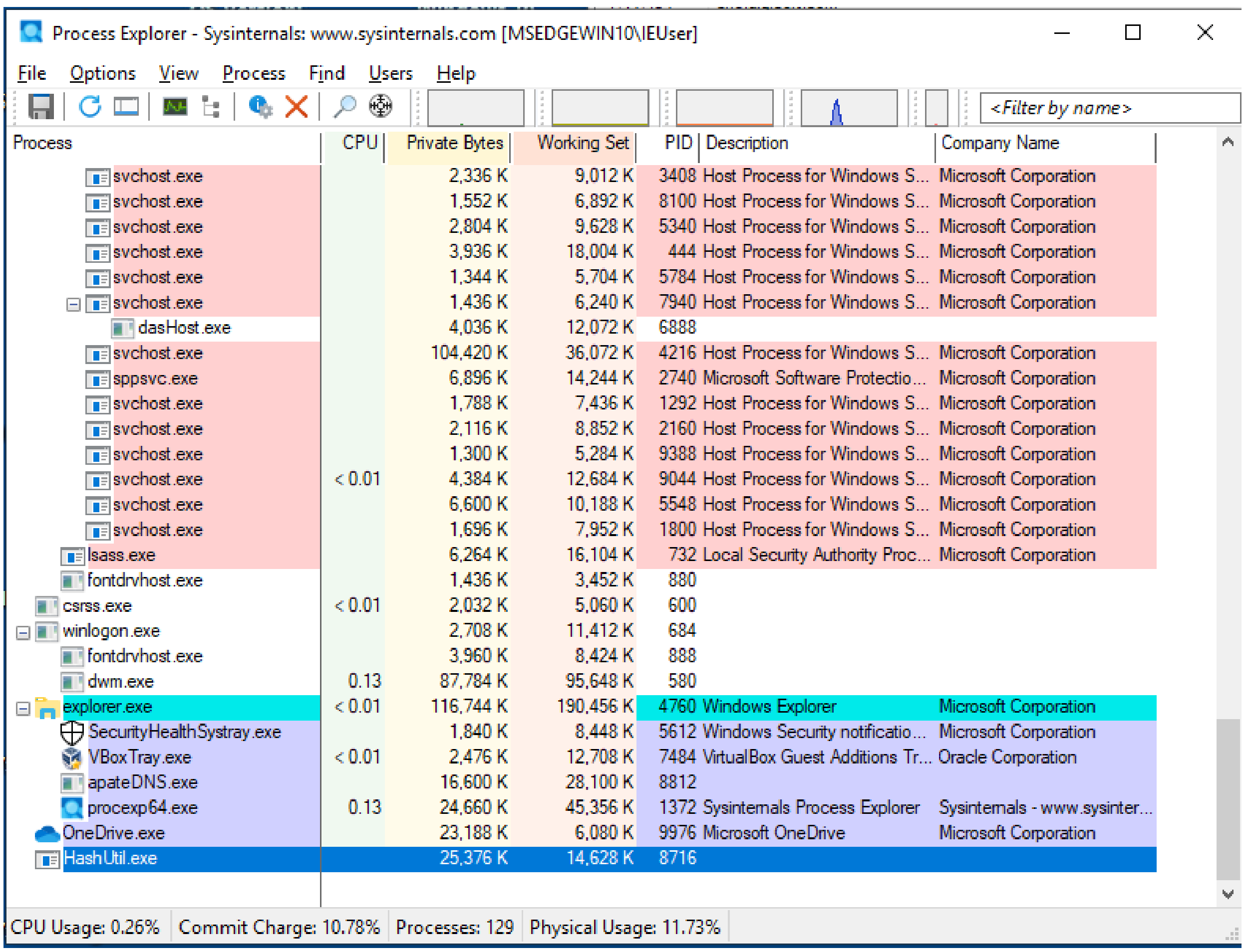
```
    BVar1 = IsDBCSLeadByteEx(param_5,TestChar);
    if (BVar1 != 0) {
      if (param_3 < 2) {
        *(byte *)param_4 = *(byte *)param_2;
        return 0xfffffffe;
      }
      uVar4 = 2;
      iVar2 = 2;
      goto LAB_140391973;
    }
  }
  if (param_5 == 0) {
    *param_1 = (ushort)*(byte *)param_2;
    return 1;
  }
  uVar4 = 1;
  iVar2 = 1;
LAB_140391973:
  iVar2 = MultiByteToWideChar(param_5,8,(LPCSTR)param_2,iVar2,param_1,1);
  if (iVar2 == 0) {
    piVar3 = _errno();
    *piVar3 = 0x2a;
    uVar4 = 0xffffffff;
  }
```

a.
    i. The decompiled function FUN_1403918a0 starts by calling IsDBCSLeadByteEx with parameters param_5 and TestChar, storing the result in BVarl. If BVarl is non-zero, the function checks if param_3 is less than 2. If so, it copies the byte at param_2 to the location pointed by param_4, and then returns 0xfffffffe. Otherwise, it sets uVar4 and iVar2 to

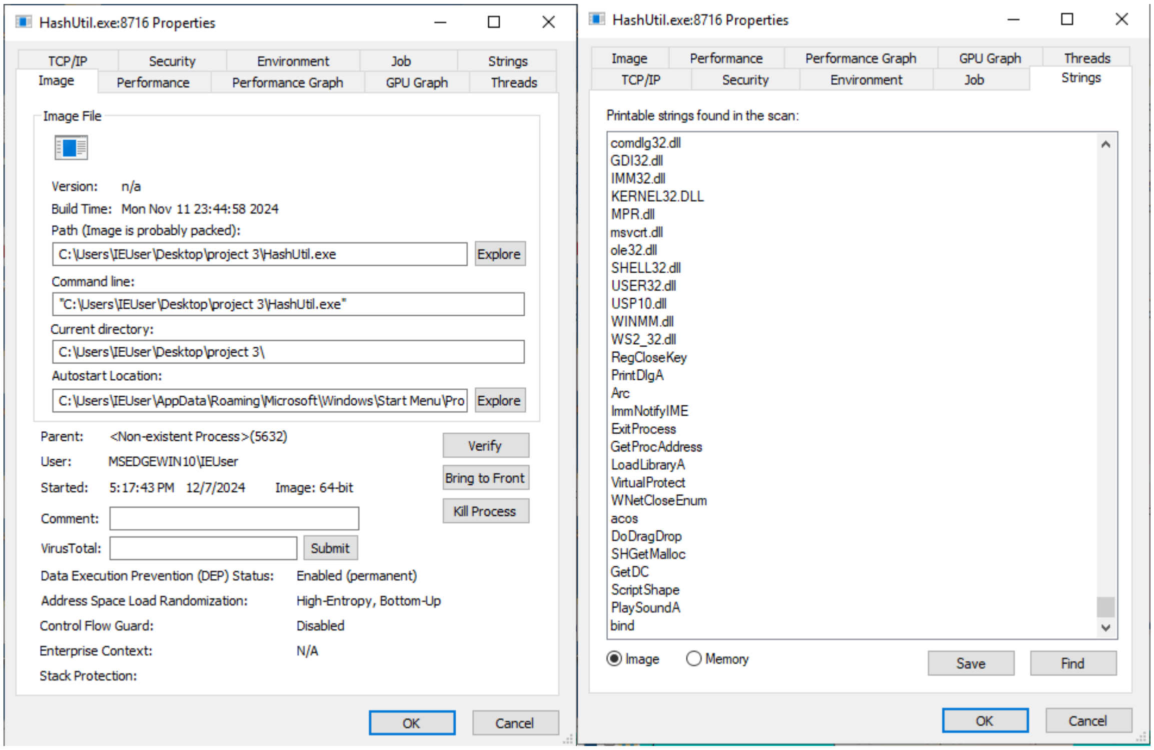
2, before jumping to the label LAB_140391973. If param_5 is zero, it assigns the byte pointed to by param_2 to param_1 as a ushort, and returns 1. At label LAB_140391973, it calls MultiByteToWideChar to convert a single byte from param_2 to a wide character, and stores the result in param_1. If the conversion fails (i.e., iVar2 is not 0), it sets the error number (errno()) to 0x2a and returns 0xffffffff.

### Process Explorer

1. Opened up Process Explorer as provided by the Sysinternals Suite. Once opened, despite the malware executable window being exited out, Process Explorer still considered it running.




2. Then double clicked on the executable and was greeted with this information.




3. Within the Strings tab, we are presented with detailed information:

   b. KERNEL32.DLL, USER32.DLL, and SHELL32.DLL suggest that the malware utilizes critical system libraries to interact deeply with the operating system. While this is common for legitimate programs, it also indicates the possibility that the malware is leveraging these APIs to conceal its presence and perform potentially harmful actions, such as accessing system resources or modifying files.

      i. KERNEL32.DLL provides essential APIs for interacting with the Windows kernel. Malware often abuses these functions for process management, memory manipulation, or system information gathering.

         1. VirtualAlloc, VirtualProtect: Allocates or changes memory protection for malicious payload executions

         2. WriteProcessMemory: Injects malicious code into processes

        3. CreateThread: Creates new threads  to execute malicious code independently

    ii. USER32.DLL handles user interface elements and is commonly used by malware to interact with or manipulate the victim's system visually.

        1. SetWindowsHookEx: Sets a hook to intercept input events like keystrokes
        2. GetForegroundWindow, GetAsyncKeyState: Tracks user activity or captures sensitive input

    iii. SHELL32.DLL provides APIs for interacting with the Windows shell, such as file operations and launching processes.

        1. SHFileOperation: Copies, moves, delete, or renames files - potentially malicious files for spreading or persistence
        2. ShellExecute: Executes files or commands, used to launch other malware components
        3. SHGetSpecialFolderPath: Access key system directories

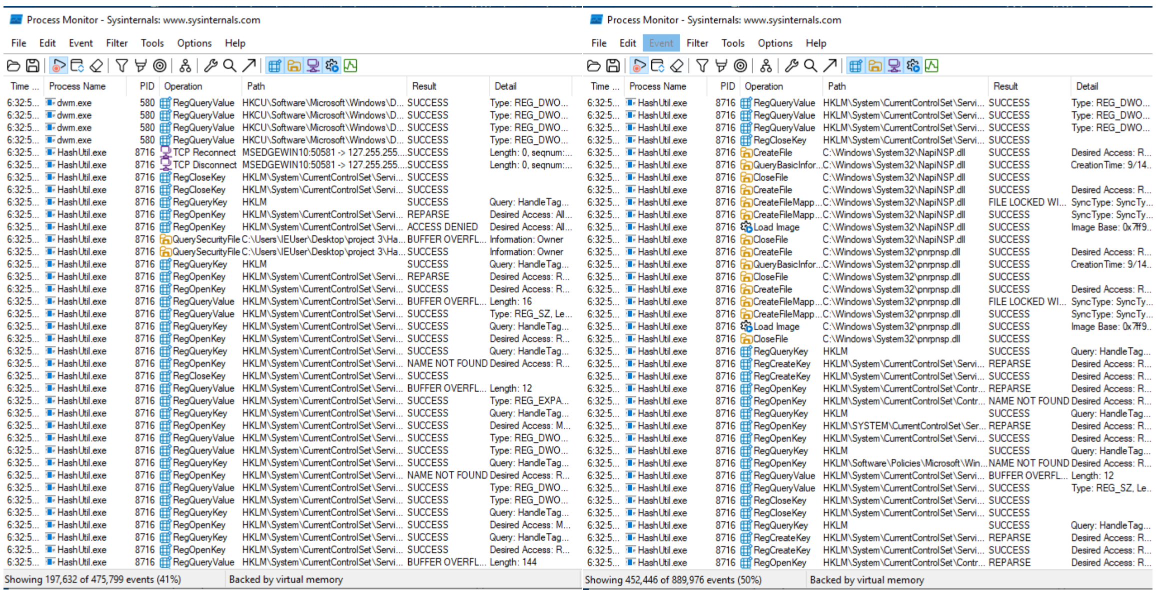
c. LoadLibraryA and GetProcAddress indicate that the malware loads functions dynamically at runtime. This approach can make static analysis more challenging, as the functionality of the malware is only revealed when the executable is run.

d. VirtualProtect allows the malware to change memory permissions, enabling it to execute code directly from memory. This behavior suggests the potential for evading detection, as the malicious code does not need to be written to disk, making it harder for traditional antivirus solutions to detect and analyze.

e. DoDragDrop and SHGetMalloc suggest that the malware may have the capability to manipulate the user interface, potentially creating fake windows, pop-ups, or other deceptive elements.

**Process Monitor**

1. Opened up Process Monitor as provided by the Sysinternals Suite. Once opened, the processes of the malware were observed.




a. From this, we can see that the malware interacts with critical system registry keys, likely attempting to ensure persistence by modifying system services or startup configurations. This behavior enables it to remain active even after a system reboot. This in turn could imply that it has long-term intent.

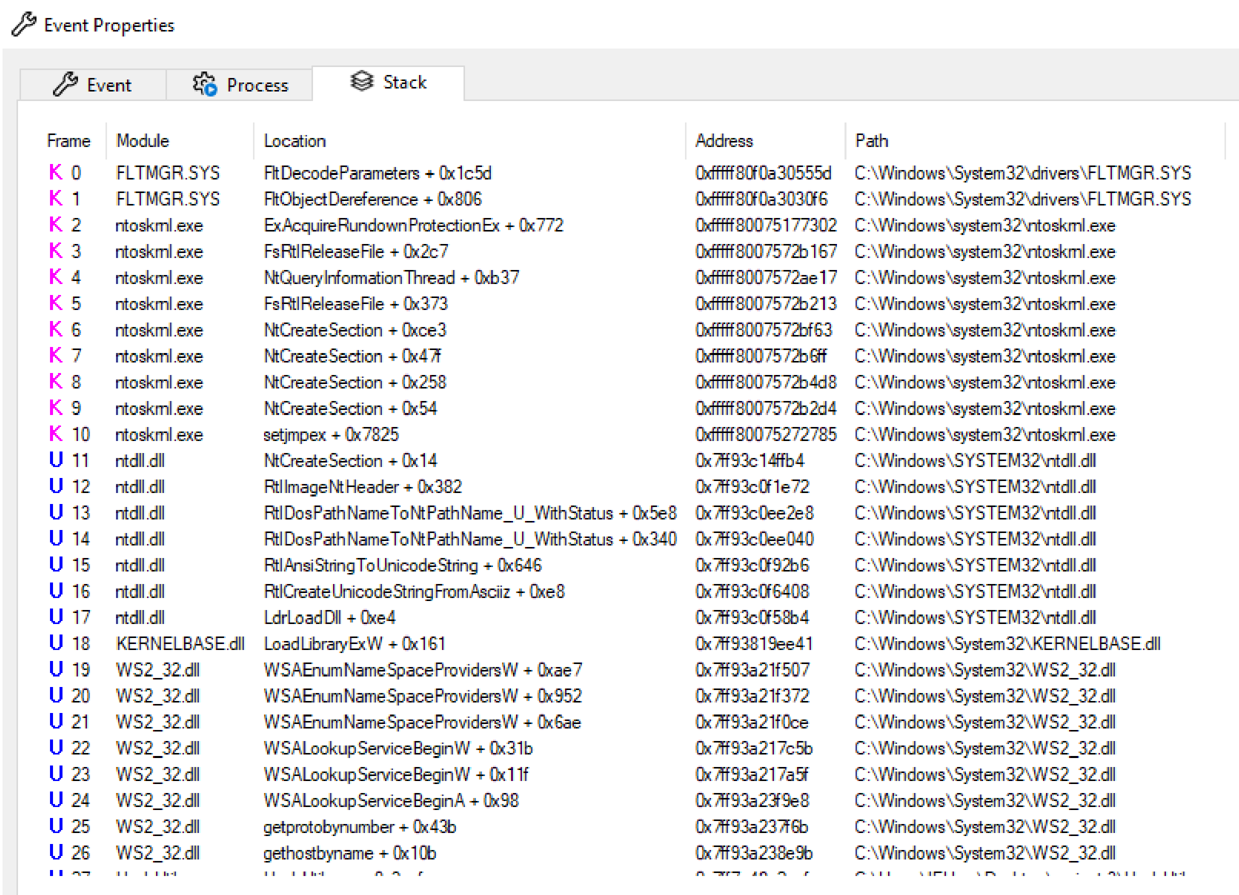
i. The malware created an entry in the RUN key to ensure it starts automatically open system reboot

ii. In - HKCU\Software\Policies\Microsoft\Windows\System - These actions suggest attempts to disable windows security features such as User Account Control or Windows Defender

iii. The malware accessed numerous registry keys under HKLM\SYSTEM\CurrentControlSet\Services, indicating efforts to discover or modify system services.

b. Having access to DLLs such as napinsp.dll and pnrpnsp.dll as well as registry queries further enhance our knowledge of the malware attempting to contact a C2 server (as shown earlier from our Step 1.)

2. Double clicked on one of the CreateFileMapping operations that correlated with the malware. Afterwards, clicked on the Stack tab and was presented with this.

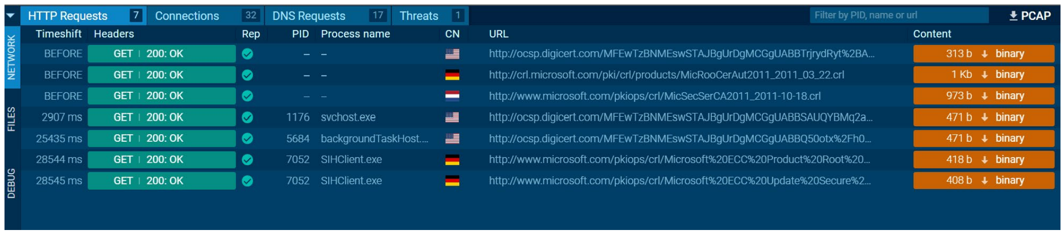



| Frame | Module | Location | Address | Path |
|---|---|---|---|---|
| K 0 | FLTMGR.SYS | FltDecodeParameters + 0x1c5d | 0xfffff80f0a30555d | C:\Windows\System32\drivers\FLTMGR.SYS |
| K 1 | FLTMGR.SYS | FltObjectDereference + 0x806 | 0xfffff80f0a3030f6 | C:\Windows\System32\drivers\FLTMGR.SYS |
| K 2 | ntoskrnl.exe | ExAcquireRundownProtectionEx + 0x772 | 0xfffff80075177302 | C:\Windows\system32\ntoskrnl.exe |
| K 3 | ntoskrnl.exe | FsRtlReleaseFile + 0x2c7 | 0xfffff8007572b167 | C:\Windows\system32\ntoskrnl.exe |
| K 4 | ntoskrnl.exe | NtQueryInformationThread + 0xb37 | 0xfffff8007572ae17 | C:\Windows\system32\ntoskrnl.exe |
| K 5 | ntoskrnl.exe | FsRtlReleaseFile + 0x373 | 0xfffff8007572b213 | C:\Windows\system32\ntoskrnl.exe |
| K 6 | ntoskrnl.exe | NtCreateSection + 0xce3 | 0xfffff8007572bf63 | C:\Windows\system32\ntoskrnl.exe |
| K 7 | ntoskrnl.exe | NtCreateSection + 0x47f | 0xfffff8007572b6ff | C:\Windows\system32\ntoskrnl.exe |
| K 8 | ntoskrnl.exe | NtCreateSection + 0x258 | 0xfffff8007572b4d8 | C:\Windows\system32\ntoskrnl.exe |
| K 9 | ntoskrnl.exe | NtCreateSection + 0x54 | 0xfffff8007572b2d4 | C:\Windows\system32\ntoskrnl.exe |
| K 10 | ntoskrnl.exe | setjmpex + 0x7825 | 0xfffff80075272785 | C:\Windows\system32\ntoskrnl.exe |
| U 11 | ntdll.dll | NtCreateSection + 0x14 | 0x7ff93c14ffb4 | C:\Windows\SYSTEM32\ntdll.dll |
| U 12 | ntdll.dll | RtlImageNtHeader + 0x382 | 0x7ff93c0f1e72 | C:\Windows\SYSTEM32\ntdll.dll |
| U 13 | ntdll.dll | RtlDosPathNameToNtPathName_U_WithStatus + 0x5e8 | 0x7ff93c0ee2e8 | C:\Windows\SYSTEM32\ntdll.dll |
| U 14 | ntdll.dll | RtlDosPathNameToNtPathName_U_WithStatus + 0x340 | 0x7ff93c0ee040 | C:\Windows\SYSTEM32\ntdll.dll |
| U 15 | ntdll.dll | RtlAnsiStringToUnicodeString + 0x646 | 0x7ff93c0f92b6 | C:\Windows\SYSTEM32\ntdll.dll |
| U 16 | ntdll.dll | RtlCreateUnicodeStringFromAsciiz + 0xe8 | 0x7ff93c0f6408 | C:\Windows\SYSTEM32\ntdll.dll |
| U 17 | ntdll.dll | LdrLoadDll + 0xe4 | 0x7ff93c0f58b4 | C:\Windows\SYSTEM32\ntdll.dll |
| U 18 | KERNELBASE.dll | LoadLibraryExW + 0x161 | 0x7ff93819ee41 | C:\Windows\System32\KERNELBASE.dll |
| U 19 | WS2_32.dll | WSAEnumNameSpaceProvidersW + 0xae7 | 0x7ff93a21f507 | C:\Windows\System32\WS2_32.dll |
| U 20 | WS2_32.dll | WSAEnumNameSpaceProvidersW + 0x952 | 0x7ff93a21f372 | C:\Windows\System32\WS2_32.dll |
| U 21 | WS2_32.dll | WSAEnumNameSpaceProvidersW + 0x6ae | 0x7ff93a21f0ce | C:\Windows\System32\WS2_32.dll |
| U 22 | WS2_32.dll | WSALookupServiceBeginW + 0x31b | 0x7ff93a217c5b | C:\Windows\System32\WS2_32.dll |
| U 23 | WS2_32.dll | WSALookupServiceBeginW + 0x11f | 0x7ff93a217a5f | C:\Windows\System32\WS2_32.dll |
| U 24 | WS2_32.dll | WSALookupServiceBeginA + 0x98 | 0x7ff93a23f9e8 | C:\Windows\System32\WS2_32.dll |
| U 25 | WS2_32.dll | getprotobynumber + 0x43b | 0x7ff93a237f6b | C:\Windows\System32\WS2_32.dll |
| U 26 | WS2_32.dll | gethostbyname + 0x10b | 0x7ff93a238e9b | C:\Windows\System32\WS2_32.dll |

a. Has multiple calls to WS2_32.dll which indicates that the malware is configuring networking capabilities.

## Step 4: Host/Network Based Indicators

Step (4) Host- and network-based indicators of compromise that can be used to determine whether the malware is present.

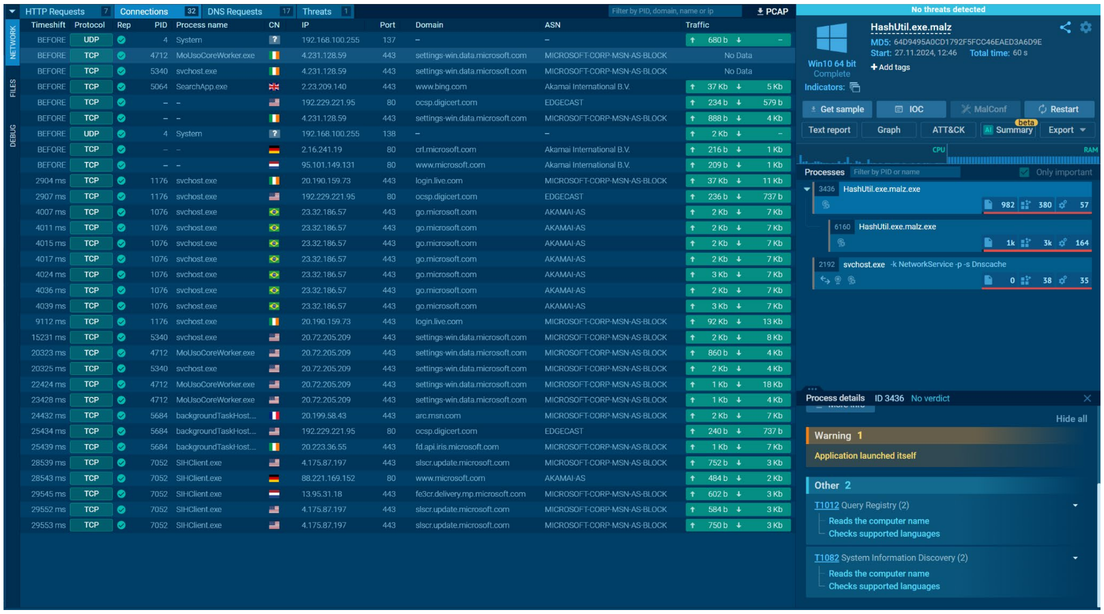
1. Any.Run
    a. HTTP Connections
        i. 

| Timeshift | Headers | Rep | PID | Process name | CN | URL | Content |
|---|---|---|---|---|---|---|---|
| BEFORE | GET 200: OK | | – | – | | http://ocsp.digicert.com/MFEwTzBNMEswSTAJBgUrDgMCGgUABBTrjrydRyt%2BA... | 313 b binary |
| BEFORE | GET 200: OK | | – | – | | http://crl.microsoft.com/pki/crl/products/MicRooCerAut2011_2011_03_22.crl | 1 Kb binary |
| BEFORE | GET 200: OK | | – | – | | http://www.microsoft.com/pkiops/crl/MicSecSerCA2011_2011-10-18.crl | 973 b binary |
| 2907 ms | GET 200: OK | | 1176 | svchost.exe | | http://ocsp.digicert.com/MFEwTzBNMEswSTAJBgUrDgMCGgUABBSAUQYBMq2a... | 471 b binary |
| 25435 ms | GET 200: OK | | 5684 | backgroundTaskHost.... | | http://ocsp.digicert.com/MFEwTzBNMEswSTAJBgUrDgMCGgUABBQ50otx%2Fh0... | 471 b binary |
| 28544 ms | GET 200: OK | | 7052 | SIHClient.exe | | http://www.microsoft.com/pkiops/crl/Microsoft%20ECC%20Product%20Root%20... | 418 b binary |
| 28545 ms | GET 200: OK | | 7052 | SIHClient.exe | | http://www.microsoft.com/pkiops/crl/Microsoft%20ECC%20Update%20Secure%2... | 408 b binary |



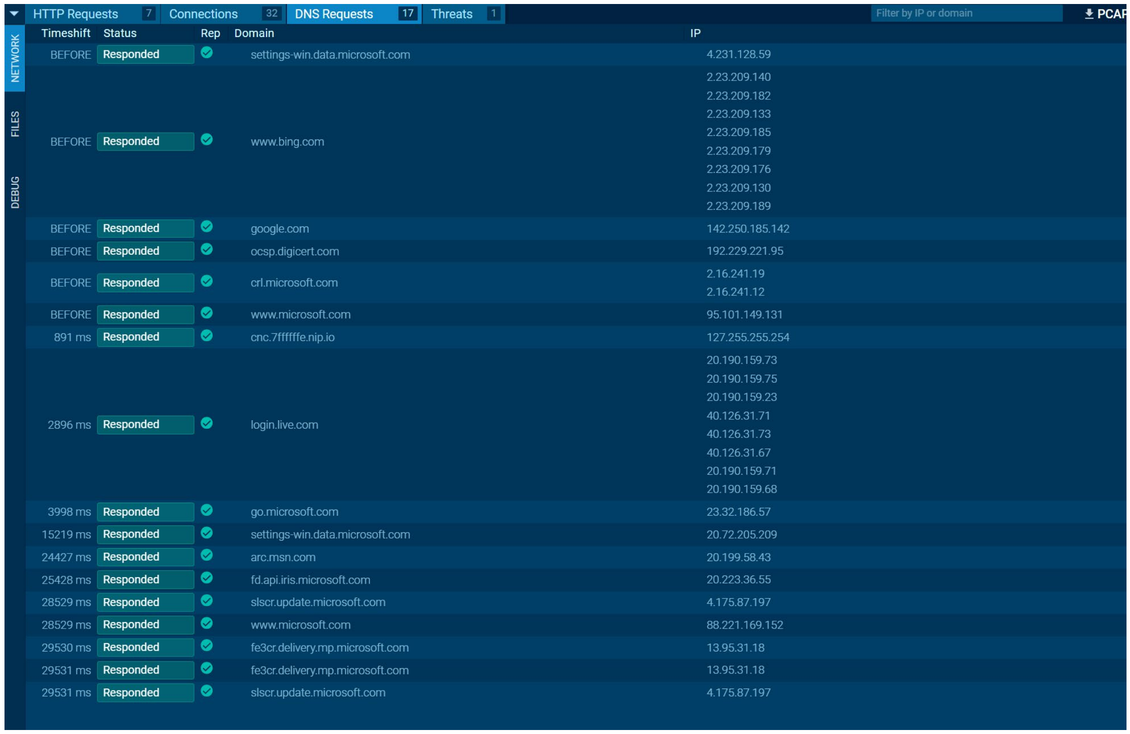
            1. Based on the host-based indicators and observed network activity, the malware shows significant signs of system compromise and unauthorized activity. Multiple HTTP requests were identified, primarily to certificate revocation lists (CRLs) and online certificate status protocol (OCSP) servers, which may suggest that the malware attempts to blend its traffic with legitimate system updates or certificate validation processes. Processes such as svchost.exe, backgroundTaskHost.exe, and SIHClient.exe are associated with these requests, which could either be legitimate Windows components exploited by the malware or masquerading processes injected with malicious code. Binary content of varying sizes was transmitted, indicating potential exfiltration or data staging activities. The observed DNS queries and PIDs align with a pattern of stealthy operation, where legitimate network traffic is leveraged to obfuscate malicious intent. This malware's use of legitimate system processes and network endpoints underscores its sophistication and poses challenges in detection and mitigation. Further analysis, including PCAP and system memory inspection, would clarify its objectives and confirm whether lateral movement or persistence mechanisms are employed.
    b. Connections

i.




1. The malware displays extensive network activity involving frequent connections to various Microsoft and Akamai domains, likely to camouflage its operations among legitimate Windows processes. HTTP and DNS requests reveal interactions withS services such as settings-win.data.microsoft.com and ocsp.digicert.com, suggesting attempts to validate certificates or mimic legitimate network traffic. Key processes implicated include svchost.exe, backgroundTaskHost.exe, and MoUsoCoreWorker.exe, with connections predominantly over ports 443 (HTTPS) and 80 (HTTP). Hostnames and IP addresses correspond to known trusted endpoints, indicating that the malware may exploit legitimate services to evade detection. Packet sizes vary from small binary fragments to larger data transfers, potentially indicating staged data exfiltration. The persistent reuse of core Windows processes and legitimate infrastructure underscores the malware's sophistication, emphasizing the need for advanced endpoint monitoring to identify anomalies in process behavior and traffic patterns.

c. DNS Requests

i. 

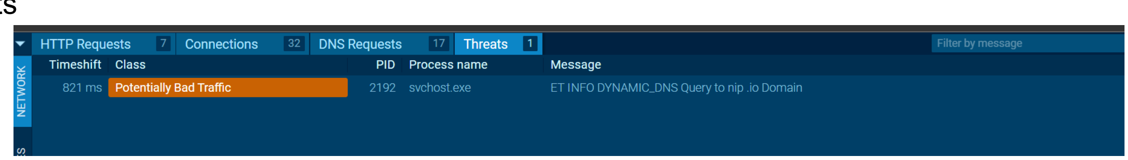
HTTP Requests 7 | Connections 32 | DNS Requests 17 | Threats 1 | Filter by IP or domain | PCAP

NETWORK | FILES | DEBUG

| Timeshift | Status | Rep | Domain | IP |
|---|---|---|---|---|
| BEFORE | Responded | | settings-win.data.microsoft.com | 4.231.128.59 |
| BEFORE | Responded | | www.bing.com | 2.23.209.140<br>2.23.209.182<br>2.23.209.133<br>2.23.209.185<br>2.23.209.179<br>2.23.209.176<br>2.23.209.130<br>2.23.209.189 |
| BEFORE | Responded | | google.com | 142.250.185.142 |
| BEFORE | Responded | | ocsp.digicert.com | 192.229.221.95 |
| BEFORE | Responded | | crl.microsoft.com | 2.16.241.19<br>2.16.241.12 |
| BEFORE | Responded | | www.microsoft.com | 95.101.149.131 |
| 891 ms | Responded | | cnc.7ffffffe.nip.io | 127.255.255.254 |
| 2896 ms | Responded | | login.live.com | 20.190.159.73<br>20.190.159.75<br>20.190.159.23<br>40.126.31.71<br>40.126.31.73<br>40.126.31.67<br>20.190.159.71<br>20.190.159.68 |
| 3998 ms | Responded | | go.microsoft.com | 23.32.186.57 |
| 15219 ms | Responded | | settings-win.data.microsoft.com | 20.72.205.209 |
| 24427 ms | Responded | | arc.msn.com | 20.199.58.43 |
| 25428 ms | Responded | | fd.api.iris.microsoft.com | 20.223.36.55 |
| 28529 ms | Responded | | slscr.update.microsoft.com | 4.175.87.197 |
| 28529 ms | Responded | | www.microsoft.com | 88.221.169.152 |
| 29530 ms | Responded | | fe3cr.delivery.mp.microsoft.com | 13.95.31.18 |
| 29531 ms | Responded | | fe3cr.delivery.mp.microsoft.com | 13.95.31.18 |
| 29531 ms | Responded | | slscr.update.microsoft.com | 4.175.87.197 |

1. The malware exhibits extensive HTTP activity, frequently connecting to domains such as settings-win.data.microsoft.com, ocsp.digicert.com, and login.live.com, alongside connections to lesser-known addresses like cnc.7ffffffe.nip.Io. The repeated responses from legitimate endpoints (e.g., Microsoft, Akamai, Google) suggest that the malware employs techniques to blend its traffic within normal system behavior, possibly leveraging trusted services to avoid detection. Specific IP addresses, such as 20.190.159.73 and 4.231.128.59, reveal consistent communication patterns that may signify command-and-control (C2) functionality or data exfiltration. Connection timestamps show persistence across intervals, reflecting deliberate, periodic activities. Furthermore, DNS requests to recognizable domains further reinforce an effort to mimic standard OS processes while maintaining covert communications with potential C2 nodes. These behaviors highlight the need for anomaly-based monitoring and correlation of endpoint activities with network patterns to isolate malicious intents effectively.

d. Threats

i. 

HTTP Requests 7 | Connections 32 | DNS Requests 17 | Threats 1 | Filter by message

NETWORK

| Timeshift | Class | PID | Process name | Message |
|---|---|---|---|---|
| 821 ms | Potentially Bad Traffic | 2192 | svchost.exe | ET INFO DYNAMIC_DNS Query to nip .io Domain |

1. The malware demonstrates potentially malicious behavior, including DNS queries to dynamic DNS domains such as nip.io, which are commonly associated with command-and-control (C2)

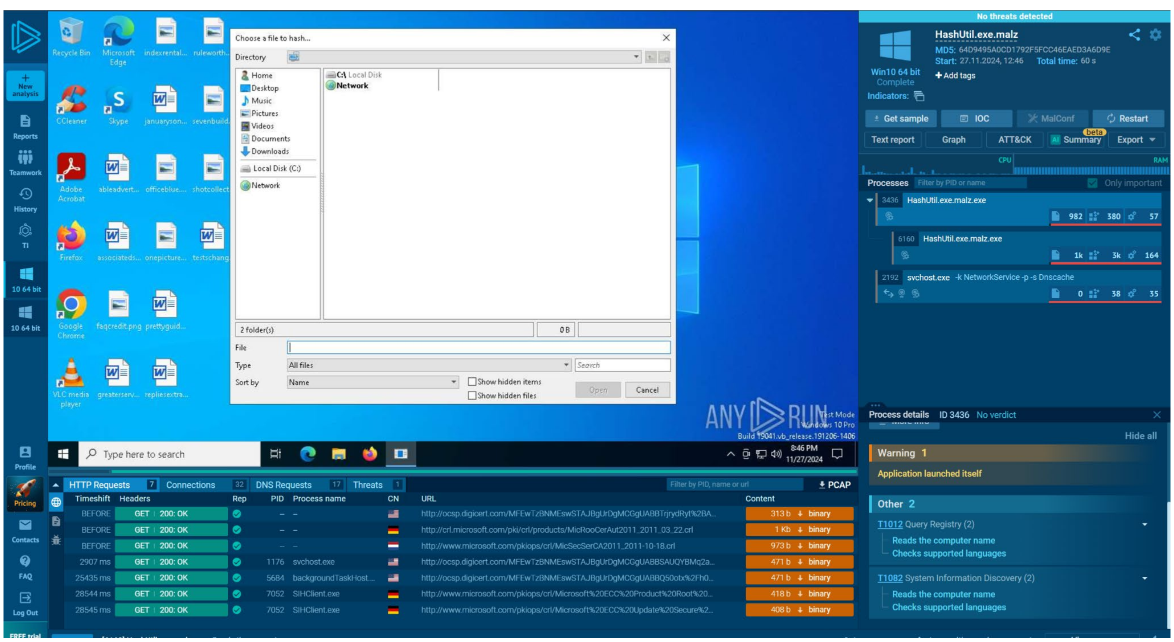
infrastructure or obfuscation tactics. The process svchost.exe (PID 2192) appears to be exploited, reflecting a known pattern of abuse where legitimate Windows processes are leveraged to execute malicious operations. The rapid HTTP request-response cycles, as short as 821 ms, indicate frequent communication attempts that could signify data exfiltration or real-time command execution. The use of dynamic DNS and potential C2 interactions underscore the malware's sophistication, necessitating the correlation of network activity with host-based indicators to effectively detect and mitigate its presence.

---

## Step 5: Attacker's Actions

Step (5) Actions that you would expect an attacker to perform once the system has been compromised

1. Any.Run

   a. 


      i. An application opens from running the program and we have several host-based identifiers to analyze, but there are no file modifications.

2. The malware appears to be attempting to establish a connection with the attacker's command-and-control (C2) server. By gaining access to terminal commands and the system kernel, the attacker effectively obtains full control over the compromised machine. This level of access allows them to establish a reverse shell, execute commands, and download or deploy additional malicious payloads onto the host system. Attackers might try to move laterally through the network and use the compromised system to access shared network resources and attack other machines on the network.

---

### Step 6: Undoing the Damage

Step (6) If a host has been compromised, how to undo the damage.

1. Immediately isolate the compromised machine from all networks to prevent further spread of the malware and unauthorized data transmission.. Run a full scan using a reputable antivirus software (such as windows defender) and boot the system into Safe Mode to minimize potential interference from the malware.
2. Use tools like RegEdit to edit the windows registry and revert malicious registry changes. Using window services such as task scheduler to review all the tasks on the system and remove the malicious tasks scheduled by the attacker.
3. If nothing is working, the final and definitive step, completely wipe the compromised system using a secure disk wiping tool (e.g., DBAN or built-in OS recovery options) to ensure no remnants of the malware remain. Then reinstall the operating system from a trusted source, ensuring that the installation media is clean, correct, and up-to-date. Reconfigure the system's security settings, including firewalls, antivirus, and patch

updates, before reconnecting it to the network. Goes without saying but never run the malware again and learn from previous mistakes.

---

## Appendix (Team Contribution)

Phu Lam: discord setup, documentation template, ghidra, host/network indicators,

Terry Ma:

Wayne Muse: Surface level functionality, set up, PEStudio, ApatesDNS, Steps 1,5,6

Yazid Soulong: Dynamic Analysis - Process Explorer, Process Monitor

Luis Valle-Arellanes: adding to process explorer and process monitor