CS4238 Computer Security Practice

AY2022/23 Semester 2

Course Project: Malware Analysis Report

Group 22

Ang Yong Ming	A0216990X
Ibrahim Sharul	A0225864X
Teo Chuan Kai	A0174217H

1. Basic Static Analysis **3**
1.1 Virustotal 3
1.2 Imports 5
1.3 Packers 7
1.4 Program Metadata 8
1.5 Strings 11
2. Basic Dynamic Analysis **13**
2.1 Program Behaviour 13
2.2 Process Activity 15
2.3 Registry Changes 25
2.4 Network Activity 27
3. Advanced Static Analysis **28**
3.1 Disassembling and Analysing Main Executable in IDA(Interactive Disassembler) 28
Program Entrypoint & Setup Functions 28
WinMain Function 31
Overview of Functionality 44
3.2 Disassembling and Analysing adobe.exe in IDA 45
Program Entrypoint and Function Calls 45
Overview of Functionality 49
3.3 Disassembling and Analysing MSVCR110.dll in IDA 50
Exported Function Within DLL 50
Overview of Functionality 59
4. Advanced Dynamic Analysis **60**
4.1 Debugging Main Executable in OllyDbg 60
4.2 Debugging adobe.exe and MSVCR110.dll 70
4.3 Debugging Shellcode Contained in MSVCR110.dat 75
5. Other Observations and Inferences **77**
6. Conclusion **78**

1. Basic Static Analysis

1.1 Virustotal

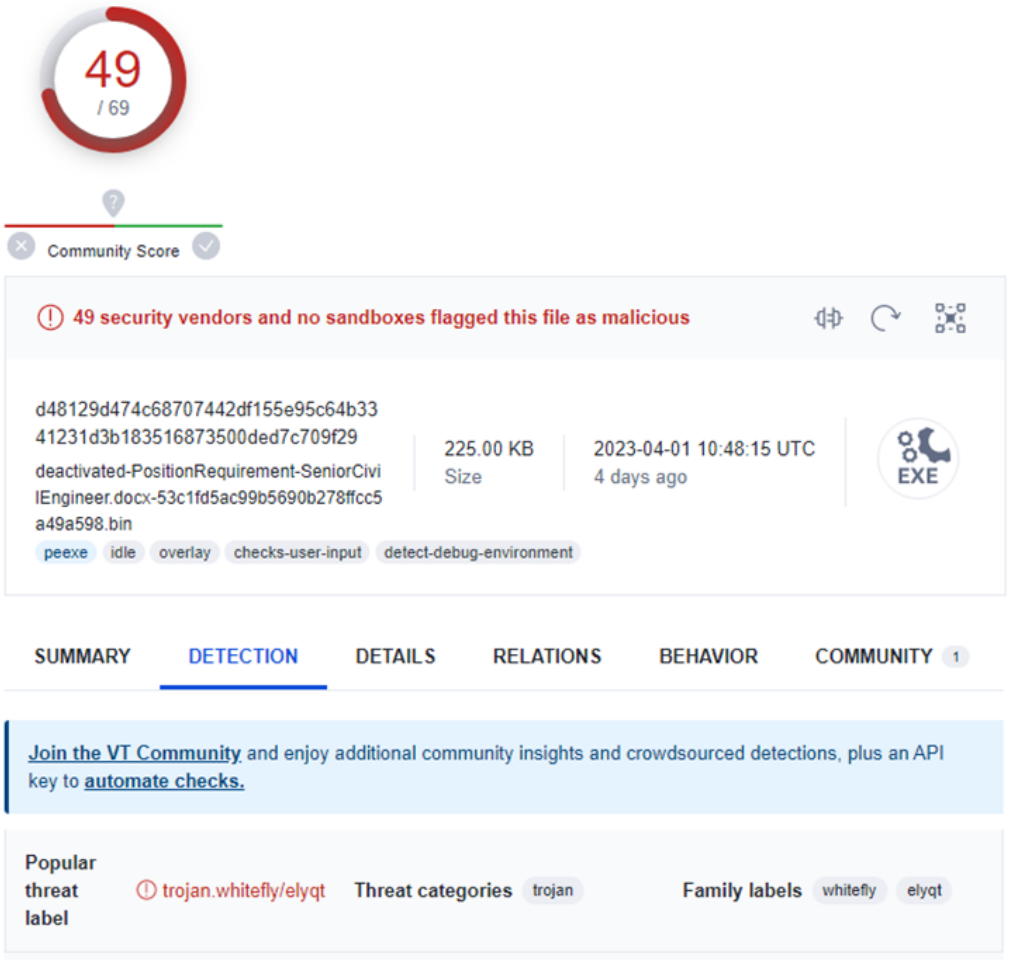


Fig 1.1.1: VirusTotal for Inactive Version of Malware

Uploading the inactive malware sample to VirusTotal shows us that it is likely a trojan originating from the threat actor, WhiteFly, and it has been tagged as:

- peexe
- idle
- overlay
- checks-user-input
- detect-debug-environment

While this is non-exhaustive and there may be false positives here, this hints to us that a potential function of the virus is to trick users into inputting user credentials on an overlay screen that replicates a legitimate login page. The credentials will then be forwarded to the hackers' C2

server. Additionally, the malware is likely to be fitted with an anti-debugger to check that the program is not running under a debugger.

Now, if we were to instead upload the active version of the malware to VirusTotal, we can once again see that it is flagged as malicious by the majority of the security vendors and sandbox. In particular, VirusTotal has identified it as a PlugX malware, which is a remote access trojan believed to have been used in the SingHealth data breach.

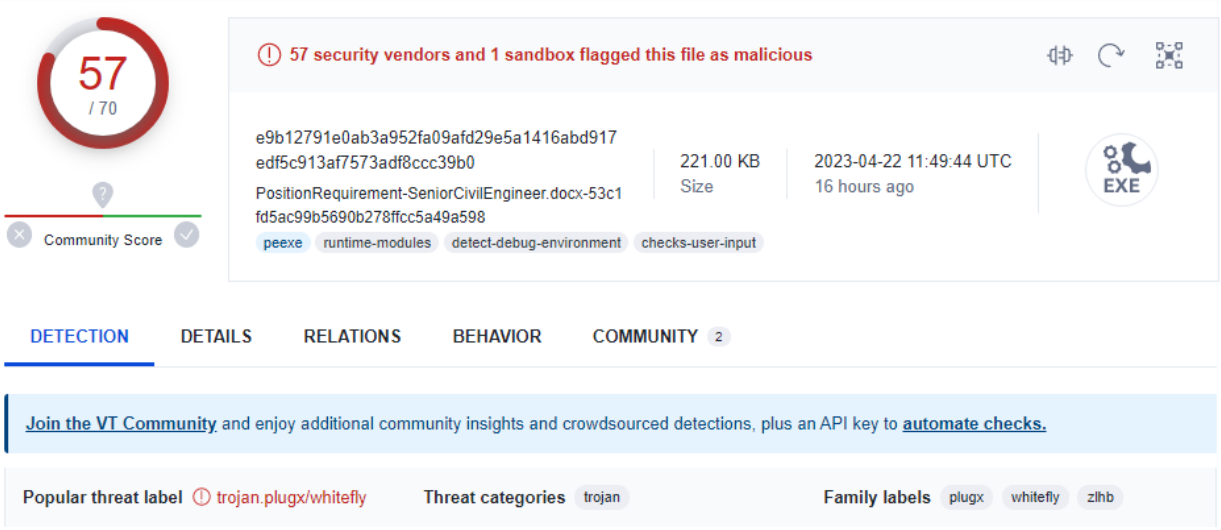


Fig 1.1.2: VirusTotal for Active Version of Malware

Looking at other sections in VirusTotal, we can also glean various notable observations, such as contacted domains, which may serve as potential network-based indicators of compromise, and dropped files, specifically, MSVCR110.dll, which could hint at potential DLL side-loading:

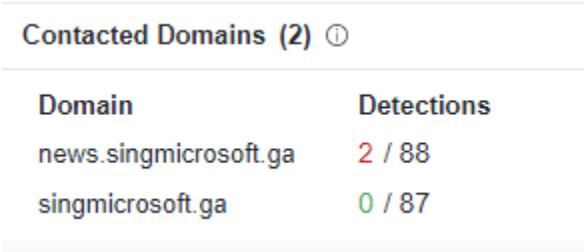


Fig 1.1.3: VirusTotal Contacted Domains for Active Version of Malware

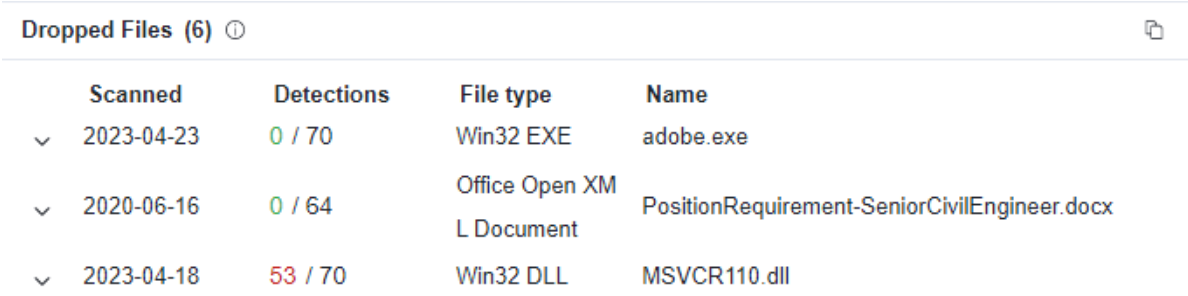


Fig 1.1.4: VirusTotal Dropped Files for Active Version of Malware

1.2 Imports

Next, we can analyse the imports using PEView:

Kernel32.dll	Shell32.dll	user32.dll
CreateFile	ShellExecuteA	wsprintfW
WriteFile	ShellExecuteW	wsprintfA
GetCurrentProcess		
TerminateProcess		
GetProcAddress		
GetCurrentProcessId		
ExitProcess		
GetCurrentThreadId		
EnterCriticalSection		
DeleteCriticalSection		
LeaveCriticalSection		
InitializeCriticalSectionAndSpinCount		
InterlockedIncrement		
InterlockedDecrement		
GetCommandLineA		
GetStartupInfoA		
IsDebuggerPresent		

Intuitively, the presence of the ShellExecute imports points to the malware executing programmes and performing certain operations.

More light can be shed under Kernel32.dll, where we can presume that the functions called by the malware can potentially manipulate files (CreateFile, WriteFile) and processes (GetCurrentProcess, TerminateProcess, GetProcAddress, GetCurrentProcessId, ExitProcess), and in particular, involving one or more multithreaded processes (GetCurrentThreadId). As threads within a single process share data with other threads, synchronisation primitives typically exist to protect the shared data from being accessed and modified simultaneously. With this in mind, some of the malware's imports (DeleteCriticalSection, LeaveCriticalSection,

EnterCriticalSection, InitializeCriticalSectionAndSpinCount, InterlockedIncrement, InterlockedDecrement) suggest that the malware is somewhat able to control which threads are able to access the critical section. Additionally, the IsDebuggerPresent import corroborates the finding from VirusTotal, which hypothesised that the programme is likely fitted with an anti-debugger.

Finally, looking at the imports from user32.dll, the wsprintfW and wsprintfA imports point to the potential for the malware to write formatted data to a buffer.

1.3 Packers

As can be seen from the screenshots below, using both PEiD and running the upx -d unpack command, we can infer that the executable is not packed.

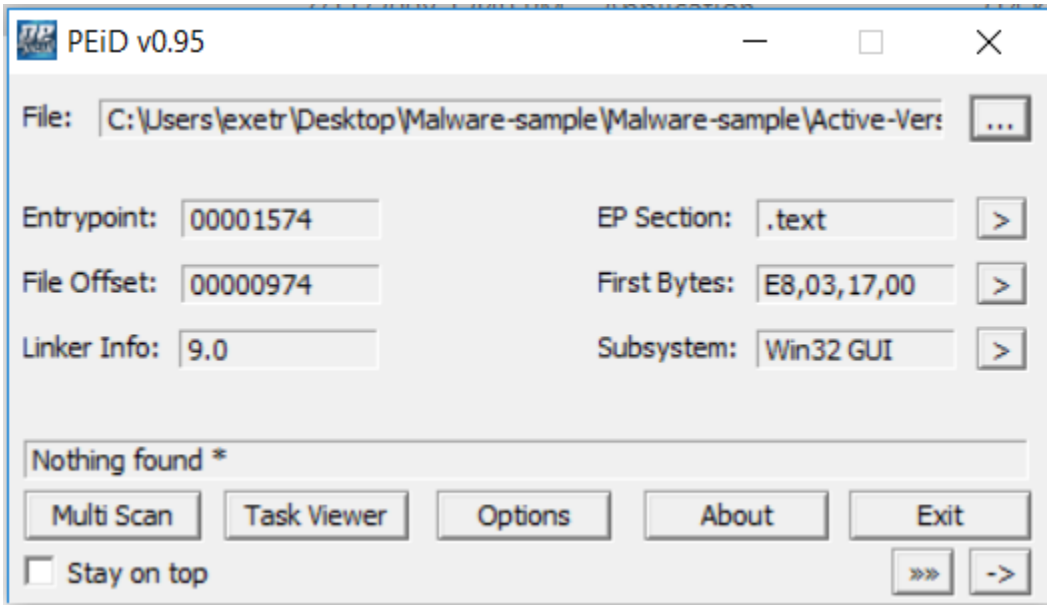


Fig 1.3.1: PEiD Output for Active Version of Malware

Fig 1.3.2: Output from Attempting to Unpack Active Version of Malware

1.4 Program Metadata

By running Get-FileHash, we can obtain the hash of executable to be the following:

Hash Algorithm	Hash
MD5	53C1FD5AC99B5690B278FFCC5A49A598
SHA1	656850FE87EAD292CEB4844C9A003F9FAC354EF6
SHA256	E9B12791E0AB3A952FA09AFD29E5A1416ABD917EDF5C913AF7573ADF8CCC39B

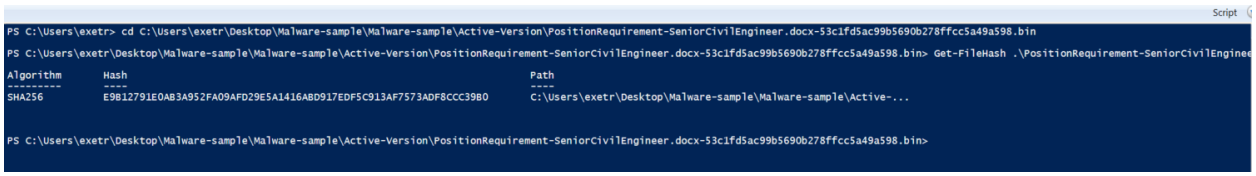


Fig 1.4.1: SHA256 Hash of Active Version of Malware

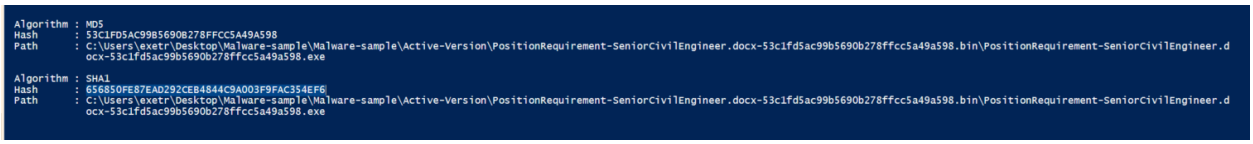


Fig 1.4.2: MD5 and SHA1 Hash of Active Version of Malware

This MD5, SHA1 and SHA256 hashes can serve as an Indicator of Compromise (IOC) that can be used by security softwares to detect if this unrun malware resides on any computer.

We can also run PEView to observe the compile time, which is 16/04/2018. This is a few months before the SingHealth attack was detected, where data was exfiltrated between 27th June and 4th July 2018.

pFile	Data	Description	Value
000000E4	014C	Machine	IMAGE_FILE_MACHINE_I386
000000E6	0005	Number of Sections	
000000E8	5AD41B92	Time Date Stamp	2018/04/16 Mon 03:42:10 UTC
000000EC	00000000	Pointer to Symbol Table	
000000F0	00000000	Number of Symbols	
000000F4	00E0	Size of Optional Header	
000000F6	0102	Characteristics	
		0002	IMAGE_FILE_EXECUTABLE_IMAGE
		0100	IMAGE_FILE_32BIT_MACHINE

Fig 1.4.3: Program Metadata found from PEView

By running ResourceHacker, we can also obtain 3 resources, of which the first 2 relates to an icon depicting the Microsoft Word icon, while the last one is the Manifest.

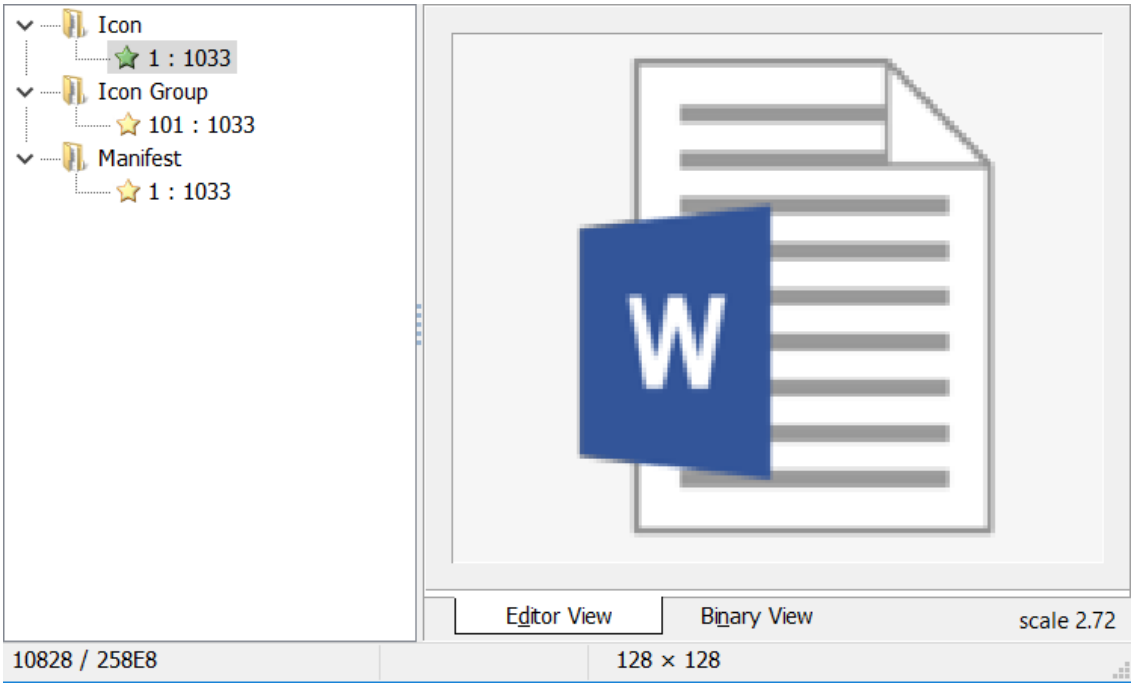


Fig 1.4.4: Resource Hacker Output

The presence of this icon depicts how the illegitimacy of the malware is masked behind having the file displaying a Word Document icon that will be visible in the File Explorer, tricking unsuspecting users who merely see the icon and the seemingly unsuspicious document name titled “PositionRequirement-SeniorCivilEngineer.docx” into failing to notice the .exe extension at the end.

As for the manifest resource file, there does not seem to be anything suspicious going on.

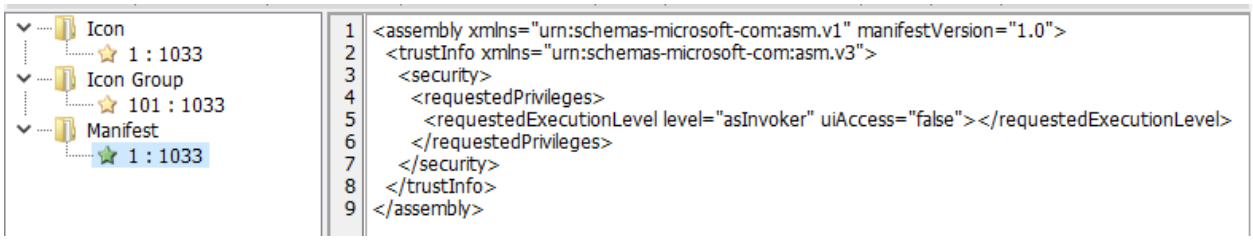


Fig 1.4.5: Resource Hacker Output

1.5 Strings

On the other hand, what is highly suspicious is that there are so few imported DLLs. As such, what we can do is to run the strings command while grepping for dll, and we can then tell from the output that there are many more DLLs involved.

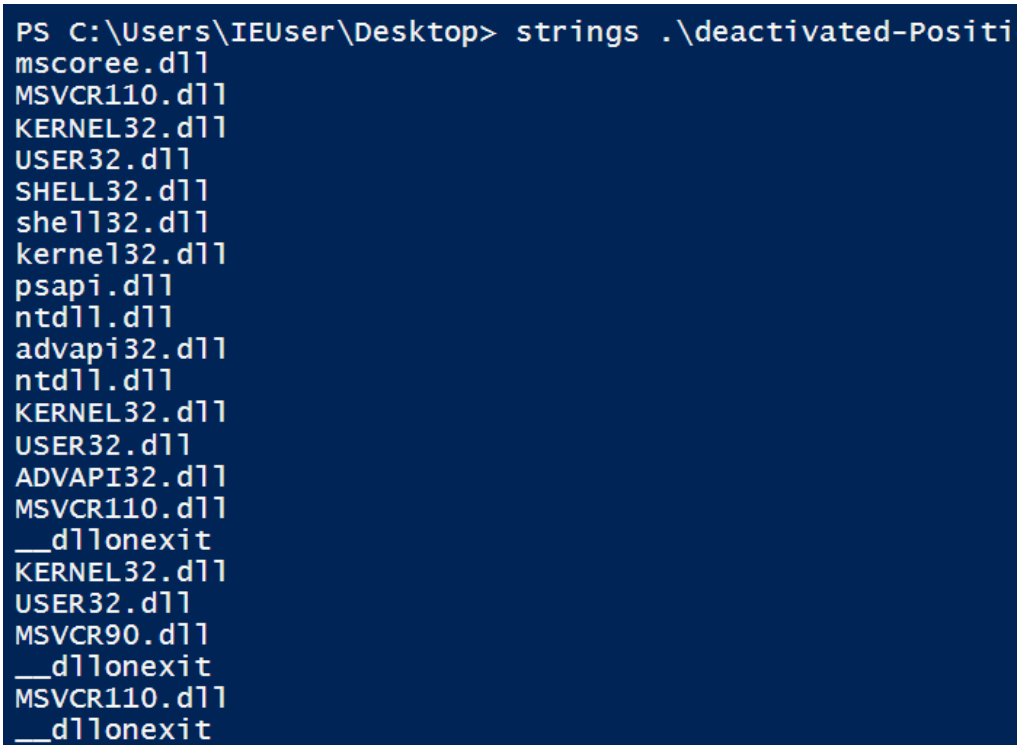


Fig 1.5.1: Strings output when grepping .dll

This suggests the probable involvement of DLL side-loading, which was identified by VirusTotal earlier, where a signed and trusted executable is leveraged as it is less likely to be subjected to rigorous anti-virus checks. When this executable is loaded, the OS will then load the DLL from the nearest directory (which is the side-loaded DLL), and the malicious payload is typically embedded in that DLL.

As such, what we can do is to run the strings command again, but this time, grepping for executable file extensions such as .bat or .exe:

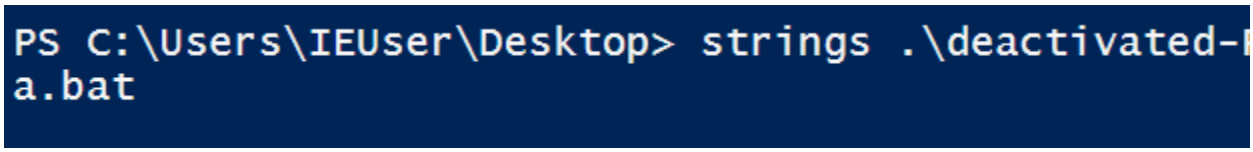


Fig 1.5.2: Strings output when grepping .bat

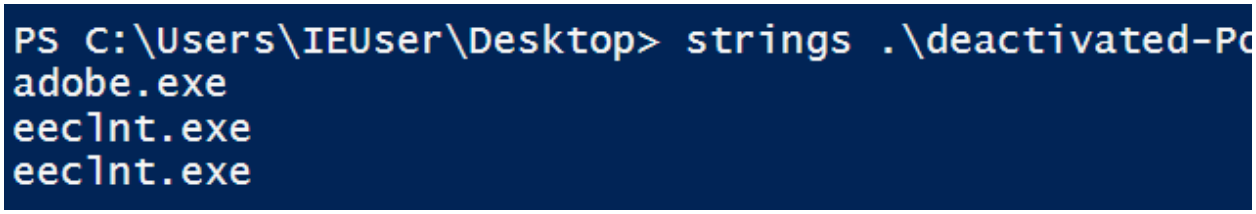


Fig 1.5.3: Strings output when grepping .exe

From these commands, we can identify the following 3 suspicious files, which will form the basis of our Advanced Static Analysis later on:

- a.bat
- adobe.exe
- eeclnt.exe

The presence of these files would also serve as potential host-based indicators of compromise.

2. Basic Dynamic Analysis

2.1 Program Behaviour

When we first run the program, the following document is opened, and this is indeed a legitimate .docx file.

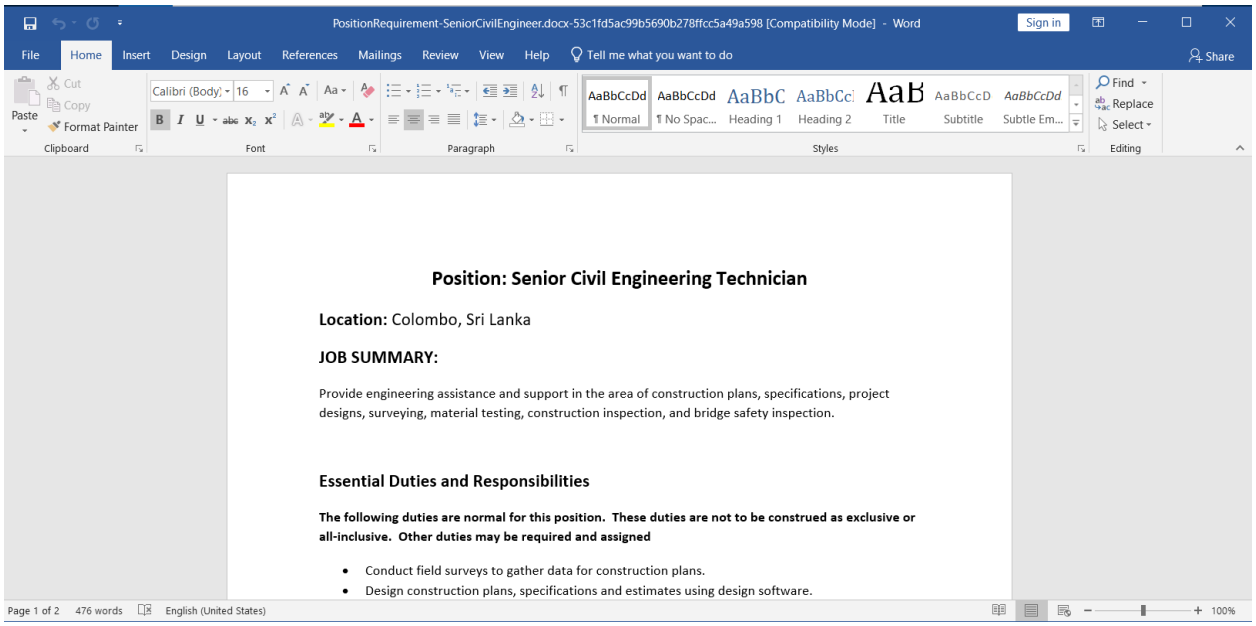


Fig 2.1.1: Screenshot of Opened Document

By observing the filename, we can see that the .exe extension is gone. Upon noticing that the date and time of the file was the date and time of executing the malware, we came to realise that this document that was opened is in fact, a new file with the exact same name as the old file, but without the .exe extension, whereas the old file with the .exe extension has been deleted. With this in mind, the Advanced Static Analysis portion of our report will dig deeper into the suspicious executables as identified earlier to see if any of them was responsible for this.

As shown in the screenshots below, this new document has a length of 13822 bytes.

```
-a----        4/13/2023   7:00 PM          13822 PositionRequirement-SeniorCivilEngine
                                                 er.docx-53c1fd5ac99b5690b278ffcc5a49a
                                                 598
```

Fig 2.1.2: Length of New File

This is in stark contrast with the ‘old’ executable, which has a file length of 226304 bytes.

```
-a----       12/28/2020   6:12 PM         226304 PositionRequirement-SeniorCivilEn
                                                 gineer.docx-53c1fd5ac99b5690b278f
                                                 fcc5a49a598.exe
```

Fig 2.1.3: Length of Original File

And the SHA256 hash of the new file is as follows:

```
PS C:\Users\IEUser\Desktop> Get-FileHash .\PositionRequirement-SeniorCivilEngineer.dc

Algorithm       Hash                                                                   |
---------       ----
SHA256          B4E630FC970052653436FC447CDC9354F7920E691642276C1D7C3E7F593B164F
```

Fig 2.1.4: Hash of New File

And this has a different hash value from the SHA256 hash of the programme before it was executed.

```
PS C:\Users\IEUser\Desktop> Get-FileHash .\PositionRequirement-SeniorCivilEngineer

Algorithm       Hash
---------       ----
SHA256          E9B12791E0AB3A952FA09AFD29E5A1416ABD917EDF5C913AF7573ADF8CCC39B0
```

Fig 2.1.5: Hash of Old File

2.2 Process Activity

When running the programme, we can also run Process Explorer simultaneously to analyse the processes initiated by the programme, and the first thing to observe will be the generated process tree, which can be found in this screenshot below.

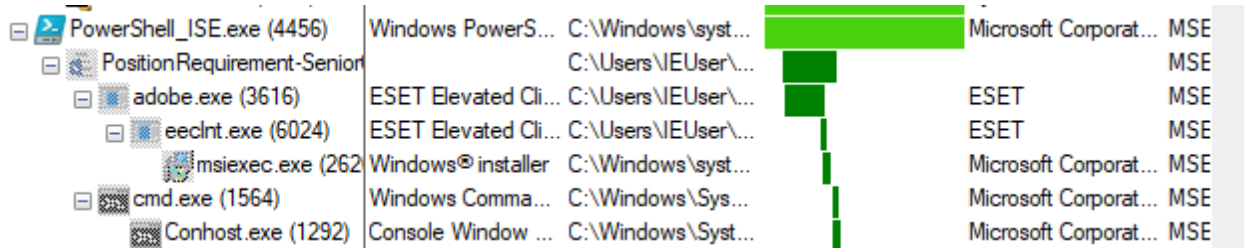


Fig 2.2.1: Process Tree

From a high-level overview, running the program spawns a process adobe.exe, which in turn initiates the process eeclnt.exe, which then triggers msiexec.exe. With this in mind, let us go more in-depth into each process.

When the active version of the malware is first run, it creates multiple threads and loads several DLLs. An interesting observation to note is that it also created and wrote to the file “MSVCR110.dat”. Additionally, as corroborated by the process tree, it creates the process adobe.exe as well as cmd.exe.

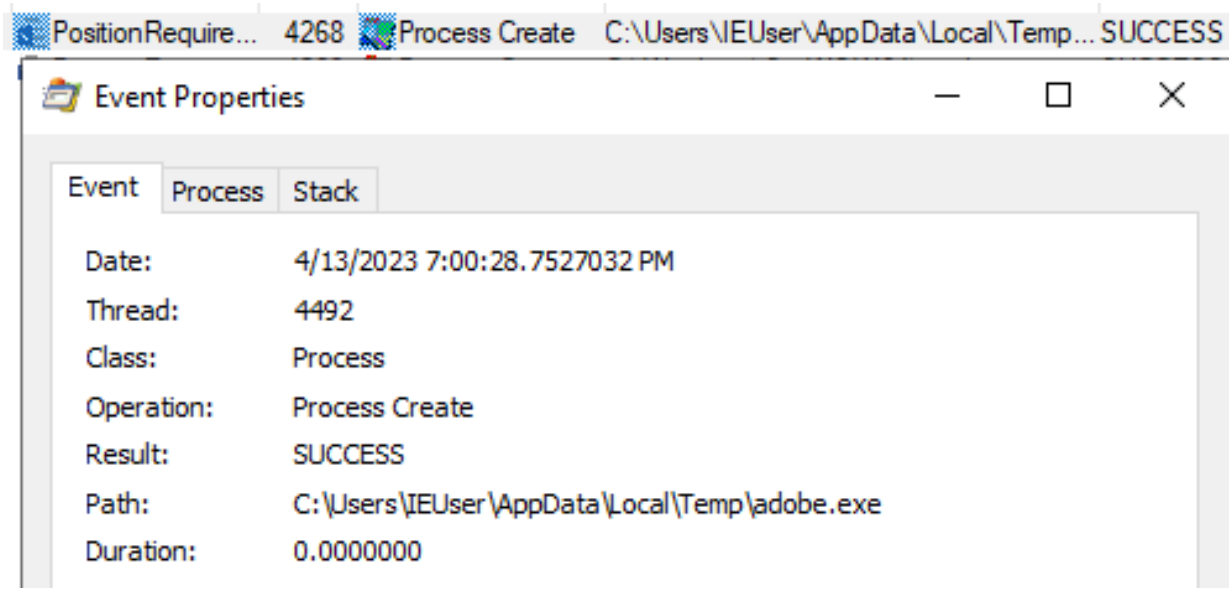


Fig 2.2.2: Spawning adobe.exe

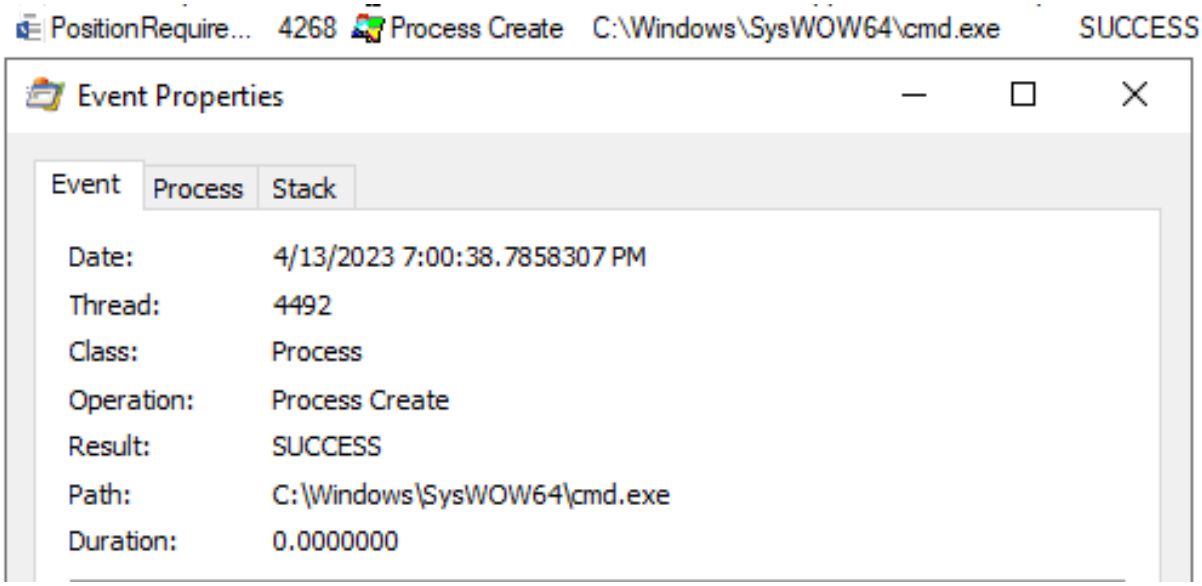


Fig 2.2.3: Spawning cmd.exe

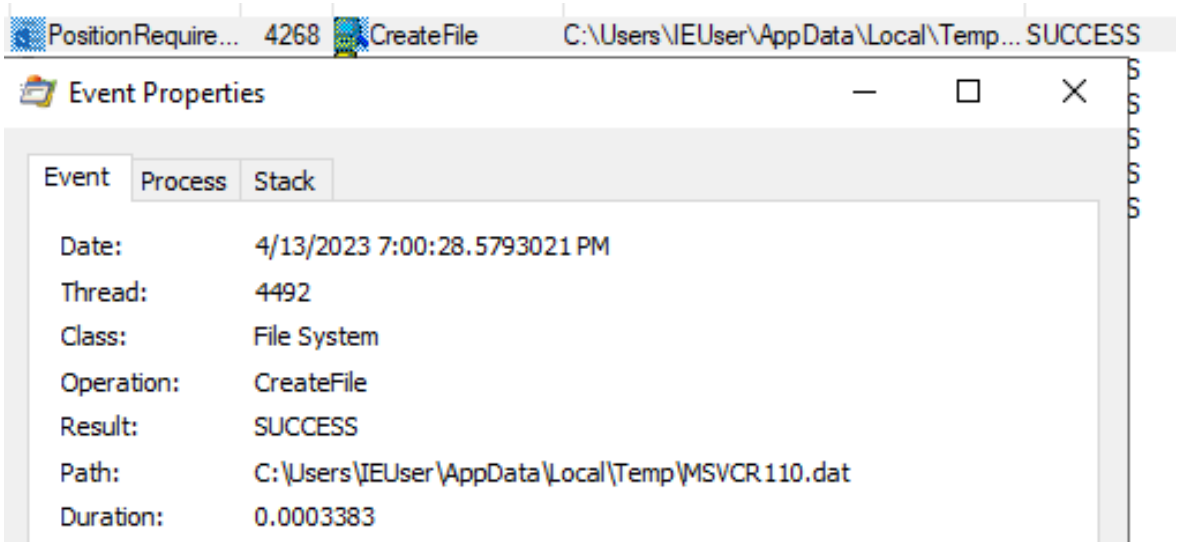


Fig 2.2.4: Manipulating MSVCR110.dat

Now, looking at adobe.exe, it similarly creates multiple threads and also loads several DLLs. In particular, a notable one would be the creation of MSVCR110.dll, which is highly suspicious given that MSVCR110.dll is a legitimate DLL which should be located in \Windows\System32, but here, it is loaded under IEUser\AppData\Local\Temp, which is where the executable is being run from. Hence, this confirms the involvement of DLL side-loading.

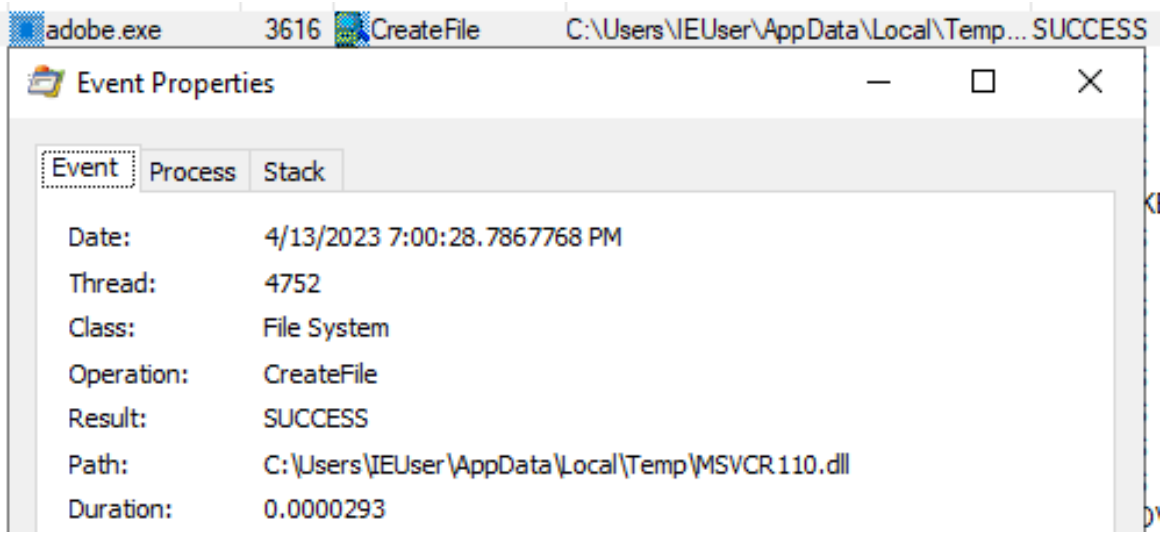


Fig 2.2.5: Creating MSVCR110.dll

Subsequently, adobe.exe launches eeclnt.exe.

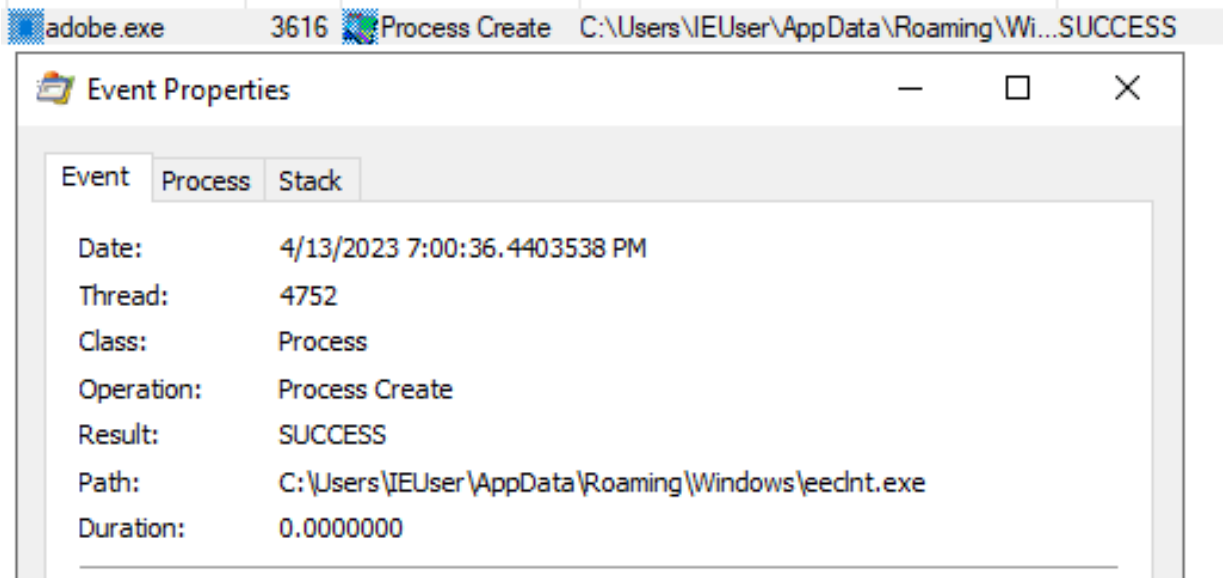


Fig 2.2.6: Spawning eeclnt.exe

Moving on to eelcnt.exe, this is where things start to get interesting. As aforementioned, eeclnt.exe spawns the process msiexeAuthc.exe. But the unusual thing about this is that when eeclnt.exe was first executed, it was fed the command line argument “258”.

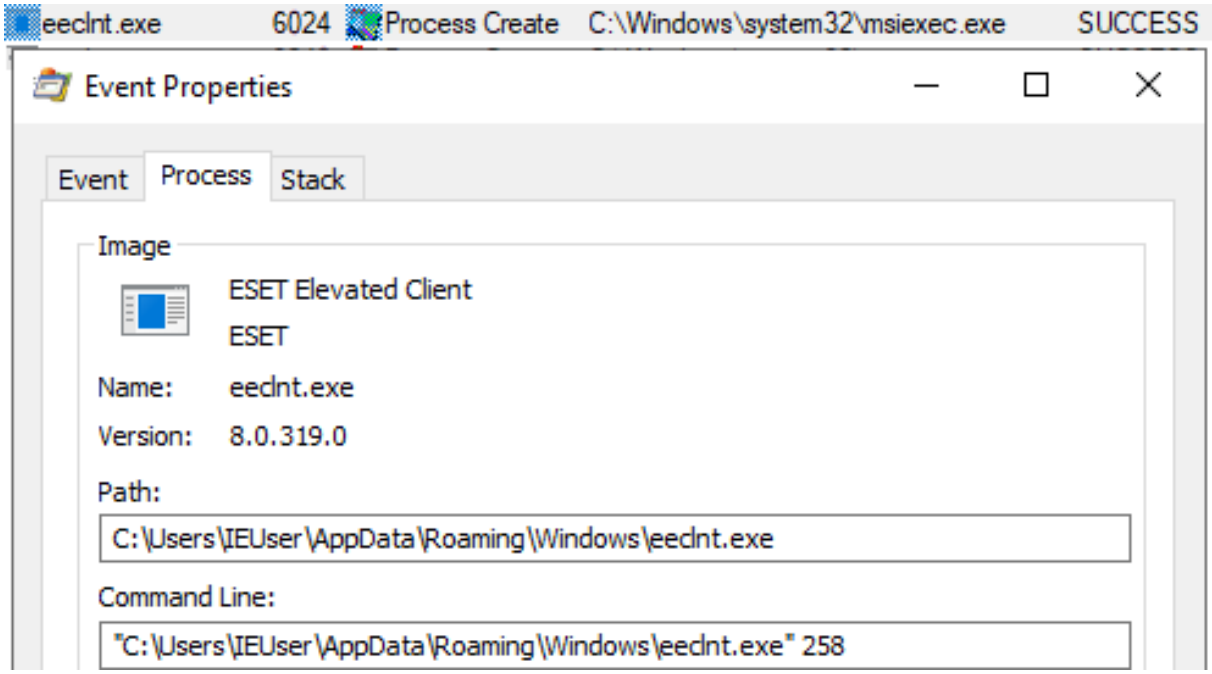


Fig 2.2.7: Running eecInt.exe with 258 as command-line argument

This then creates the process msiexec.exe, with “259” as the command line input.

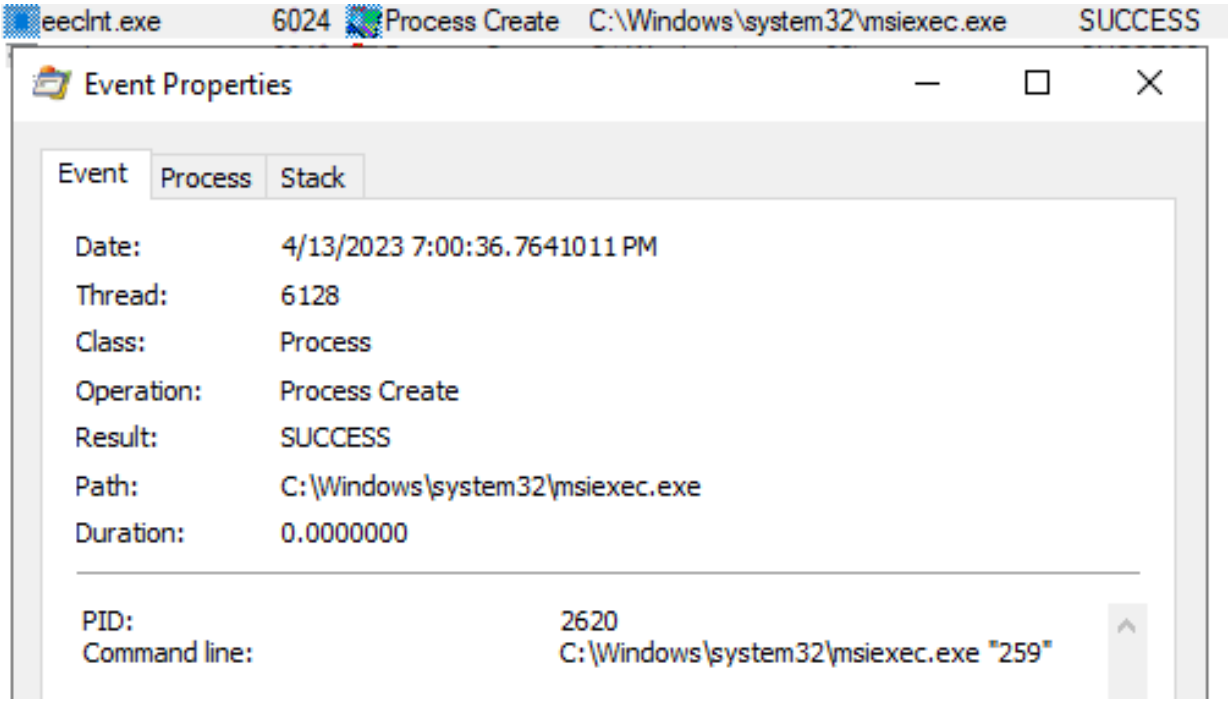


Fig 2.2.8: Running msiexec.exe with 259 as command-line argument

Following this, the process then exits.

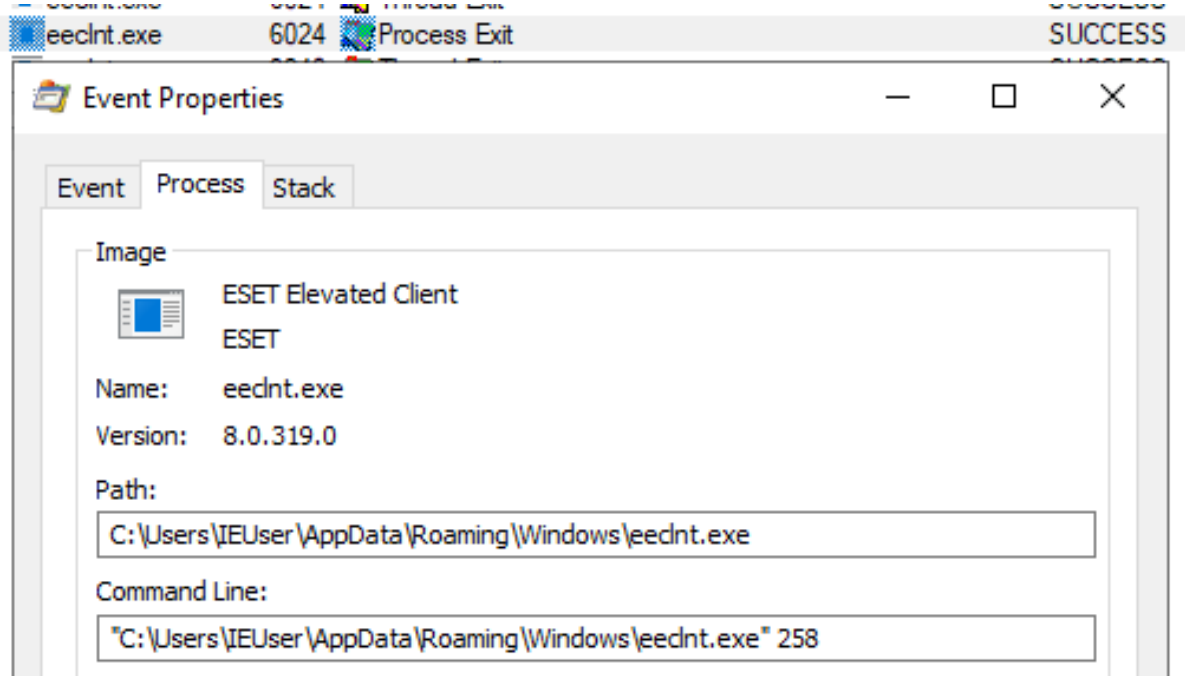


Fig 2.2.9: Process created by running `eeclnt.exe 258` exits

But then, eeclnt.exe is run again, the only difference being this time, the command line argument “260” is being fed as input, instead of the earlier “258”.

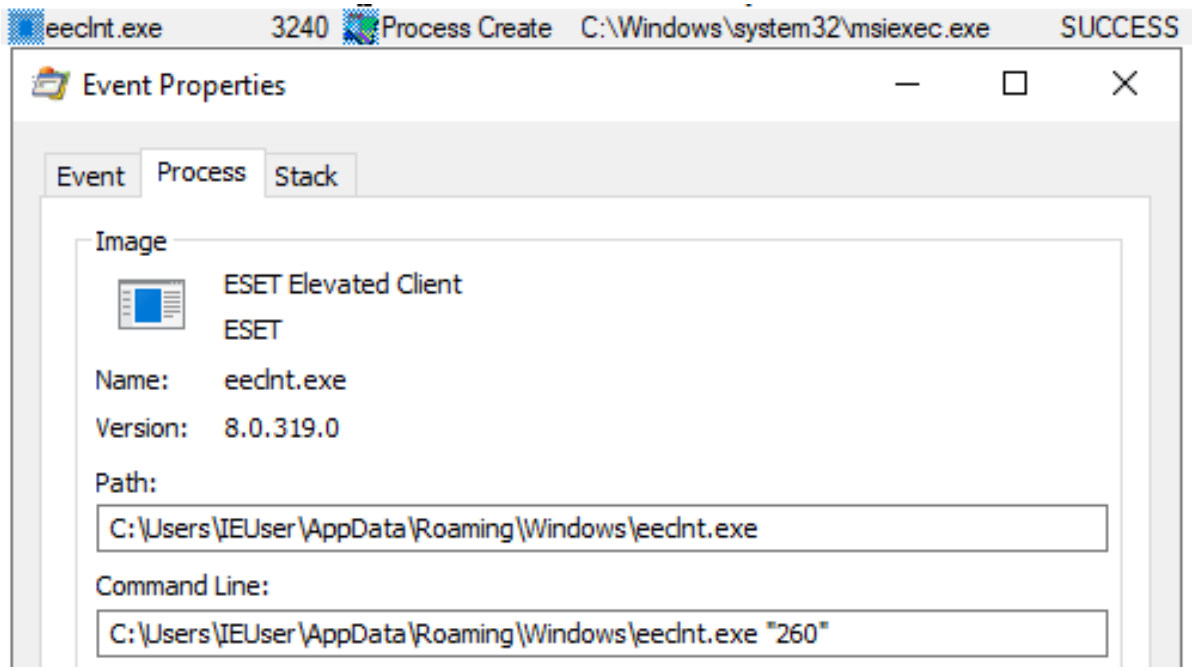


Fig 2.2.10: Running eeclnt.exe with 260 as command-line argument

Once again, this spawns the process msiexec.exe, with the command line argument being “261” this time instead of the earlier “259”.

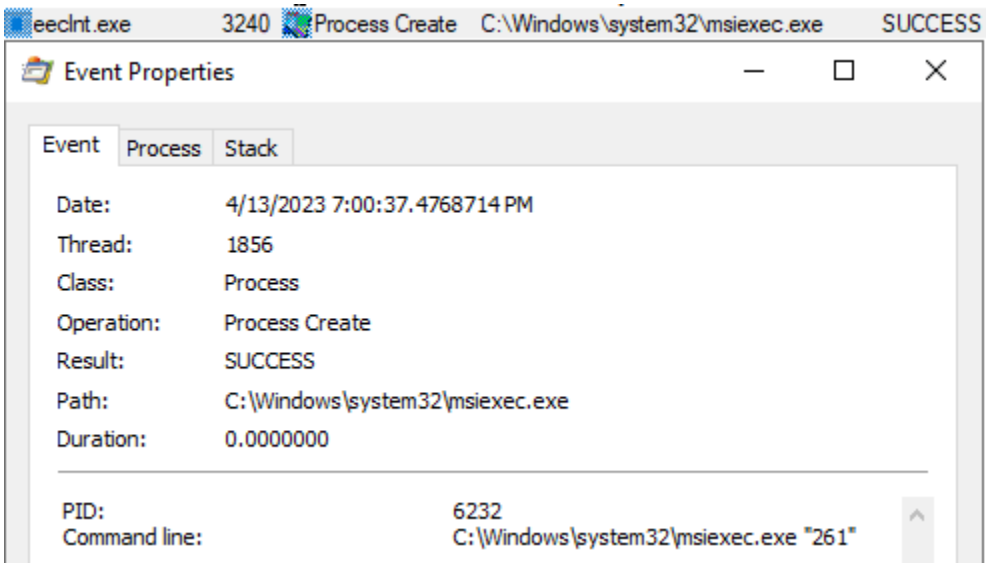


Fig 2.2.11: Running msiexec.exe with 261 as command-line argument

While not 100% sure why, our group initially suspected that this sequence of processes being called works to create persistence, and to confirm our hunch, we decided to corroborate with what can be found on RegShot, which will be elaborated on later.

While msiexec.exe was running, WerFault.exe, which is Windows' error reporting process, was triggered.

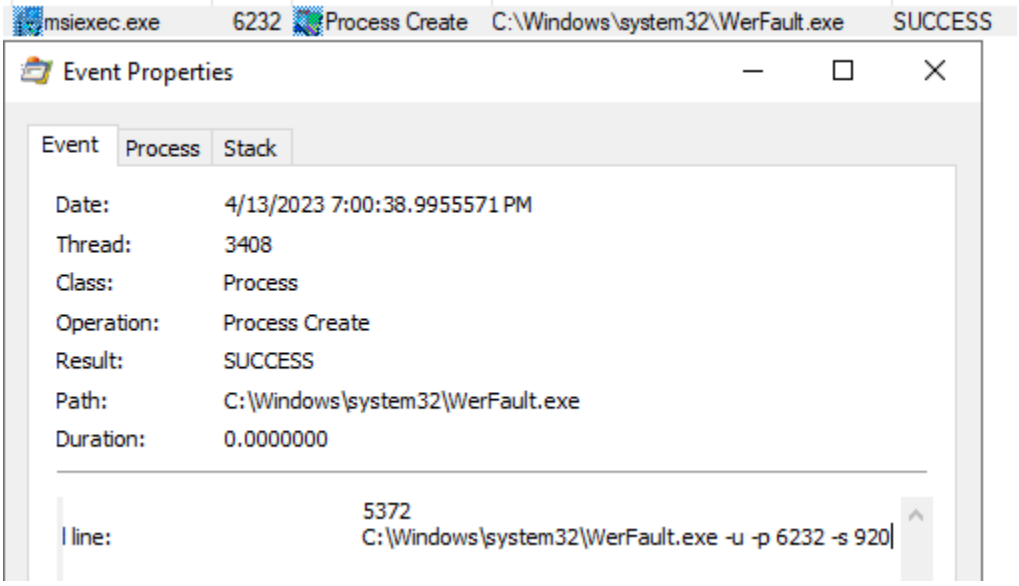


Fig 2.2.12: WerFault.exe triggered

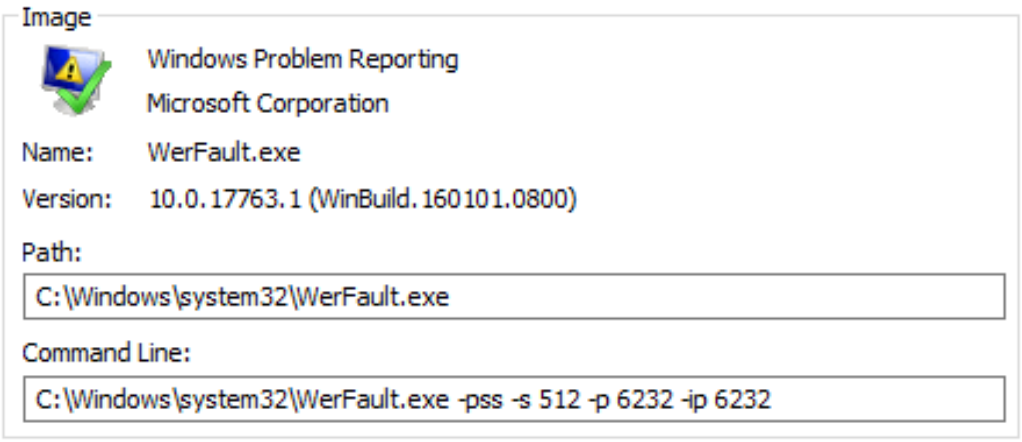


Fig 2.2.13: WerFault's command-line arguments

Finally, going to the cmd.exe process spawned by running the active version of the malware, notable observations include the calling of conhost.exe. As conhost.exe is found in the directory Windows\System32, which is where it is supposed to be located, we have no reason to suspect that this process could be malicious.

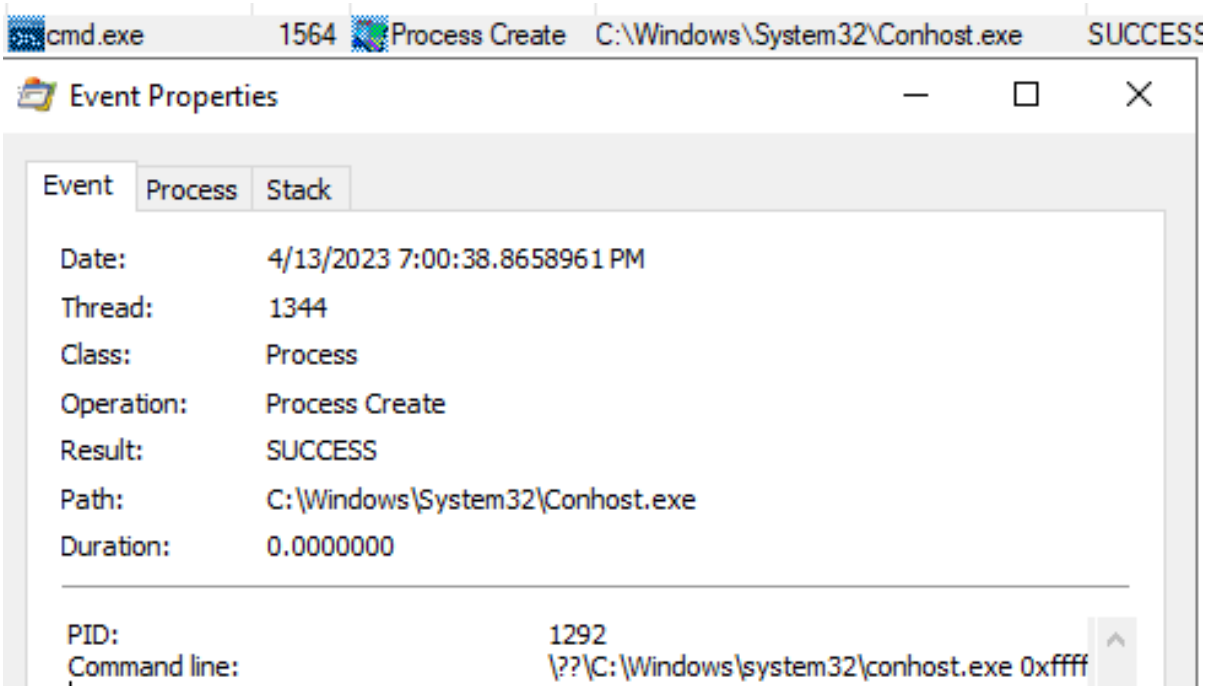


Fig 2.2.14: conhost.exe being called

Overall, the execution of the malicious programme led to various suspicious files being dropped, most of which can be found under Users\IEUser\AppData\Local\Temp as seen in the screenshot below:

```
PS C:\Users\IEUser\AppData\Local\Temp> ls


    Directory: C:\Users\IEUser\AppData\Local\Temp


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        4/23/2023  12:22 AM                WPF
-a----        4/22/2023   8:27 PM             53 .ses
-a----        4/22/2023   9:17 PM             53 18e190413af045db88dfbd29609eb877.db.ses
-a----        4/23/2023  12:23 AM            245 a.bat
-a----        4/23/2023  12:23 AM          53448 adobe.exe
-a----        4/23/2023  12:06 AM        3145782 BGInfo.bmp
-a----        4/22/2023   9:17 PM              0 mat-debug-4144.log
-a----        4/23/2023  12:06 AM         426002 MicrosoftEdgeUpdate.log
-a----        4/22/2023   8:56 PM           9131 msedge_installer.log
-a----        4/23/2023  12:23 AM          38499 MSVCR110.dat
-a----        4/23/2023  12:23 AM           9728 MSVCR110.dll
-a----        4/22/2023   8:20 PM          74648 wct560B.tmp
-a----        4/22/2023   8:25 PM          74648 wct784.tmp
-a----        4/22/2023   8:25 PM          74648 wct9A53.tmp
-a----        4/22/2023   8:25 PM          74648 wctD038.tmp
-a----        4/22/2023   8:25 PM          74648 wctD374.tmp
-a----        4/21/2023   8:05 AM            899 wctE2E7.tmp
-a----        4/22/2023   9:25 PM          74648 wctE8A7.tmp
-a----        4/22/2023   8:20 PM          74648 wctF5FA.tmp
```

Fig 2.2.15: Files in the Temp Directory

For the suspicious files such as a.bat, adobe.exe, MSVCR110.dat and MSVCR110.dll, we can derive their following hash values:

```
PS C:\Users\IEUser\AppData\Local\Temp> Get-FileHash a.bat

Algorithm       Hash
---------       ----
SHA256          D08BAD64CD2EDF6DE847A8C633935E6F095D1E02477C97AD7EF176B61CB1DD97
```

Fig 2.2.16: SHA256 Hash of a.bat

```
PS C:\Users\IEUser\AppData\Local\Temp> Get-FileHash adobe.exe

Algorithm       Hash
---------       ----
SHA256          36D76999E9090C99FAE2388CD3476134464807FC597F67C60EEBC76E32339683
```

Fig 2.2.17: SHA256 Hash of adobe.exe

```
PS C:\Users\IEUser\AppData\Local\Temp> Get-FileHash .\MSVCR110.dat

Algorithm       Hash
---------       ----
SHA256          2201C3AC955148A078D366DC1E9F552FCA4A872756D3B6DA93494CDE8D5DECD5
```

Fig 2.2.18: SHA256 Hash of MSVCR110.dat

```
PS C:\Users\IEUser\AppData\Local\Temp> Get-FileHash .\MSVCR110.dll

Algorithm       Hash
---------       ----
SHA256          D784A12FEC628860433C28CAA353BB52923F39D072437393629039FA4B2EC8AD
```

Fig 2.2.19: SHA256 Hash of MSVCR110.dll

As for eeclnt.exe, it can be found under Users\IEUser\AppData\Roaming\Windows. However, attempting to access it from Powershell leads to a deadend, as the following screenshot shows that the Windows folder in Roaming is likely to be hidden:

```
PS C:\Users\IEUser\AppData\Roaming> ls


    Directory: C:\Users\IEUser\AppData\Roaming


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         3/19/2019   3:49 AM                Adobe
d-----         10/3/2020   7:09 PM                Hex-Rays
d-----         10/3/2020   8:39 AM                JetBrains
d---s-         10/5/2020   1:54 AM                Microsoft
d-----         3/19/2019   4:29 AM                NuGet
d-----         4/23/2023  12:28 AM                Wireshark
```

Fig 2.2.20: \Users\IEUser\AppData\Roaming from Powershell

Instead, we can view it from the File Explorer, which indicates that the Windows folder is indeed hidden:

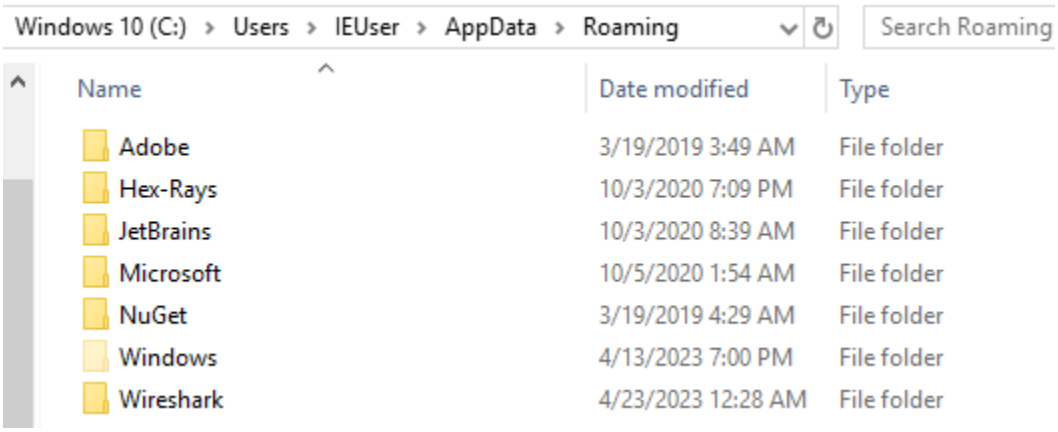


Fig 2.2.21: \Users\IEUser\AppData\Roaming from File Explorer

And when we enter the folder, we can see that eeclnt.exe as well as the MSVCR110 files can be found there:

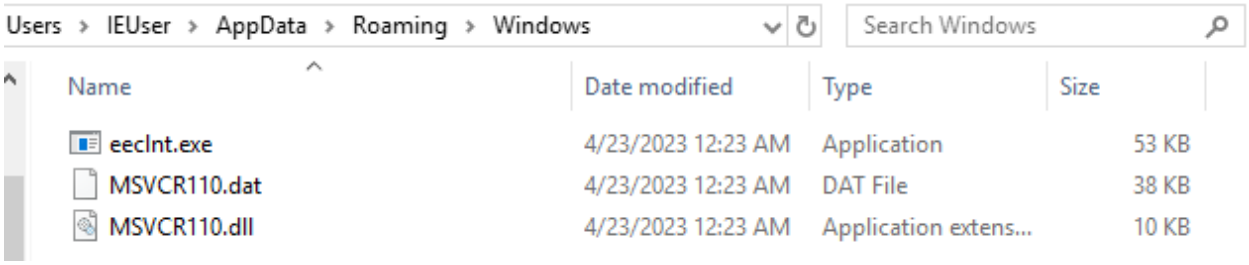


Fig 2.2.22: \Users\IEUser\AppData\Roaming\Windows from File Explorer

Ostensibly, the malware went to great lengths to hide eeclnt.exe by creating it in this directory and making it hidden.

2.3 Registry Changes

As mentioned earlier, our group wanted to gain a more thorough understanding of what the execution of eeclnt.exe entails, and as observed from this screenshot below, we can tell that when eeclnt.exe is run with command line argument “260”, the program eeclnt is registered under the registry key Software\Microsoft\Windows\CurrentVersion\Run, which makes the program run when a user logs on, implying that this is indeed for persistence.

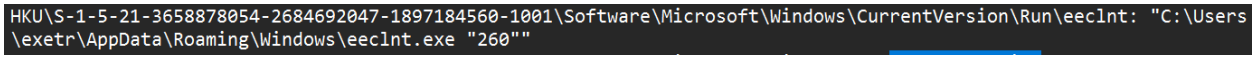


Fig 2.3.1: Changes Identified by RegShot

The rest of the changes made are with reference to the .exe file being run, such as adding an entry to the Most Recently Used (MRU).

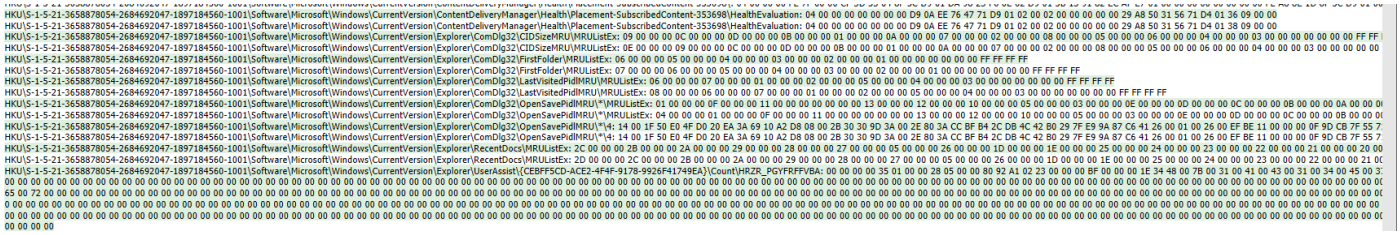


Fig 2.3.2: Changes Identified by RegShot

We also noticed a lot of drivers being deleted from the driver database as noted in the screenshot below, but nothing conclusive could be derived from this.

Keys deleted: 27250

HKLM\DRIVERS
HKLM\DRIVERS\DriverDatabase
HKLM\DRIVERS\DriverDatabase\DeviceIds
HKLM\DRIVERS\DriverDatabase\DeviceIds*AEI0276
HKLM\DRIVERS\DriverDatabase\DeviceIds*AEI9240
HKLM\DRIVERS\DriverDatabase\DeviceIds*AIW1038
HKLM\DRIVERS\DriverDatabase\DeviceIds*AKY00A1
HKLM\DRIVERS\DriverDatabase\DeviceIds*AKY1001
HKLM\DRIVERS\DriverDatabase\DeviceIds*AKY1005
HKLM\DRIVERS\DriverDatabase\DeviceIds*AKY1009
HKLM\DRIVERS\DriverDatabase\DeviceIds*AKY1013
HKLM\DRIVERS\DriverDatabase\DeviceIds*ANX2101
HKLM\DRIVERS\DriverDatabase\DeviceIds*AZT0003
HKLM\DRIVERS\DriverDatabase\DeviceIds*AZT3001
HKLM\DRIVERS\DriverDatabase\DeviceIds*AZT4001
HKLM\DRIVERS\DriverDatabase\DeviceIds*AZT4004
HKLM\DRIVERS\DriverDatabase\DeviceIds*AZT4017
HKLM\DRIVERS\DriverDatabase\DeviceIds*AZT4021
HKLM\DRIVERS\DriverDatabase\DeviceIds*BDP0156
HKLM\DRIVERS\DriverDatabase\DeviceIds*BDP2336
HKLM\DRIVERS\DriverDatabase\DeviceIds*BDP3336
HKLM\DRIVERS\DriverDatabase\DeviceIds*BRI1400
HKLM\DRIVERS\DriverDatabase\DeviceIds*BRI3400
HKLM\DRIVERS\DriverDatabase\DeviceIds*BRI9400
HKLM\DRIVERS\DriverDatabase\DeviceIds*BRIB400
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPI4050
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQA0D2
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQA0D4
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQA0D6
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQA0E1
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQA0E2
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQA0E4
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQB01D
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQB05C
HKLM\DRIVERS\DriverDatabase\DeviceIds*CPQB0EF
HKLM\DRIVERS\DriverDatabase\DeviceIds*CSC0001
HKLM\DRIVERS\DriverDatabase\DeviceIds*CSC0101
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL3001
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL3014
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL3063
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL3064
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL3067
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL3068
HKLM\DRIVERS\DriverDatabase\DeviceIds*CTL7001

Fig 2.3.3: Drivers that were deleted

We also noticed some of the registry records being deleted relating to the docx, likely to cover tracks.

HKU\S-1-5-21-3658878054-2684692047-1897184560-1001\Software\Classes\.docx-53c1fd5ac99b5690b278ffcc5a49a598
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001\Software\Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001\Software\Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file\shell
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001\Software\Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file\shell\open
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001\Software\Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file\shell\open\command
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\9\0
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\9\0\0
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\BagMRU\9\0\0\0
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\Bags\101\Shell\{5C4F28B5-F869-4E84
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\Bags\102
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\Bags\102\ComDlg
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\Bags\102\ComDlg\{5C4F28B5-F869-4E8
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\Local Settings\Software\Microsoft\Windows\Shell\Bags\102\Shell
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\.docx-53c1fd5ac99b5690b278ffcc5a49a598
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file\shell
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file\shell\open
HKU\S-1-5-21-3658878054-2684692047-1897184560-1001_Classes\docx-53c1fd5ac99b5690b278ffcc5a49a598_auto_file\shell\open\command

Fig 2.3.4: Some tracks covered

2.4 Network Activity

When we ran the malicious programme, we also ran ApateDNS simultaneously to monitor any DNS requests made by the malware. In particular, we observe that the following domain listed in the screenshot below was requested, and this seems highly suspicious especially since it has a country code top-level domain of .ga, which corresponds to Gabon.

Fig 2.4.1: ApateDNS Output

This also confirms the finding earlier from VirusTotal and thus, connections to this domain would serve as a potential network-based indicator of compromise.

3. Advanced Static Analysis

3.1 Disassembling and Analysing Main Executable in IDA(Interactive Disassembler)

Program Entrypoint & Setup Functions

When disassembling PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.bin under IDA, it can be firstly observed that the entrypoint of the program is a function named start.

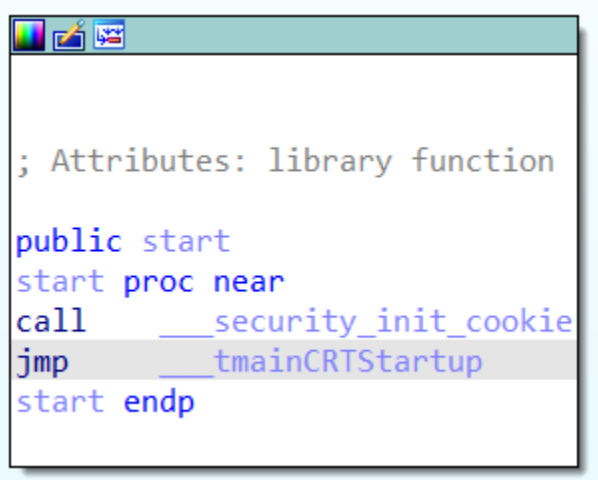


Fig 3.1.1: Start function

The start function contains only two instructions. The first is a call to __security_init_cookie. This is a function built into Microsoft's C Runtime Library (CRT). When called, it initialises the global security cookie which is used for buffer overrun protection. It provides functionality similar to a stack canary, which is placed in the stack and checks for buffer overflows.

The second is a jump to __tmainCRTStartup. Research indicates that this function is mainly used as the entrypoint to the Microsoft CRT. It initialises the CRT, which calls any static initialisers written in the code.

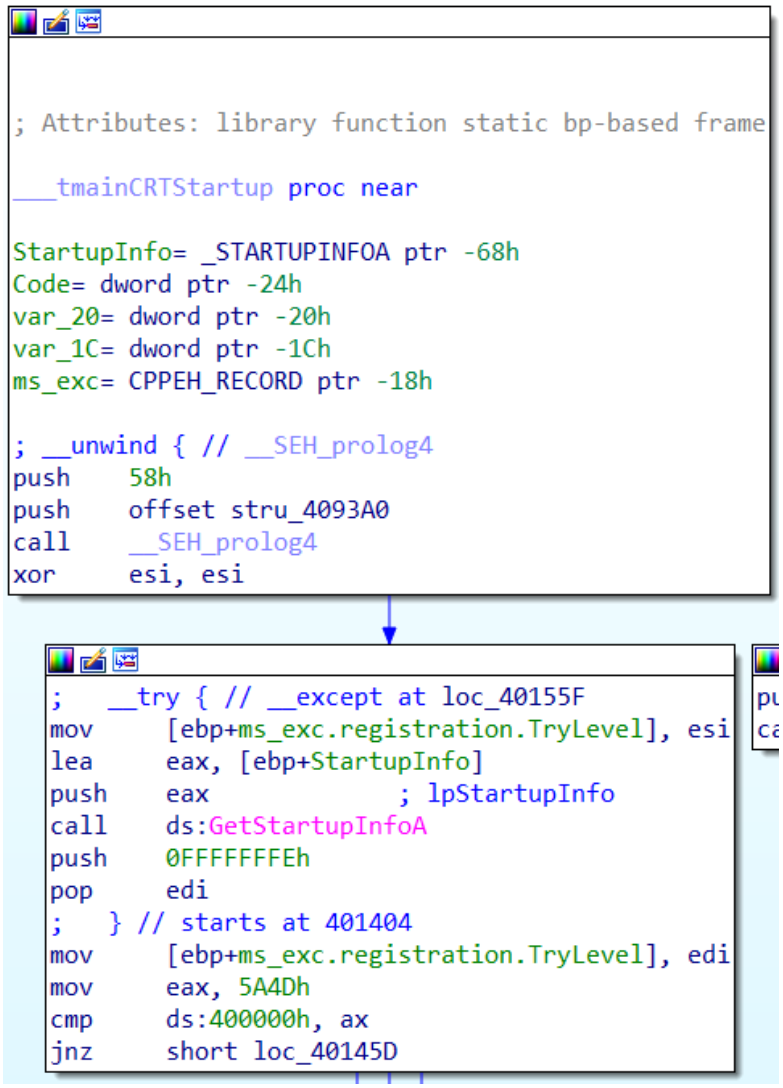


Fig 3.1.2: __tmainCRTStartup code blocks

The __tmainCRTStartup function likely consists of code automatically generated by the compiler used for this program. It consists primarily of logic that retrieves system information and environment variables.

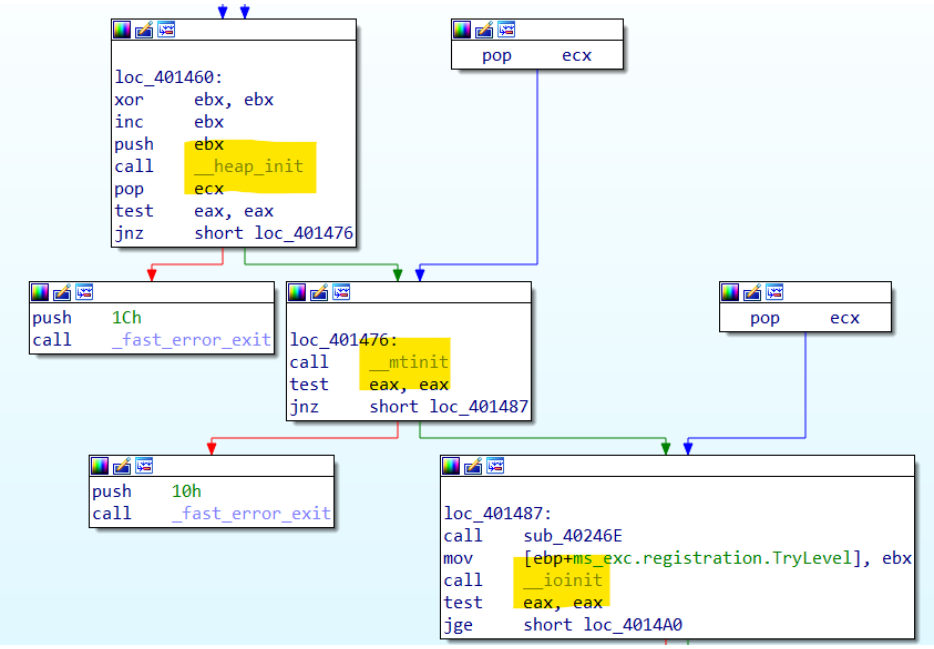


Fig 3.1.3: Init function calls

There are also calls to functions which initialise data structures such as heaps and startup functions for IO calls.

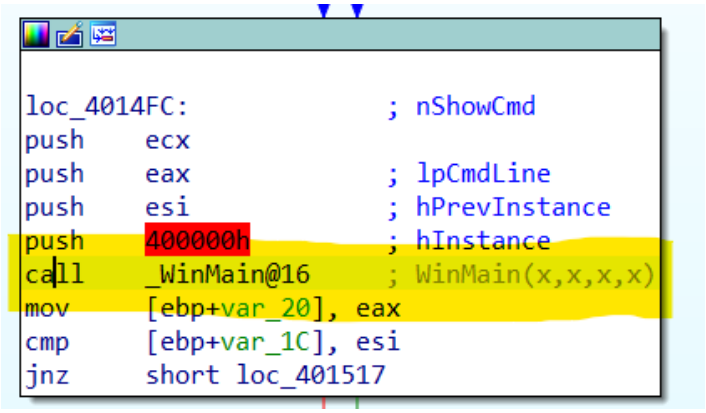


Fig 3.1.4: Call to WinMain()

One important instruction within the __tmainCRTStartup function is the call to WinMain(). WinMain() is a function provided by the Windows API that acts as an entry point for a graphical Windows-based application. It is reasonable to assume that the majority of the malware's logic can be located within this function.

This initial code block provides some background on the executable. Mainly, that the executable was compiled most likely under Visual Studio and is a C++ program that utilises Microsoft's C Runtime Library. It can also be observed that the program uses a stack canary to detect buffer

overflows. Lastly, the malware was compiled as a graphical Windows-based application evident from the main method of the program being WinMain().

WinMain Function

When viewing the disassembled code of WinMain() within IDA, it appears that the majority of the malware's functionalities within this executable are contained within this function.

```
mov     eax, 1028h
call    __alloca_probe
mov     eax, ___security_cookie
xor     eax, esp
mov     [esp+1028h+var_4], eax
```

Fig 3.1.5: Use of __security_cookie

Firstly, the loading of __security_cookie demonstrates the buffer overflow protection provided by CRT being used in the scope of this function.

```
push    206h            ; Size
push    eax             ; Val
lea     ecx, [esp+1040h+var_7FA]
push    ecx             ; void *
mov     [esp+1044h+NumberOfBytesWritten], 0
mov     [esp+1044h+FileName], ax
call    memset
```

Fig 3.1.6: Declaration of void pointers and call to memset

Within the first code block of the WinMain() function, we were able to observe assembly code that appears to declare void pointers of size 0x206 or 518 bytes. There are multiple offsets used with regards to the values moved into the registers, which include "FileName", "Buffer" and "File". This is followed by a call to the memset function which is a C function that fills a block of memory with a certain value. There were observed to be a total of 5 instances of this block of code, indicating that 5 void pointers were specifically declared within this code block.

```
add     esp, 3Ch
lea     ecx, [esp+1038h+Buffer]
push    ecx             ; lpBuffer
push    104h            ; nBufferLength
call    ds:GetTempPathW
```

Fig 3.1.7: Call to GetTempPathW

Following the declaration of the void pointers, there is a subsequent series of assembly code involving a call to the GetTempPathW function from the Windows API. This function retrieves the

path of the directory designated for temporary files. Microsoft API documentation indicates that examples of the returned value can be directories such as “C:\TEMP\”. The result is indicated to be stored into a buffer location specified by the value in register ECX with a length of 0x104 or 260 bytes.

```
mov     esi, ds:wsprintfW
push    offset aAdobeExe ; "adobe.exe"
lea     edx, [esp+103Ch+Buffer]
push    edx
lea     eax, [esp+1040h+File]
push    offset aSS       ; "%s%s"
push    eax              ; LPWSTR
call    esi ; wsprintfW
push    offset aMsvcr110Dll ; "MSVCR110.dll"
lea     ecx, [esp+104Ch+Buffer]
push    ecx
lea     edx, [esp+1050h+var_5F4]
push    offset aSS       ; "%s%s"
push    edx              ; LPWSTR
call    esi ; wsprintfW
push    offset aMsvcr110Dat ; "MSVCR110.dat"
lea     eax, [esp+105Ch+Buffer]
push    eax
lea     ecx, [esp+1060h+FileName]
push    offset aSS       ; "%s%s"
push    ecx              ; LPWSTR
call    esi ; wsprintfW
push    offset aABat     ; "a.bat"
lea     edx, [esp+106Ch+Buffer]
push    edx
lea     eax, [esp+1070h+var_A04]
push    offset aSS       ; "%s%s"
push    eax              ; LPWSTR
call    esi ; wsprintfW
```

Fig 3.1.8: Strings used and calls to wsprintfW

Subsequent lines of assembly code as shown above in figure 3.1.8 depict multiple calls to the wsprintfW function that also appears to involve a number of strings previously discovered during basic static analysis. Each wsprintfW function call is prefaced with strings that indicate filenames and the format specified “%s%s”. As wsprintfW is a function under the printf family of functions, it appears that the the use of the double string format specifier will cause concatenation of each of the values of “adobe.exe”, “MSVCR110.dll”, “MSVCR110.dat” and “a.bat” with another string.

WsprintfW is a function under the Windows API that writes formatted data to the specified buffer. Considering that previously, the call to GetTempPathW was called, causing the temporary directory used by Windows to be stored within the stack, this block of code could be used to form absolute file paths for the four files in question. The output from the 4 calls to wsprintfW are directed to buffers specified in four different addresses.

```
add     esp, 40h
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    2               ; dwShareMode
push    40000000h       ; dwDesiredAccess
mov     esi, ds:CreateFileW
lea     ecx, [esp+1050h+FileName]
push    ecx             ; lpFileName
call    esi ; CreateFileW
```

Fig 3.1.9: Calls to CreateFileW

Following the calls to wsprintfW, we were able to observe a block of code that pushes a number of parameters onto the stack before calling the CreateFileW function from the Windows API which creates or opens a file or I/O device. There were 4 instances of calls to this function. For each of the calls, the following parameters were pushed to the stack before being passed to the function:

Parameter	Value	Definition or Usage
hTemplateFile	0	A valid handle to a template file with the GENERIC_READ access right. Current value provides no handle to the function
dwFlagsAndAttributes	0x80	The file or device attributes and flags. Current value specifies FILE_ATTRIBUTE_NORMAL, meaning no other attributes are set.
dwCreationDisposition	2	An action to take on a file or device that exists or does not exist, current value will always create a new file regardless of whether a file is already existing.
lpSecurityAttributes	0	A pointer to a SECURITY_ATTRIBUTES structure, determining whether a created file or device can be inherited by child processes. Current value states it cannot be inherited by child processes.

dwShareMode	2	Requested sharing mode of the file or device, current value indicates FILE_SHARE_WRITE, where each subsequent open operation on a file or device can get read access.
dwDesiredAccess	0x40000000	Requested access to the file or device, current value indicates GENERIC_WRITE, allowing write access.
lpFileName	Found in stack (differs for each call)	Name of the file or device to be created or opened.

Considering the functionalities of the malware observed up to now, these 4 calls to CreateFileW are likely to correspond to the creation of the files “adobe.exe”, “MSVCR110.dll”, “MSVCR110.dat” and “a.bat” on the filesystem. When the CreateFileW function returns, if the resource is successfully created, the open handler will be returned that points to the specified file or device. However, if it fails, it will return a file handler with the value INVALID_HANDLE_VALUE.

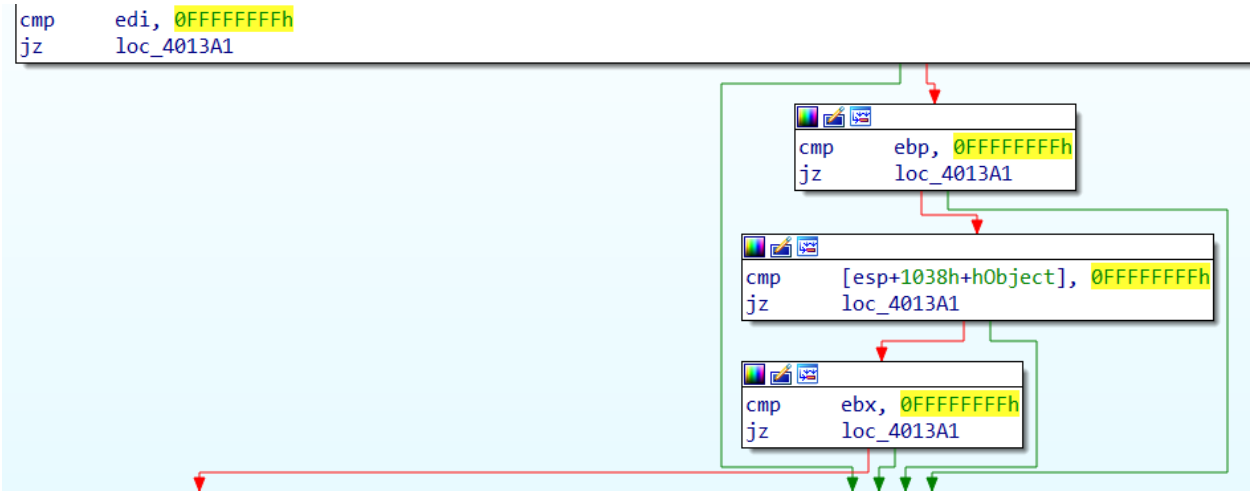


Fig 3.1.10: Checks for successful file creation

Following the calls to the CreateFileW function, there exists 4 comparisons along with jump-if-zero functions that compare the values of several registers and memory addresses against 0xFFFFFFF. It is to be noticed that 0xFFFFFFF represents -1 when converted to a signed integer. For the values returned by CreateFileW, Microsoft’s documentation states that the INVALID_HANDLE_VALUE returned by the function upon failure when creating the resource holds the value of -1, except that it is created under the file handler type. As such, these four comparison functions are related to checking whether the 4 calls to CreateFileW function have failed.

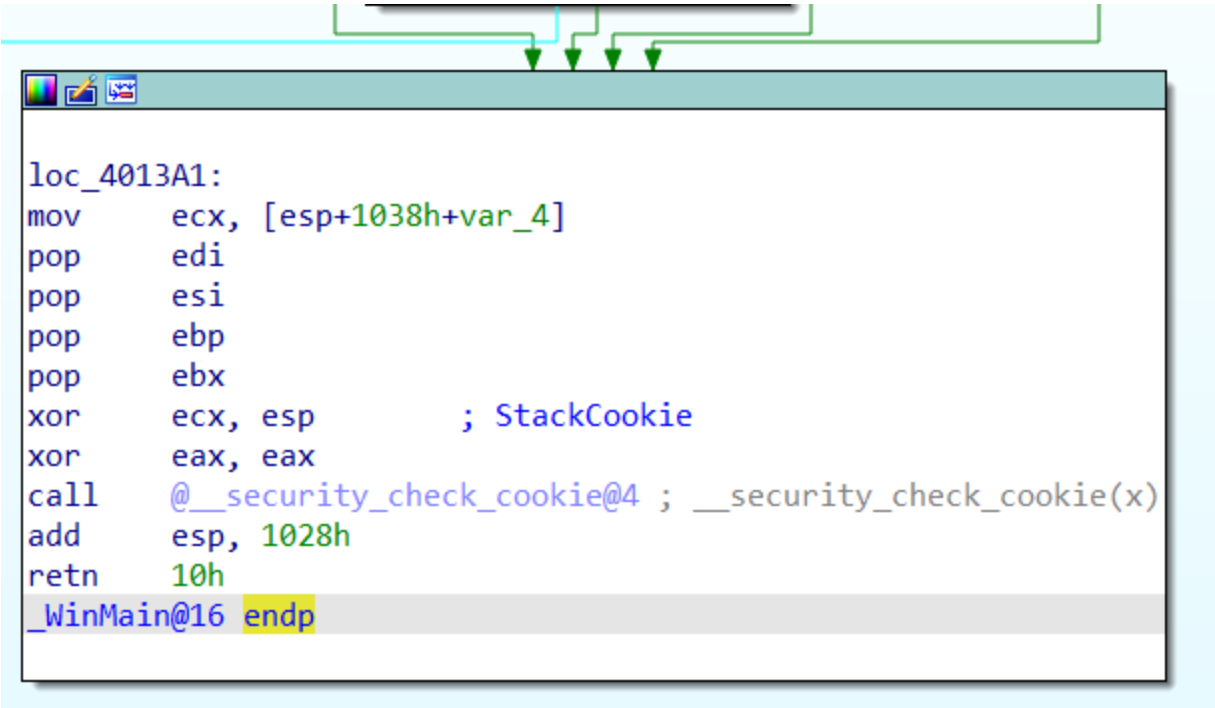


Fig 3.1.11: Code block triggering return from WinMain()

If any of the files were not able to be successfully created, the program will jump to a code block which causes the WinMain() function to exit and return to its caller. It can also be observed that there is a call to __security_check_cookie at this juncture, which checks against buffer overflows within the stack.

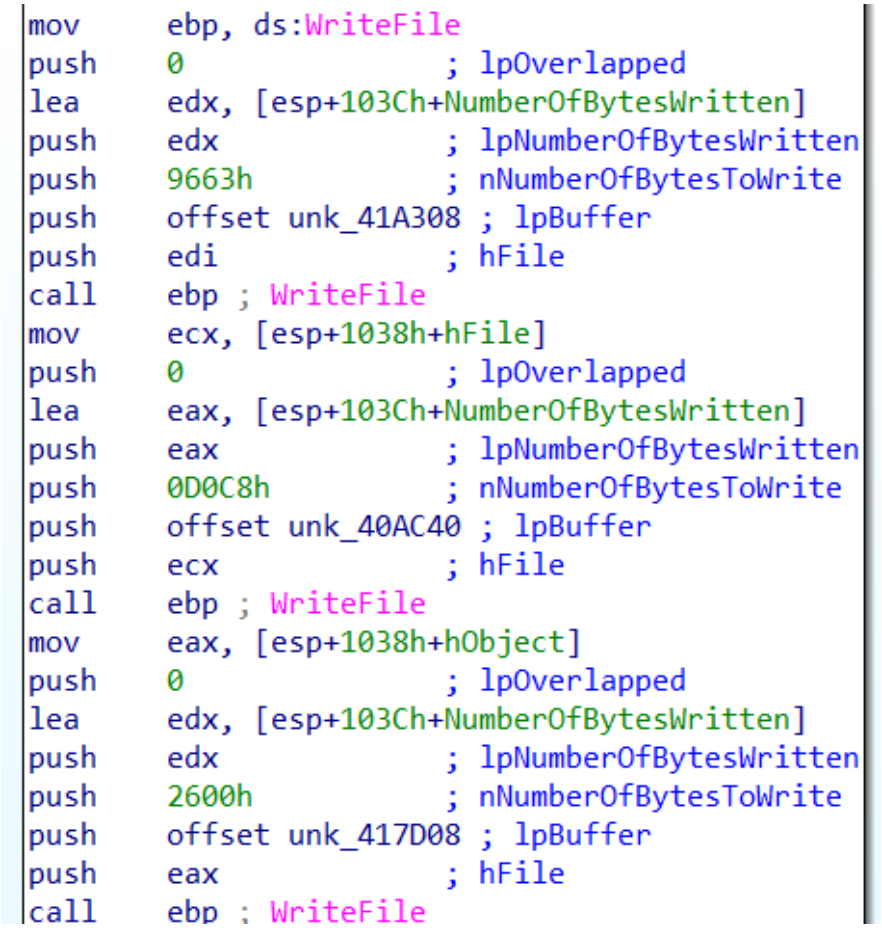


Fig 3.1.12: Calls to WriteFile function

Provided that all four calls to CreateFileW are successful, it can be observed through IDA that the subsequent code blocks involve three calls to the WriteFile function, part of the Windows API which Writes data to the specified file or input/output (I/O) device. It is highly likely that these three function calls pertain to writing information to three out of the four files previously created. Comparing the sizes passed through the nNumberOfBytesToWrite parameter for the function, we are able to guess that these three functions will write content to the following files

- adobe.exe (nNumberOfBytesToWrite = 0xD0C8, actual size = 53488B)
- MSVCR110.dat (nNumberOfBytesToWrite = 0x9663, actual size = 38499B)
- MSVCR110.dll (nNumberOfBytesToWrite = 0x2600, actual size = 9278B)

```
push    103h            ; Size
lea     ecx, [esp+103Ch+var_101B]
push    0               ; Val
push    ecx             ; void *
mov     [esp+1044h+Filename], 0
call    _memset
push    103h            ; Size
lea     edx, [esp+1048h+var_F17]
push    0               ; Val
push    edx             ; void *
mov     [esp+1050h+var_F18], 0
call    _memset
add     esp, 18h
push    104h            ; nSize
lea     eax, [esp+103Ch+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    ds:GetModuleFileNameA
```

Fig 3.1.13: Call to GetModuleFileNameA

Subsequently, the code block involves declaration of two void pointers. Both void pointers are initialized with a value of 0 and a size of 0x103 or 259 bytes. The values are then placed in memory with the call to the _memset function. The executable executes a call to the function GetModuleFileNameA. This is a function from the Windows API which retrieves the fully qualified path for the file that contains the specified module loaded by the current process.

The executable likely utilises one of the void pointers to store the value of the module whose path is being requested and the other void pointer to store the fully qualified path of the module.

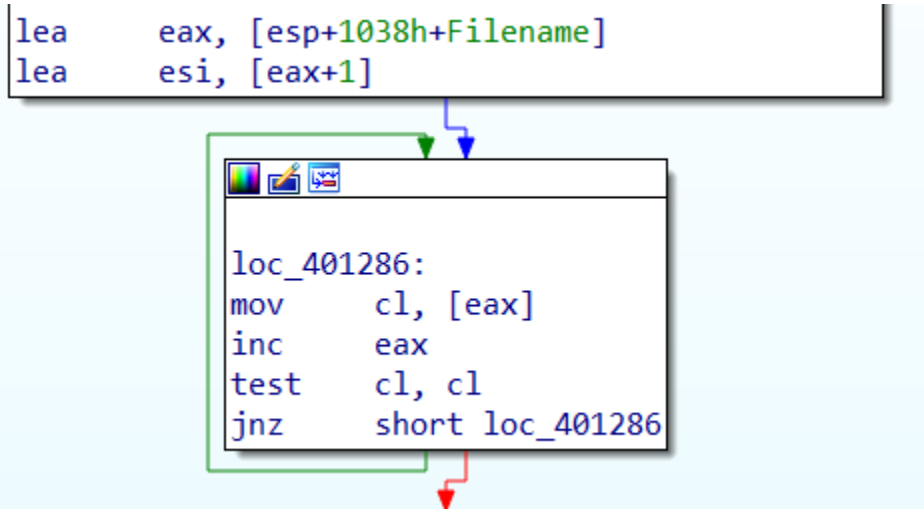
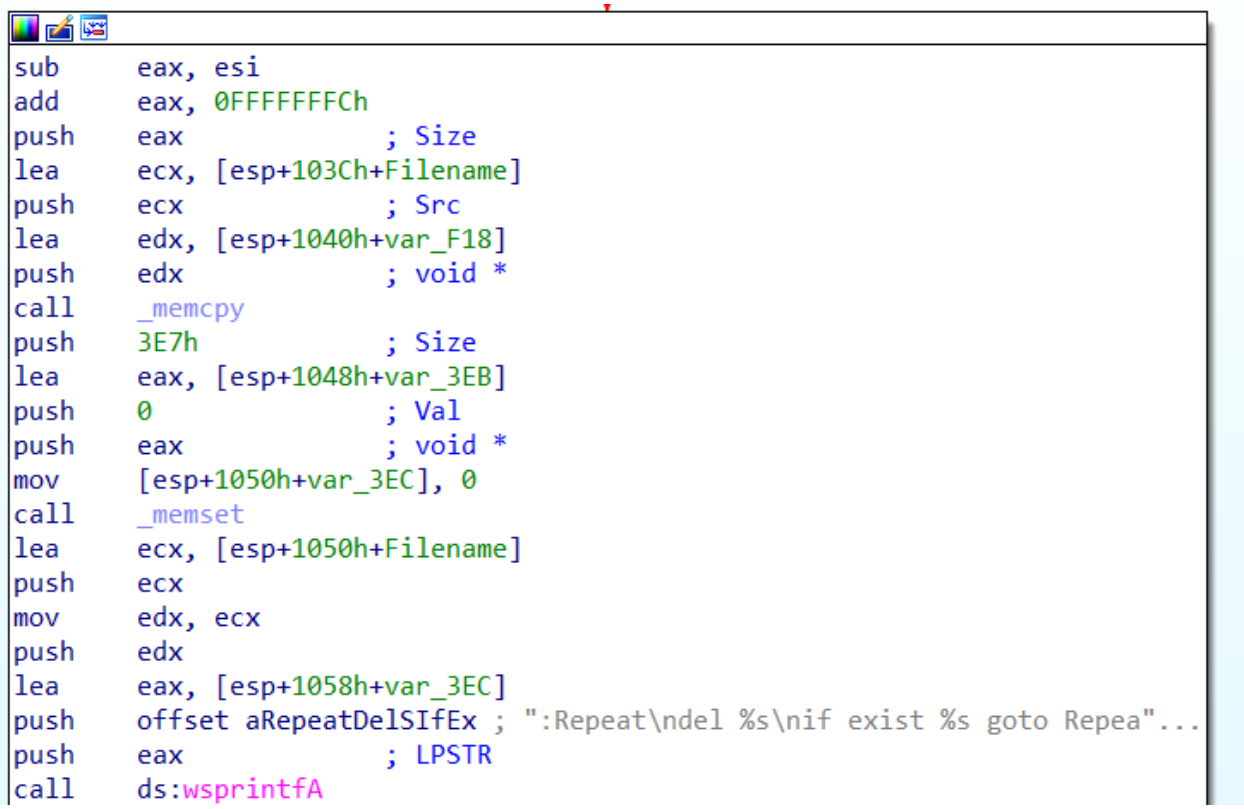


Fig 3.1.14: Loop and check

There is a following loop coding structure that increments a counter that is stored in the EAX register. This loop structure after the effective address of a variable is loaded into the stack, as indicated by the load effective address from the stack pointer. As such, it most likely performs a character-by-character check or validation against the path returned by the prior call to GetModuleFileNameA.



```
sub     eax, esi
add     eax, 0FFFFFFFCh
push    eax             ; Size
lea     ecx, [esp+103Ch+Filename]
push    ecx             ; Src
lea     edx, [esp+1040h+var_F18]
push    edx             ; void *
call    _memcpy
push    3E7h            ; Size
lea     eax, [esp+1048h+var_3EB]
push    0               ; Val
push    eax             ; void *
mov     [esp+1050h+var_3EC], 0
call    _memset
lea     ecx, [esp+1050h+Filename]
push    ecx
mov     edx, ecx
push    edx
lea     eax, [esp+1058h+var_3EC]
push    offset aRepeatDelSIfEx ; ":Repeat\ndel %s\nif exist %s goto Repea"...
push    eax             ; LPSTR
call    ds:wsprintfA
```

Fig 3.1.15: WsprintfA function call related to batch file

Similar to previously observed structures, within the next block of code after the loop structure, there is a declaration of and initialisation of two void pointers with the values of 0. This is followed by a call to the wsprintfA function, which is a function under the printf family which prints formatted data to a specified buffer. Before wsfprintfA is called, it is observed that the string containing format specifiers “Repeat\ndel %s\nif exist %s goto Repeat” is pushed ontot he stack.

Considering the flow of operations, it is likely that the first void pointer was used to store the value to be substituted to the string format specifier, while the second void pointer is used to store the resulting value after the wsprintfA operation. The string passed to the string format specifier is likely to be a fully qualified path. This combined with the string which appears to be the contents of a batch file. The “del” and “if exist” syntax accept file paths.

```
add     esp, 28h
push    0               ; lpOverlapped
lea     ecx, [esp+103Ch+NumberOfBytesWritten]
push    ecx             ; lpNumberOfBytesWritten
push    eax             ; nNumberOfBytesToWrite
lea     edx, [esp+1044h+var_3EC]
push    edx             ; lpBuffer
push    ebx             ; hFile
call    ebp ; WriteFile
```

Fig 3.1.16: Writing of contents to batch file

Having observed that the formatted string to be stored in the stack at location with offset pointed to by var_3EC, there is a subsequent call to the WriteFile function which writes content to a provided file handler. Considering the previously analysed instructions, it is likely that this WriteFile call will initiate the writing of the formatted string “Repeat\ndel %s\nif exist %s goto Repeat” with fully qualified paths to the “a.bat” file.

One interesting difference in this call to WriteFile compared to previous calls is that the value passed as the nNumberOfBytesToWrite parameters seems to be read off the stack, meaning that it is dynamically set. This fits the context as different systems would have differently sized fully qualified paths to the target destination due to factors such as different usernames. As such, the total size of the batch file varies across systems and has to be calculated dynamically.

```
mov     esi, ds:CloseHandle
push    edi             ; hObject
call    esi ; CloseHandle
mov     eax, [esp+1038h+hFile]
push    eax             ; hObject
call    esi ; CloseHandle
mov     ecx, [esp+1038h+hObject]
push    ecx             ; hObject
call    esi ; CloseHandle
push    ebx             ; hObject
call    esi ; CloseHandle
```

Fig 3.1.17: Calls to CloseHandle

Afterwards, there were observed to be 3 calls to the CloseHandle function from the Windows API which closes an open object handle. This is likely to be used to close the handles previously returned when CreateFileW was called to create and write contents to the files. Currently, we are unable to ascertain which file handles were specifically closed.

```
mov     ebx, ds:ShellExecuteW
push    0                   ; nShowCmd
push    0                   ; lpDirectory
push    0                   ; lpParameters
lea     edx, [esp+1044h+File]
push    edx                 ; lpFile
push    offset Operation ; "open"
push    0                   ; hwnd
call    ebx ; ShellExecuteW
```

Fig 3.1.18: Execution of open command in shell

Further down the current code block, it can be observed that there is a call to the ShellExecuteW function from the Windows API, which performs an operation on a specified file. Specifically, the lpFile parameter is set to reference a location from the stack, which from observing the offset value used of [esp+1044h+File], indicates that it points to the path of the “adobe.exe” executable previously created. The associated operation is “open”, which will open the executable and effectively start running it. The following parameters were passed to the ShellExecuteW function.

Parameter	Value	Definition or Usage
nShowCmd	0	The flags that specify how an application is to be displayed when it is opened. Current value displays no windows.
lpDirectory	0	A pointer to a null-terminated string that specifies the default (working) directory for the action. Current value specifies current working directory.
lpParameters	0	Specifies the parameters to be passed to the application. Currently, no parameters are passed to application
lpFile	Found in stack	A pointer to a null-terminated string, specifies the file or object on which to execute the specified operation.
lpOperation	open	A pointer to a null-terminated string, specifies the action to be performed. “Open” will open a specified file or folder
hwnd	0	Handle to the parent window used for displaying a UI or error messages. Current value indicates no associated GUI window

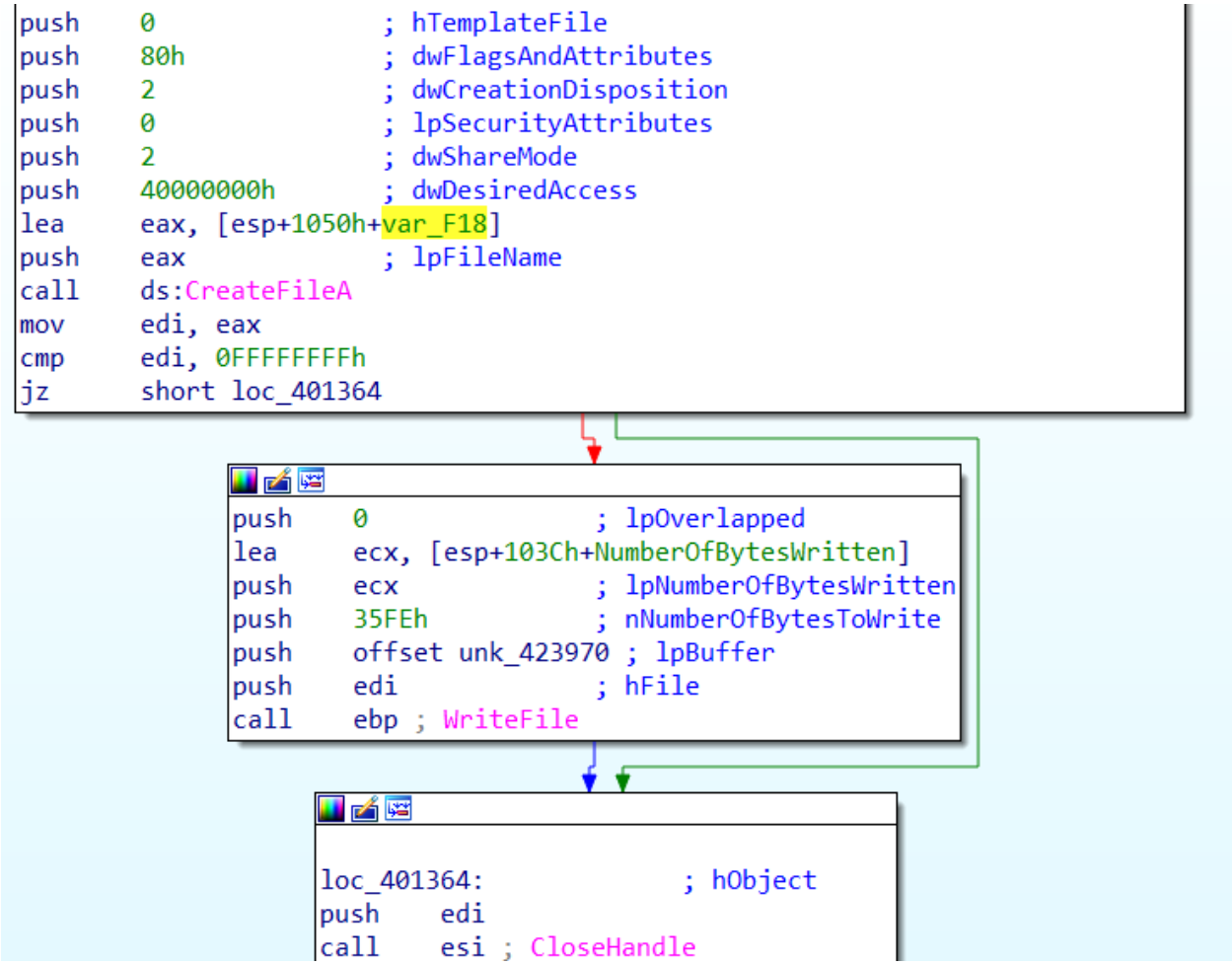


Fig 3.1.19: Additional calls to CreateFileA, WriteFile and CloseHandle

The next series of instructions provide the functionality of creating a new file, writing contents to the file, and closing the handler returned when creating the file. The parameters passed to the CreateFileA function through pushing onto the stack are the same as the ones used in prior calls to CreateFileA. The filename in this instance is stored onto the stack at an offset of [esp+1050h+var_F18], which upon further analysis appears to reference the returned value from the GetModuleFileNameA function call, which contains the name of the current executable. Afterwards, 0x35FE or 13822 bytes of data were written to the file, before the handle referencing this file is closed.

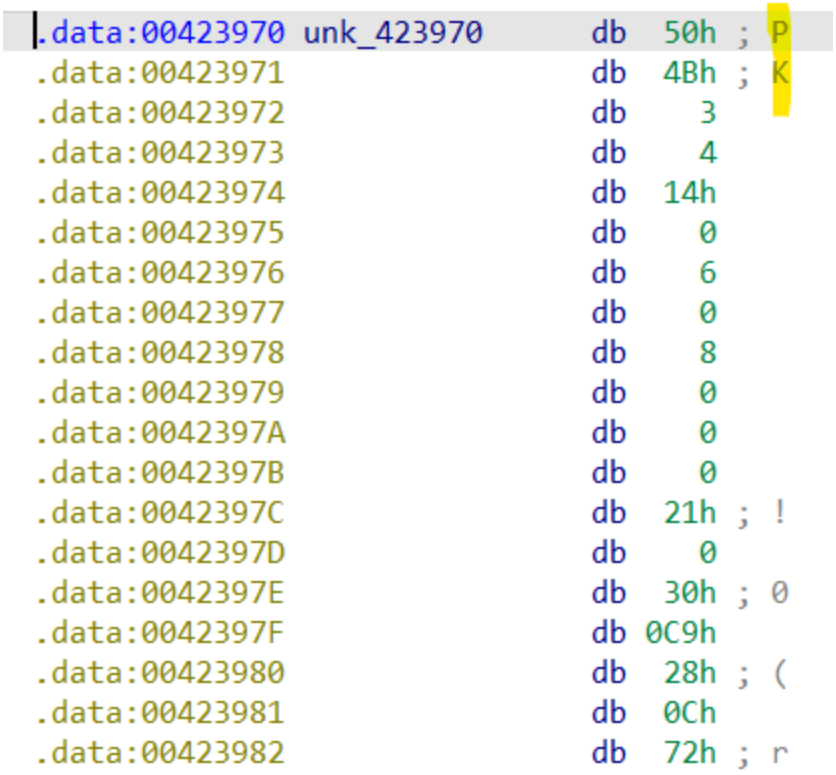


Fig 3.1.20: Contents written to new file

Further analysing the contents that are written to the file, it can be observed that the contents appear to have the “PK” header, hinting that the file is treated as a ZIP archive. When considering the context of this action, this function appears to effectively remove the malicious executable file currently being run in an attempt to remove any traces of infection. Furthermore, the PK header is a valid header to represent Microsoft Word documents. This matches the behaviour previously noted in basic dynamic analysis, that the size of the malicious file is now significantly smaller, but it still appears to be treated properly as a word document.

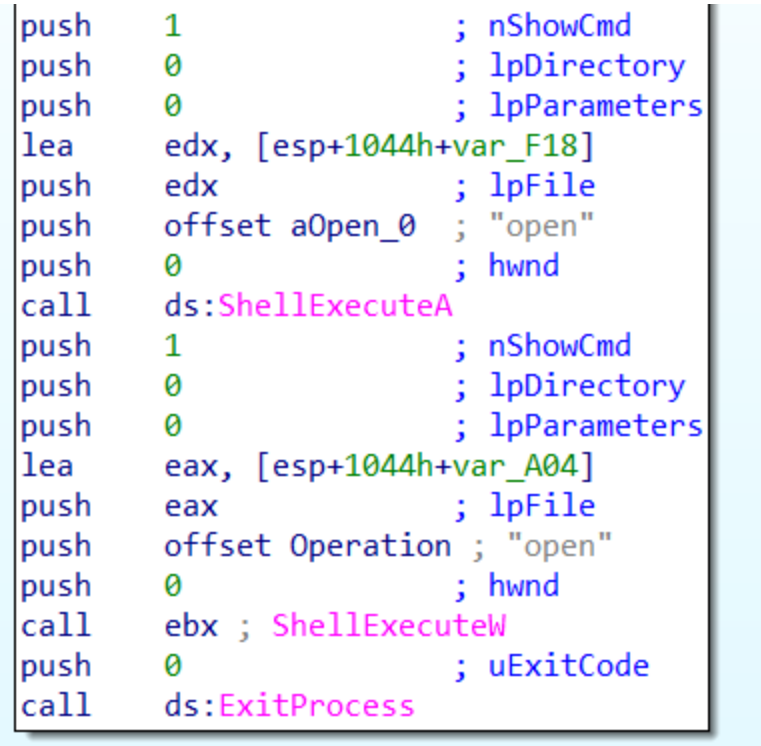


Fig 3.1.21: Further calls to ShellExecuteW and exiting of process

After the previous steps taken by the executable to create a new file in attempting to erase its tracks, there were observed to be two more calls to ShellExecuteA. The first call references the stack at offset [esp+1044h+var_F18] which, based on previous observations, is the path to the word document recently created. The open function is called, along with the nShowCmd being set to 1. This will result in the word document being opened by the operating system. These are the instructions that trigger the display of a word document when the executable is initially ran during basic dynamic analysis

The second call to ShellExecuteW has the parameter lpFile referencing the stack at [esp+1044h+var_A04]. Similarly, it is executed with the nShowCmd parameter set to 1, meaning that there will be a resulting window shown upon execution of the function. Based on observations of the stack offset, it can be concluded that the value passed to the function is the full path of “a.bat”. When executed, it will loop to delete the initial malicious executable in an attempt to cover the tracks of the malware.

Lastly, the program calls ExitProcess, which is a Windows API function that terminates the calling function and all of its threads. This marks the end of the functionality of the initial PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.bin executable.

Overview of Functionality

Through our advanced static analysis of the initial PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.exe file for the malware, it appears that this executable serves the primary purpose of a dropper, where it only acts to create and drop more malicious files within the filesystem, execute or start them, then remove traces of the current file. Many of the assembly code that was reported under this section correspond to the behaviour we have previously observed through basic static and dynamic analysis.

3.2 Disassembling and Analysing adobe.exe in IDA

Program Entrypoint and Function Calls

Firstly, it should be noted that the adobe.exe executable is signed and Windows has shown that the contents of the executable are still valid according to the certificates that have been used to sign the executable. As such, it is unlikely that any of the functionalities of adobe.exe are malicious by nature when considered in isolation.

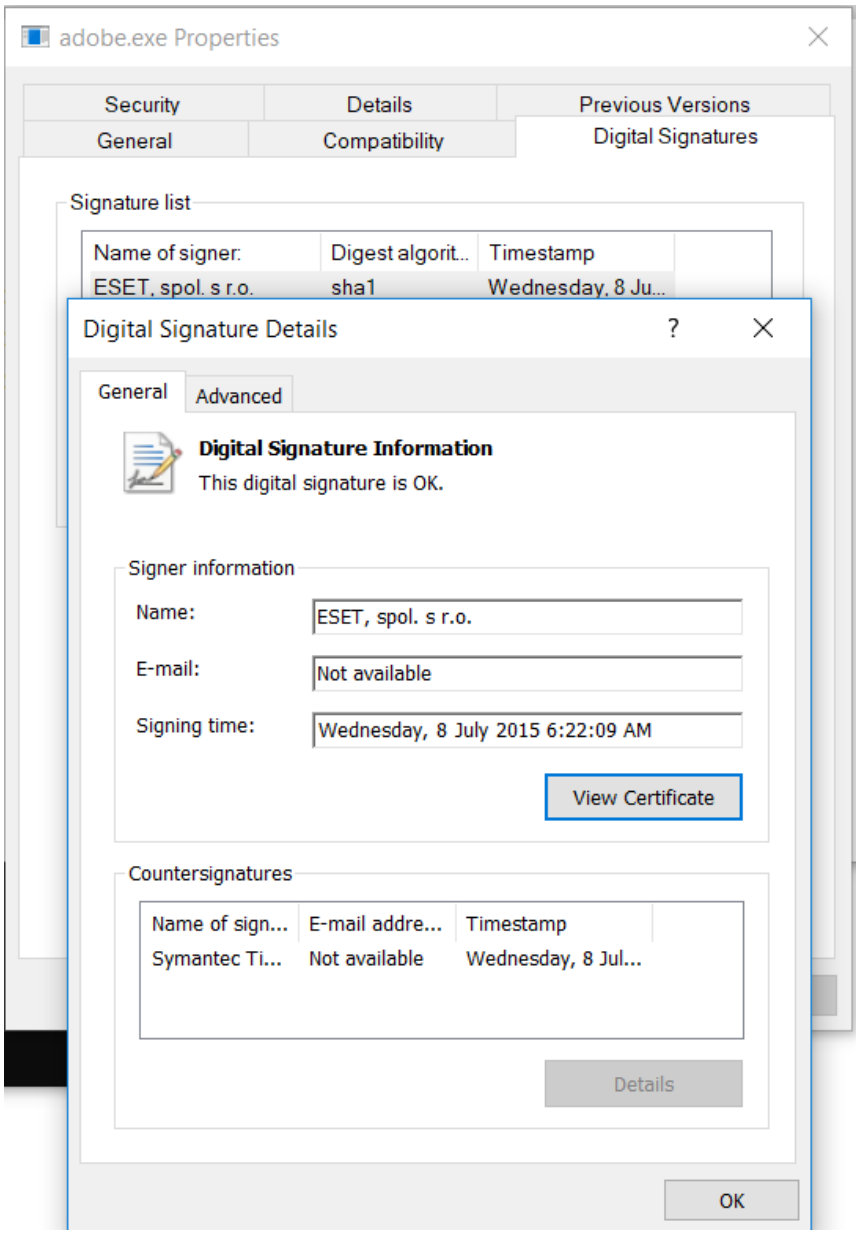


Fig 3.2.1: Digital signature of adobe.exe.

However, it was observed that the signer was specified as ESET, which is a company providing antivirus and internet security solutions, which has no relation to Adobe. Hence, it is likely that the signed executable was used to utilise additional functions exported by other modules as a step towards malicious actions.

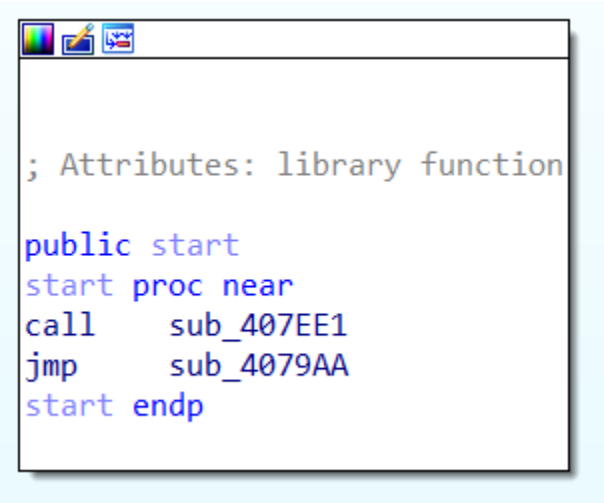


Fig 3.2.2: Entrypoint of adobe.exe

We disassembled the adobe.exe executable and observed the entrypoint which is the Start function as shown in the figure above. It is noticed that this entrypoint is similar to that of the initial executable, with a call to a subroutine and a jump to another subroutine.

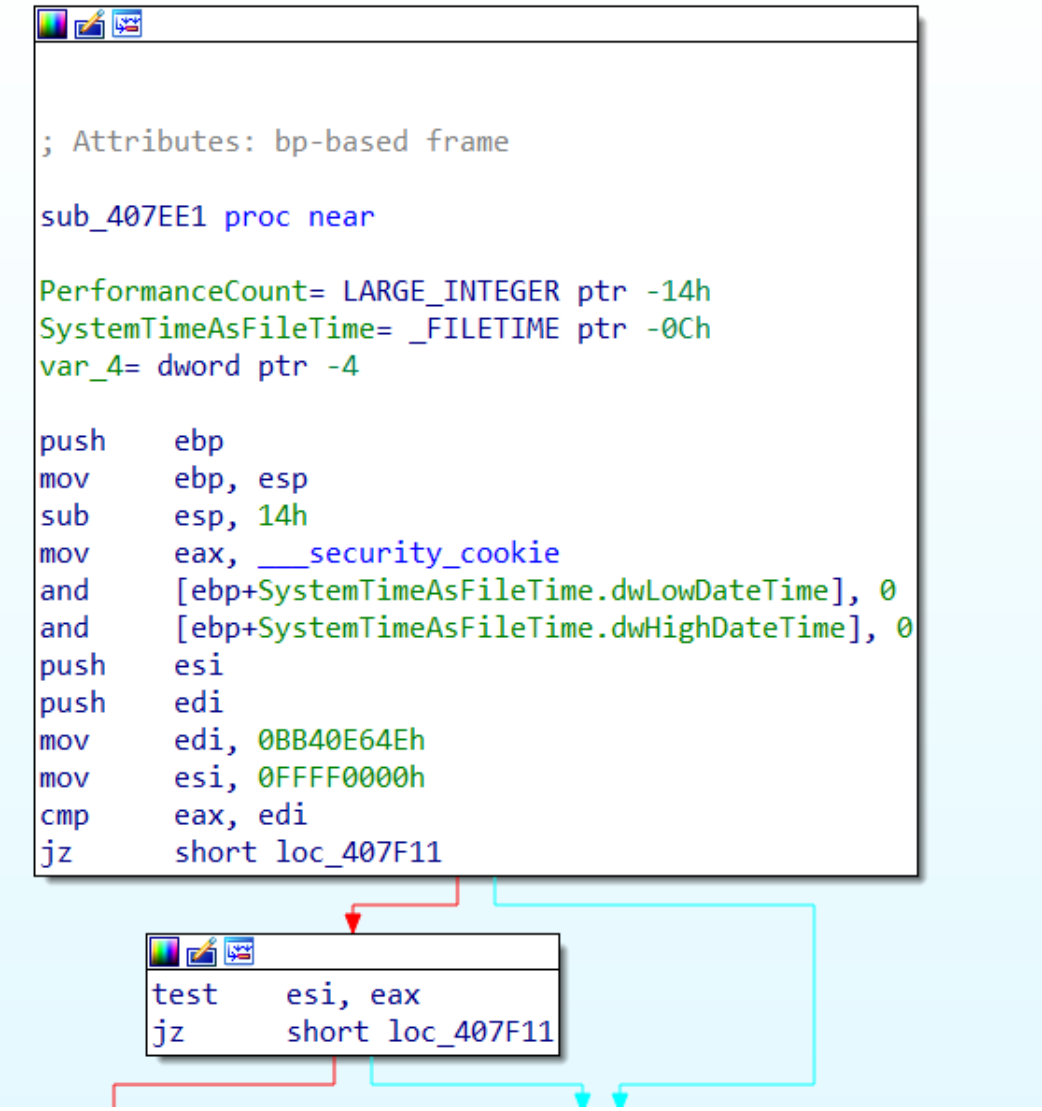


Fig 3.2.3: Beginning of sub_407EE1

The first subroutine located at sub_407EE1 hints that this function primarily deals with a declared __security_cookie. Based on research, it is indicated that this is a Windows native implementation of a defence against buffer overflows. The function ends with a return to the caller.

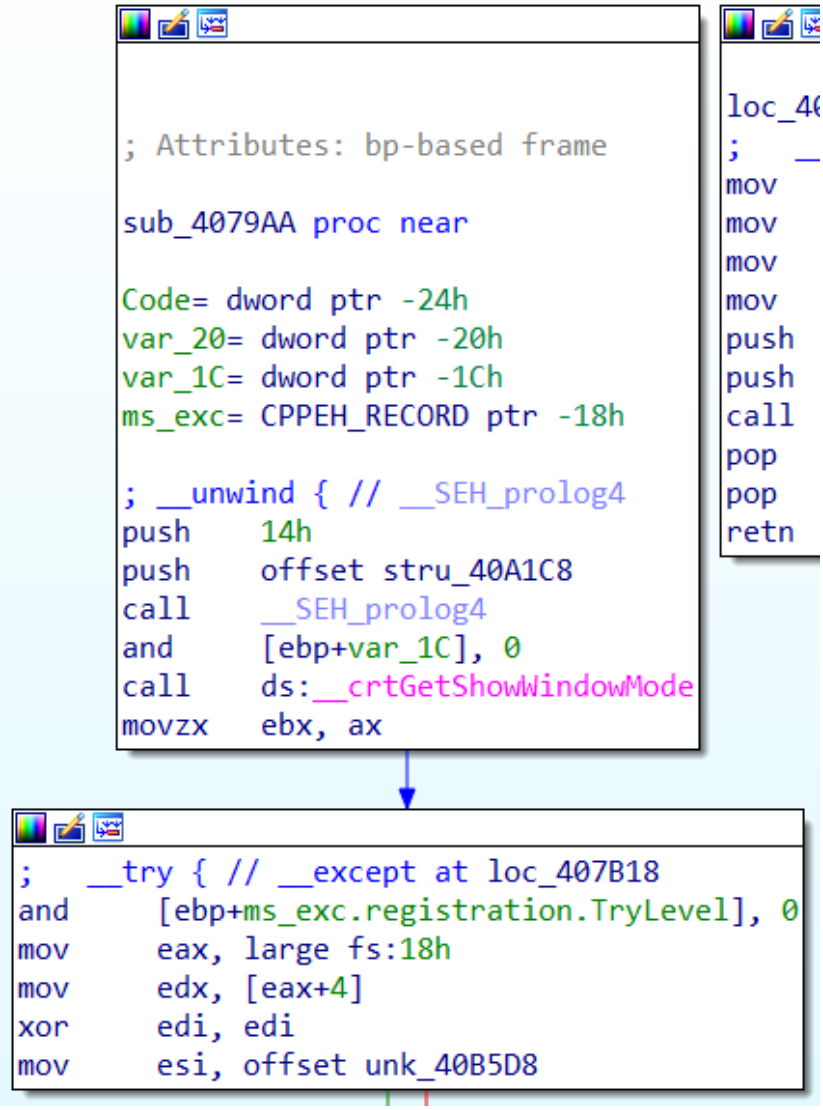


Fig 3.2.4: Subroutine 0x4079AA

The following call to the subroutine at 0x4079AA similarly appears to be a complex function which is automatically generated by the compiler, related to the C Runtime library (CRT), evident from the call to __crtGetShowWindowMode within the first code block of the subroutine.

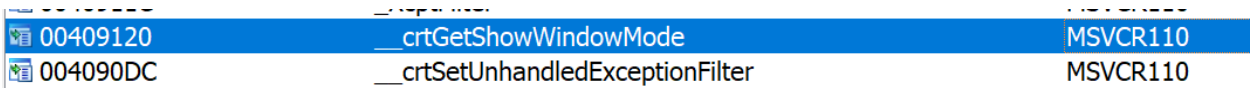


Fig 3.2.5: Location of __cerGetShowWindowMode imported from MSVCR110

A peculiar nature of this function in the current context is that the __crtGetShowWindowMode is not locally defined, but called towards a function of the same name that was exported by another module, namely MSVCR110.dll.

Further advanced static analysis of the adobe.exe was not conducted by the group as we deemed the remainder of the functions to be non-malicious due to the valid signature of the executable, indicating that the software was designed to work as intended.

Overview of Functionality

The executable adobe.exe by itself does not seem to contain any malicious code as its authenticity can be validated by its signature. However, it most likely is used as a medium for attackers to run code that is exported by other modules. In this specific case, the suspicious behaviour would be the call to the __crtGetShowWindowMode function that is exported by MSVCR110.DLL.

Further research on the behaviour of DLL loading shows that conditions that allow DLL search order hijacking can be leveraged to allow legitimate and non-malicious programs to run code from malicious DLLs.

Considering that adobe.exe is located in the %TEMP% folder as specified by Windows, it was also noticed that MSVCR110.dll was also dropped in this same directory. The DLL load order will cause the operating system to first search the directory from which the executable was run from, and in this case, the malicious dropped MSVCR110.dll will be the effective DLL loaded by adobe.exe. When the function call to __crtGetShowWindowMode is made, the corresponding function exported by MSVCR110.dll will be invoked.

3.3 Disassembling and Analysing MSVCR110.dll in IDA

Exported Function Within DLL

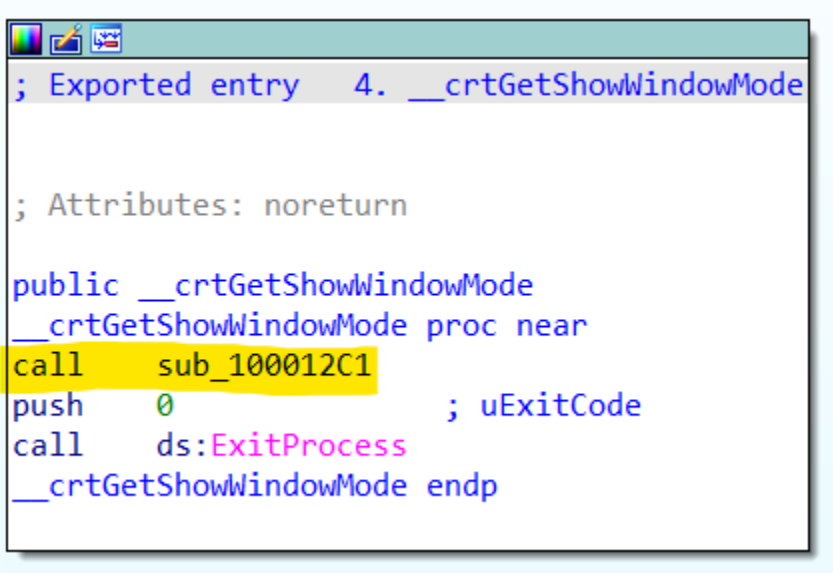


Fig 3.3.1: Entrypoint of exported function __crtGetShowWindowMode

Within the entrypoint of the exported __crtGetShowWindowMode function in the MSVCR110.dll dropped by the malware, it is observed that there is only a call to a subroutine located at 0x100012C1. When the subroutine returns, the current process (adobe.exe) will be terminated with an exit code of 0.

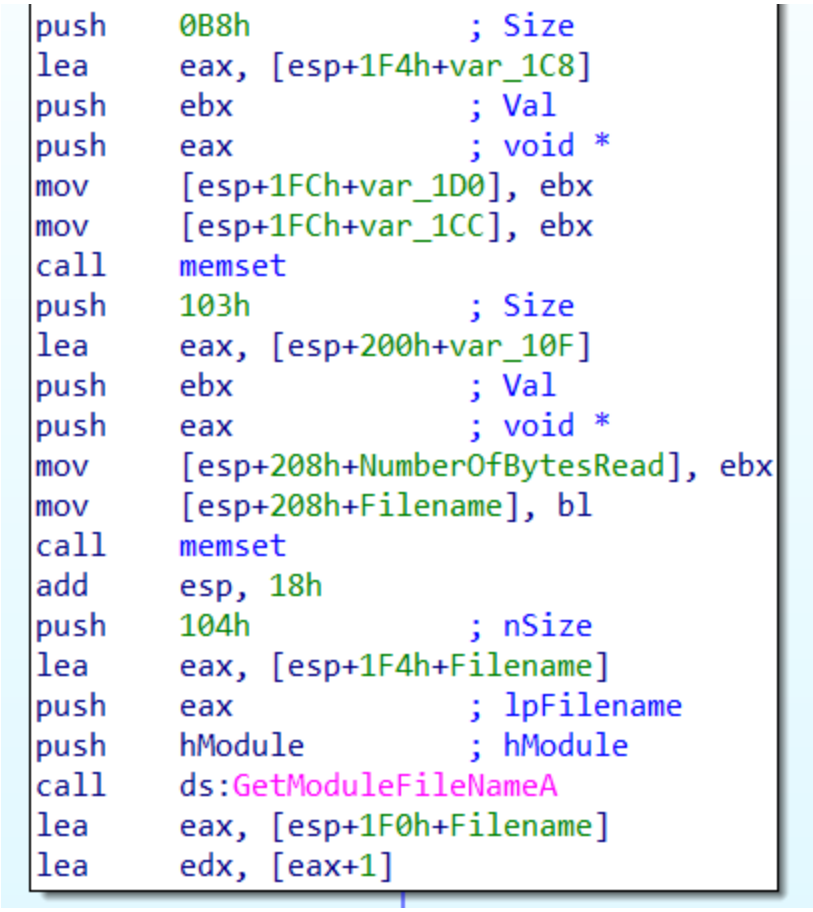


Fig 3.3.2: Call to GetModuleFileNameA

The first significant section of code involves a call to the function GetModuleFileNameA, which will return the full path of the current process being run. In this case, it is expected that the path to MSVCR110.dll will be returned by this function.

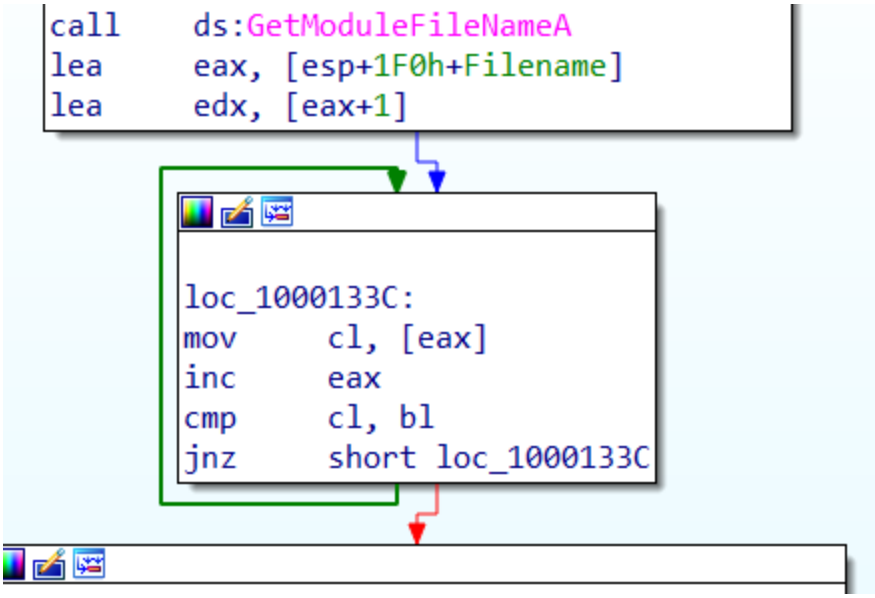


Fig 3.3.3: Loop structure after function call

After the call to GetModuleFileNameA, there is observed to be a loop coding structure which presumably performs certain checks or further processing of the string returned from the function call on a character-by-character basis.

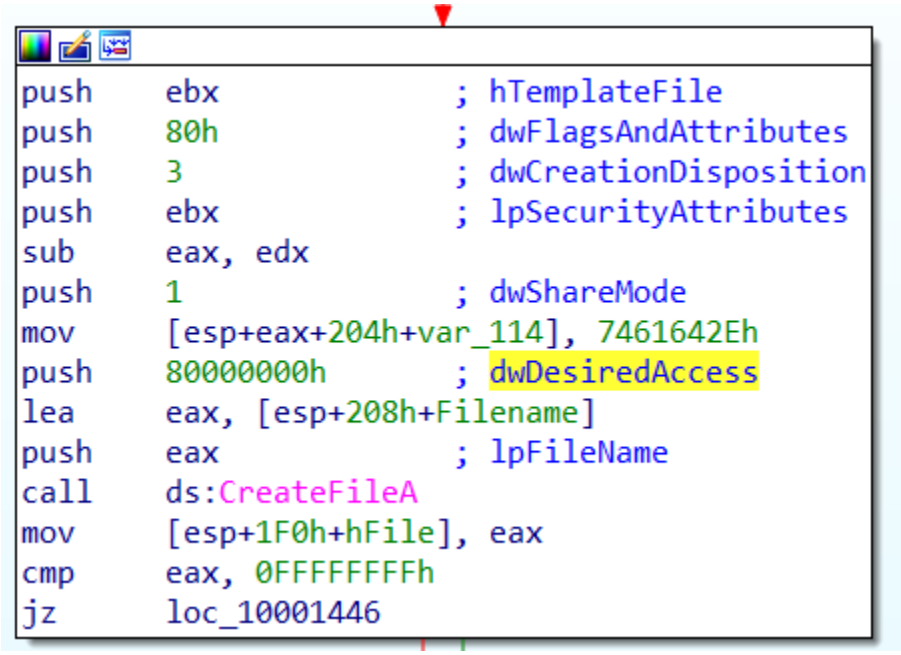


Fig 3.3.4: Call to CreateFileA

The subsequent code block mainly consists of a call to the CreateFileA function along with the pushing of the parameters onto the stack. The following parameters were set:

Parameter	Value	Definition or Usage
hTemplateFile	Specified in ebx register	A valid handle to a template file with the GENERIC_READ access right.
dwFlagsAndAttributes	0x80	The file or device attributes and flags. Current value specifies FILE_ATTRIBUTE_NORMAL, meaning no other attributes are set.
dwCreationDisposition	3	An action to take on a file or device that exists or does not exist, current value will only open a file if it exists, otherwise throwing an error
lpSecurityAttributes	Specified in ebx register	A pointer to a SECURITY_ATTRIBUTES structure, determining whether a created file or device can be inherited by child processes.
dwShareMode	1	Requested sharing mode of the file or device, current value indicates FILE_SHARE_READ, where each subsequent open operation on a file or device can get read access.
dwDesiredAccess	0x80000000	Requested access to the file or device, current value indicates GENERIC_READ, allowing read access.
lpFileName	Found in stack (differs for each call)	Name of the file or device to be created or opened.

Compared to other calls to CreateFileA previously seen within the report, this call appears to have a different function, that is used to get a file handler in order to perform read operations of the data within an existing file.

Furthermore, it can be observed that before the lpFileName parameter is pushed onto the stack, the value 0x7461642E is moved onto a particular position in the stack. When converted back to ASCII characters, it holds the value of ".dat" in little-endian format. This points to a high possibility that the file handler created will be pointing towards the MSVCR110.dat file previously dropped by the malware.

Lastly, there is a compare instruction that checks against whether the CreateFileA function has returned an error, by checking against the value -1 in the register EAX. If the function has failed, it will jump to the end of the function and return to the caller.

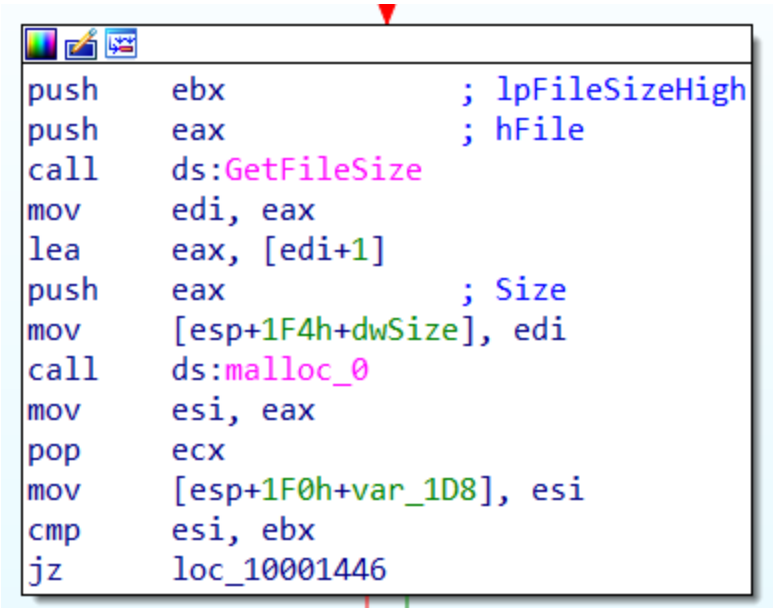


Fig 3.3.5: Calls to GetFileSize and malloc

If the CreateFileA call was successful, the program then proceeds to call the GetFileSize function and the malloc function. This appears to have the purpose of gauging the size of the contents of the MSVCR110.dat file and allocating space within the stack to store the data. Similarly, there is a check for whether the malloc function has returned with a null pointer with the jump-if-zero instruction. If the function has failed, it also jumps to the end of the function, triggering a return to the caller.

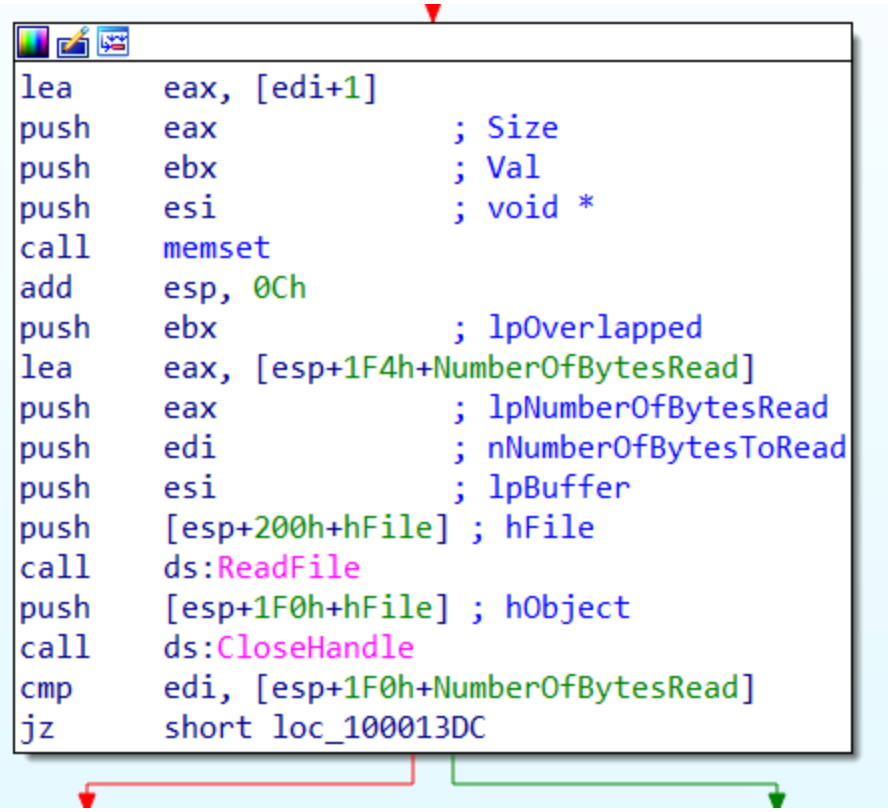


Fig 3.3.6: Calls to ReadFile and CloseHandle

The next block of code executed if the malloc function was successfully run involves calls to ReadFile and CloseHandle. The ReadFile function accepts a dynamically determined number of bytes to read from the file handler pointing to the dropped MSVCR110.dat file. The contents are stored into a specific buffer specified by the value within the ESI register. Lastly, the CloseHandle function is called to release the file handler. The compare instruction checks against whether the correct number of bytes have been read into the buffer, if not, it will trigger a jump to the end of the function.

```
add     esi, 4
lea     edi, [esp+1F0h+var_1D0]
push    eax
lea     ebx, [esp+1F4h+var_1D0]
rep movsd
call    sub_10001239
mov     eax, [esp+1F4h+var_1D0]
mov     ecx, [esp+1F4h+var_1C8]
push    [esp+1F4h+var_178]
mov     ebx, [esp+1F8h+var_1D8]
lea     esi, [eax+ecx]
add     ebx, 0C4h
push    esi
call    sub_100011B0
add     esp, 0Ch
```

Fig 3.3.7: Calls to sub_10001239 and sub_100011B0

In the subsequent code block, there are two points of interest. The program will attempt to call two functions, named sub_10001239 and sub_100011B0 by IDA.

```
; Attributes: bp-based frame fpd=88h

sub_10001239 proc near

var_108= byte ptr -108h
var_F7= byte ptr -0F7h
var_4= dword ptr -4
arg_0= byte ptr  8

push    ebp
lea     ebp, [esp-88h]
sub     esp, 108h
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+88h+var_4], eax
push    esi
push    edi
mov     esi, offset a6e2euigffuw3 ; "%6e2euiGFfuw3&:?"
lea     edi, [ebp+88h+var_108]
```

Fig 3.3.8: Beginning of sub_10001239

The beginning of the first function call to sub_10001239 shows that there is a string “%6e2euiGFfuw3&:?” that is loaded into a register.

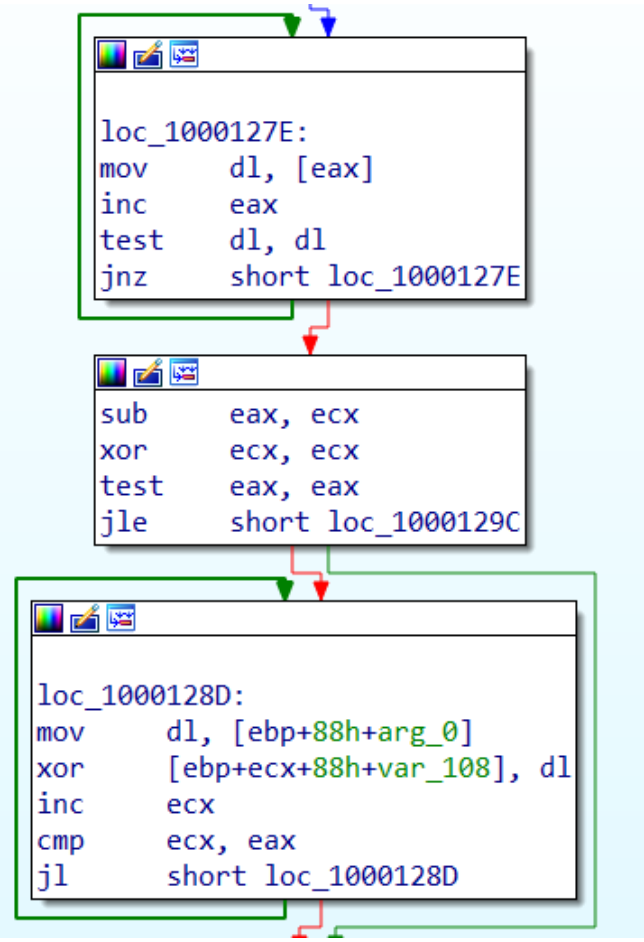


Fig 3.3.9: Loop structure and XOR operations

Subsequently, there are two loop coding structures. The instructions within this section hint at capabilities to perform string manipulation. Particularly, the program appears to perform a XOR operation by individual characters.

Our hypothesis is that the program will attempt to perform XOR operations against the content previously loaded from MSVCR110.dat against the key of “%6e2euiGFfuw3&:?” specified above, as part of a decoding process

```
; Attributes: bp-based frame fpd=88h

sub_100011B0 proc near

var_108= byte ptr -108h
var_F7= byte ptr -0F7h
var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= byte ptr  0Ch

push    ebp
lea     ebp, [esp-88h]
sub     esp, 108h
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+88h+var_4], eax
push    esi
push    edi
mov     esi, offset a6da2aw75UIe ; "6Da2aw7#5<)u+=ie"
lea     edi, [ebp+88h+var_108]
movsd
movsd
movsd
movsd
```

Fig 3.3.10: Beginning of sub_100011B0

Further analysing the structure of the second function called, named sub_100011B0, we notice that it has an overall similar structure, with a key difference being a different value of “6Da2aw7#5<)u+=ie” being defined and set into a register. The loop coding structures and XOR comparisons are also present in this subroutine.

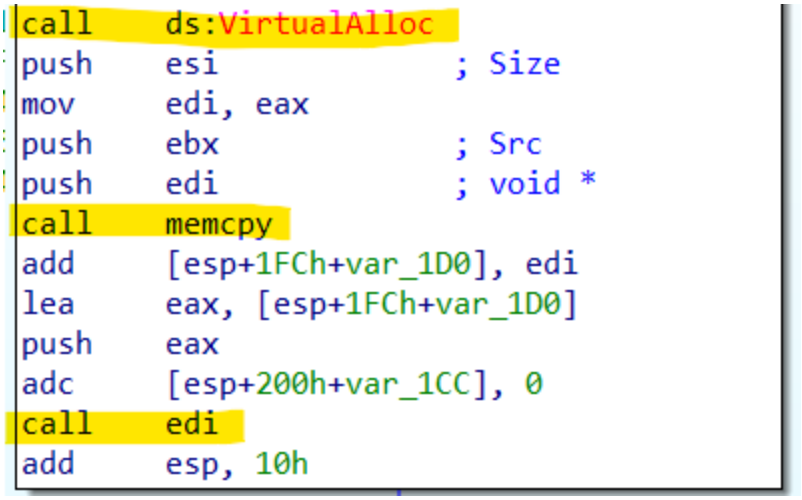


Fig 3.3.11: Call to VirtualAlloc, memcpy and address specified in register EDI

After calls to both sub_10001239 and sub_100011B0 have returned, the program is observed to first call VirtualAlloc. This reserves a region of pages in the virtual address space of the current process, with the memory space automatically initialised to 0.

Next, there is a call to memcpy, which presumably is used to copy the contents of MSVCR110.dat that have been modified into the space reserved by VirtualAlloc. Lastly, a call is made towards a dynamic address stored in the EDI register. The program exits and returns to the caller afterwards.

Overview of Functionality

Considering the functionalities of MSVCR110.dll observed through advanced static analysis, we can reasonably conclude that the primary purpose of this DLL is to load the contents stored within the MSVCR110.dat file dropped by the malware. Correspondingly, we also hypothesise that the contents of MSVCR110.dat are meant to be shellcode that is executed at the end of the program, when the “call edi” instruction is executed. The pre-requisites of this operation are that the contents of MSVCR110.dat are first loaded into memory, decoded, then executed by the program.

At this juncture, we are unable to further analyse the functionality provided by the shellcode within MSVCR110.dat as it is encoded. Additionally, as the eeclnt.exe executable along with MSVCR110.dll and MSVCR110.dat stored in the “C:\Users\exetr\AppData\Roaming\Windows” directory are the same, we have opted against performing further advanced static analysis on these dropped files, assuming that the functionalities are the same.

4. Advanced Dynamic Analysis

4.1 Debugging Main Executable in OllyDbg

In combination with the behaviour of the PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.exe as described in the advanced static analysis section, we also ran the executable under OllyDbg to further ascertain its functionalities.

Fig 4.1.1: Entrypoint of executable

Loading the executable into OllyDbg and running it, it can be observed that the address of the entrypoint is at 0x00DA1574. At this location, there is a call instruction to a function and a jump instruction to another function. Compared with the disassembled code from IDA, these two functions are __security_init_cookie() and __tmainCRTStartup() respectively. We stepped through these two instructions, bringing us into the first instruction within __tmainCRTStartup().

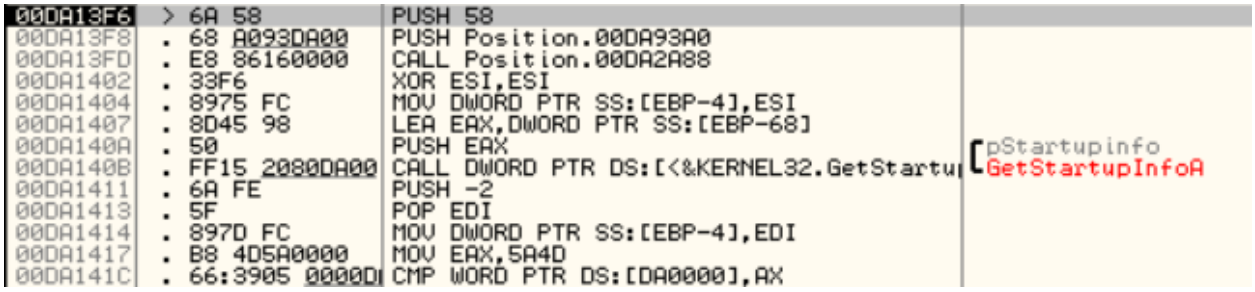


Fig 4.1.2: Beginning of __tmainCRTStartup()

As observed within our static analysis of the disassembled code, the majority of the __tmainCRTStartup() function is automatically generated by the compiler to initialise and set up resources and data structures within the C Runtime library.

Fig 4.1.3: Breakpoint set at call to WinMain()

As such, we set a breakpoint at the call towards the WinMain() function which contains the malicious activity where additional malicious files are dropped onto the system. When the breakpoint was hit, OllyDbg was used to step into the WinMain() function at 0x00DA10000.

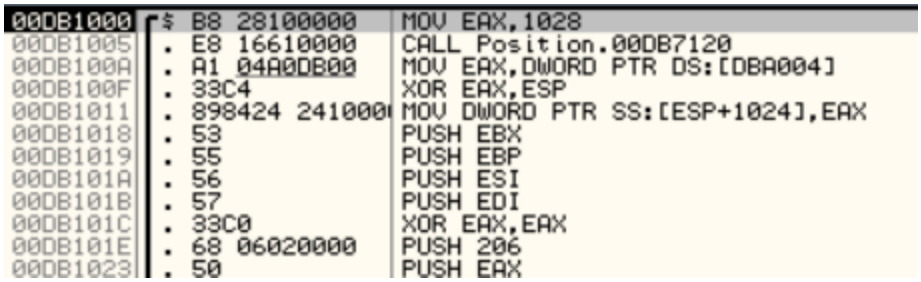


Fig 4.1.4: WinMain() function

Within the WinMain() function, execution of instructions were stepped through and paused at areas where interesting activity was noticed during advanced static analysis of the executable.

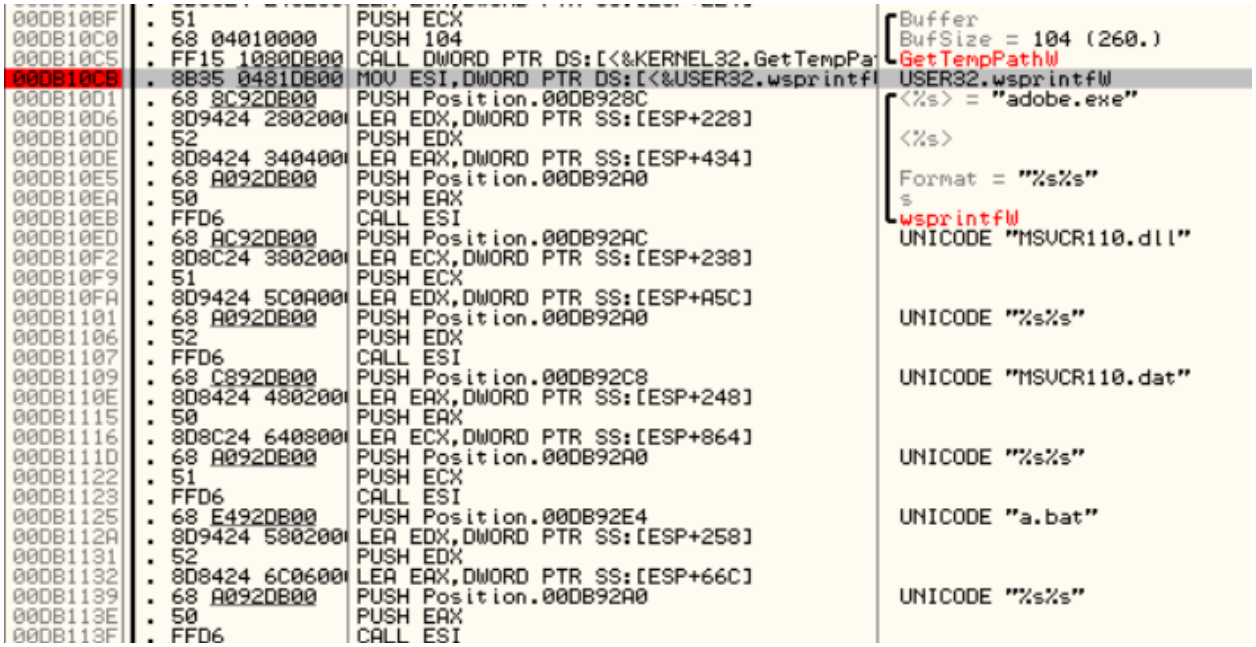


Fig 4.1.5: String operations

The first interesting section of code involves the call to GetTempPathW and several calls to wsprintfW. As previously mentioned, this section of code likely constructs the target file paths for where the malicious files are to be dropped.

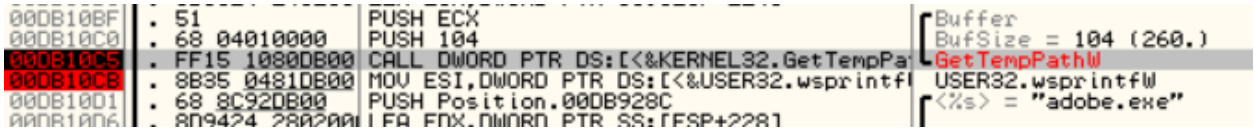


Fig 4.1.6: After calling GetTempPathW

```
004FEBEC  00DB10CB  Position.00DB10CB
004FEBF0  00000104
004FEBF4  004FEE1C  UNICODE "C:\Users\exetr\AppData\Local\Temp\"
004FEBF8  FFFFFFFE
```

Fig 4.1.7: Window temporary path stored onto stack at 0x004FEBF4

Stepping through to after the GetTempPathW is called, it can be observed that the local Windows temporary default path is stored onto the stack. In this case, the specific value observed was “C:\Users\exetr\AppData\Local\Temp” as shown in figure 4.1.7.

```
00DB10D1  . 68 8C92DB00   PUSH Position.00DB928C        <%s> = "adobe.exe"
00DB10D6  . 8D9424 280200 LEA EDX,DWORD PTR SS:[ESP+228]
00DB10DD  . 52            PUSH EDX                      <%s>
00DB10DE  . 8D8424 340400 LEA EAX,DWORD PTR SS:[ESP+434]
00DB10E5  . 68 A092DB00   PUSH Position.00DB92A0        Format = "%s%s"
00DB10EA  . 50            PUSH EAX                      s
00DB10EB  . FFD6          CALL ESI                      wsprintfW
```

Fig 4.1.8: Call to wsprintfW

After which, stepping through the instructions until after the call to wsprintfW, the exact behaviour of how the executable deals with the file paths can be verified.

```
004FEBE8  004FF024  UNICODE "C:\Users\exetr\AppData\Local\Temp\adobe.exe"
004FEBEC  00DB92A0  UNICODE "%s%s"
004FEBF0  004FEE1C  UNICODE "C:\Users\exetr\AppData\Local\Temp\"
004FEBF4  00DB928C  UNICODE "adobe.exe"
```

Fig 4.1.9: Contents of stack after call to wsprintfW

Considering this first instance of the call to wsprintfW, the format specifier is set as “%s%s” which effectively concatenates two strings. The two strings used in this operation are the temporary path provided by the operating system, “C:\Users\exetr\AppData\Local\Temp” and the name of the executable to be dropped, “adobe.exe”. Both strings are loaded onto the stack at 0x004FEBF0 and 0x004FEBF4 respectively. After the wsprintfW function is called, the resulting string is “C:\Users\exetr\AppData\Local\Temp\adobe.exe”, which can be observed being stored onto the stack at 0x004FEBE8 as shown in figure 4.1.9. The result from this operation is that the fully qualified file path for the target destination of the files to be dropped by the malware has been determined.

```
004FEBB4  00DB1141  Position.00DB1141
004FEBB8  004FF22C  UNICODE "C:\Users\exetr\AppData\Local\Temp\a.bat"
004FEBBC  00DB92A0  UNICODE "%s%s"
004FEBC0  004FEE1C  UNICODE "C:\Users\exetr\AppData\Local\Temp\"
004FEBC4  00DB92E4  UNICODE "a.bat"
004FEBC8  004FF434  UNICODE "C:\Users\exetr\AppData\Local\Temp\MSVCR110.dat"
004FEBCC  00DB92A0  UNICODE "%s%s"
004FEBD0  004FEE1C  UNICODE "C:\Users\exetr\AppData\Local\Temp\"
004FEBD4  00DB92C8  UNICODE "MSVCR110.dat"
004FEBD8  004FF63C  UNICODE "C:\Users\exetr\AppData\Local\Temp\MSVCR110.dll"
004FEBDC  00DB92A0  UNICODE "%s%s"
004FEBE0  004FEE1C  UNICODE "C:\Users\exetr\AppData\Local\Temp\"
004FEBE4  00DB92AC  UNICODE "MSVCR110.dll"
004FEBE8  004FF024  UNICODE "C:\Users\exetr\AppData\Local\Temp\adobe.exe"
004FEBEC  00DB92A0  UNICODE "%s%s"
004FEBF0  004FEE1C  UNICODE "C:\Users\exetr\AppData\Local\Temp\"
004FEBF4  00DB928C  UNICODE "adobe.exe"
004FEBF8  FFFFFFFE
```

Fig 4.1.10: Fully qualified file paths for all files to be dropped in stack

This process is repeated three more times, each for the “a.bat”, “MSVCR110.dll" and “MSVCR110.dat”. After the four calls to wsprintfW, all results can be observed to be stored in the stack of the program.

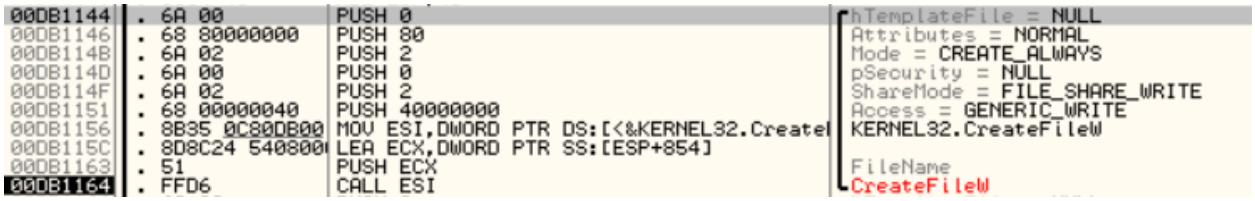


Fig 4.1.11: Call to CreateFileW

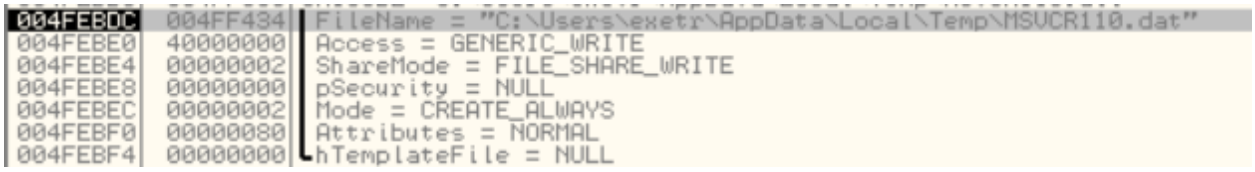


Fig 4.1.12: Parameters for CreateFileW stored in stack

The next significant portion of instruction involves calls to the CreateFileW function. Stepping through to just before the call to CreateFileW instruction is executed, it can be observed that all the parameters to be passed to the function can be found within the stack. Importantly, the filenames are specified as the fully qualified paths previously determined. As shown in figure 4.1.12, the first CreateFileW call will create the file “MSVCR110.dat”

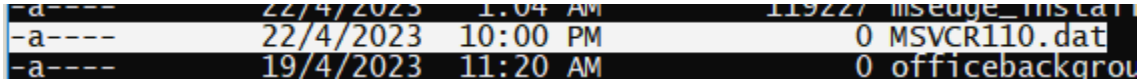


Fig 4.1.13: MSVCR110.dat file created

When the files are first created, it is noticed through figure 4.1.13 that the files will have no content within.

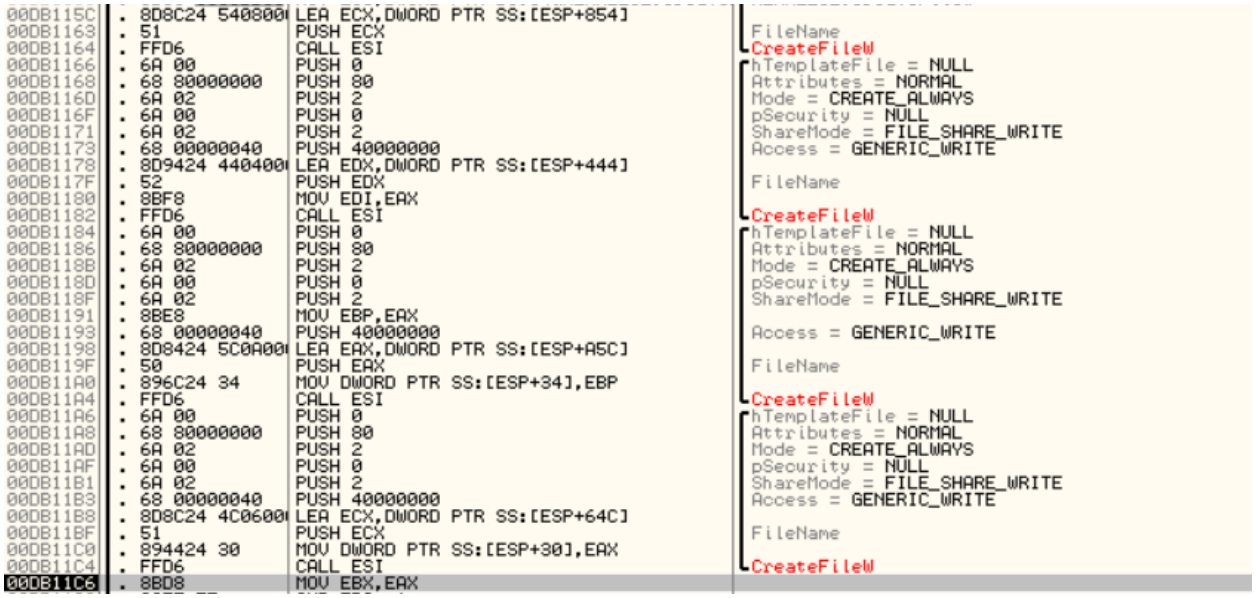


Fig 4.1.14: Four calls to CreateFileW.

Within this group of instructions, it can be seen that there are 4 calls to CreateFileW, each call corresponding to the four files dropped by the malware.

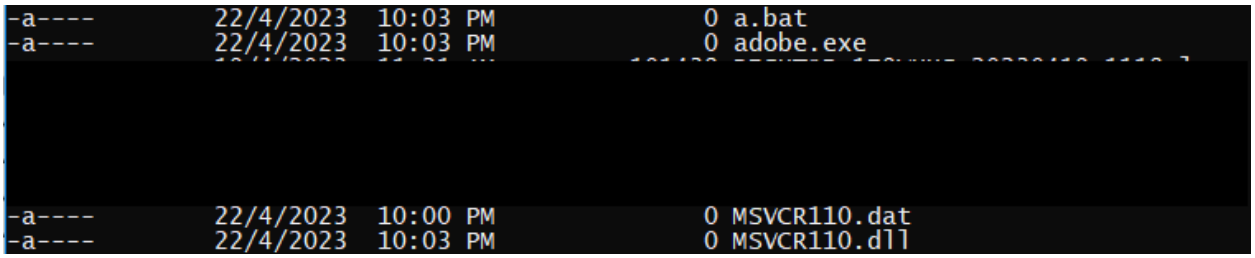


Fig 4.1.15: Empty files created

After the calls to CreateFileW are taken, it can be observed that within the folder “C:\Users\exetr\AppData\Local\Temp\” that there are the files “a.bat”, “adobe.exe”, “MSVCR110.dat” and “MSVCR110.dll” that now exist, all with a size of 0 bytes.

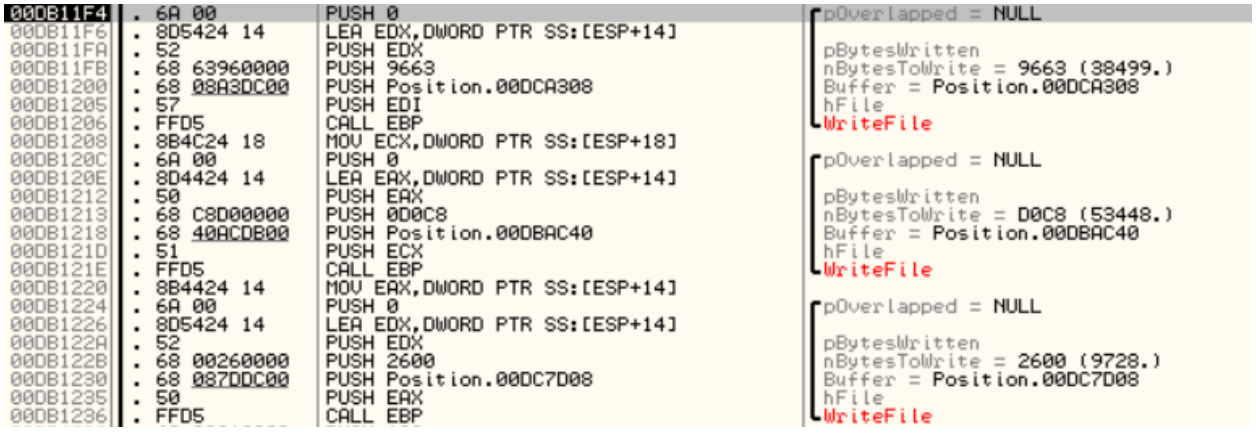
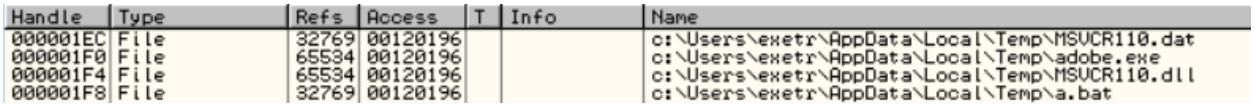


Fig 4.1.16: Three calls to WriteFile.



Handle	Type	Refs	Access	T	Info	Name
000001EC	File	32769	00120196			c:\Users\exetr\AppData\Local\Temp\MSVCR110.dat
000001F0	File	65534	00120196			c:\Users\exetr\AppData\Local\Temp\adobe.exe
000001F4	File	65534	00120196			c:\Users\exetr\AppData\Local\Temp\MSVCR110.dll
000001F8	File	32769	00120196			c:\Users\exetr\AppData\Local\Temp\a.bat

Fig 4.1.17: File Handles

Having created the files, the program now has the four handles to each of the files stored within memory. This is followed by three calls to the WriteFile function.

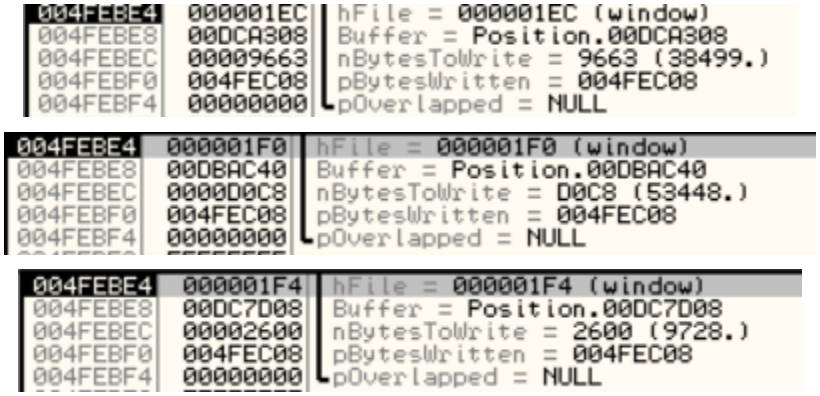


Fig 4.1.18: Options stored into the stack before WriteFile is called.

When WriteFile is called, the values of the parameters passed to the program are stored within the stack for each call. It is noted that for the parameter to specify the file, instead of the absolute path being passed to the function, a file handler is instead passed to the function. In this case, the values 0x1EC (MSVCR110.dat), 0x1F0 (adobe.exe) and 0x1F4 (a.bat) are used as shown in figure 4.1.17 and figure 4.1.18.

The next important section of code addresses the writing of contents to “a.bat” through basic dynamic analysis of the batch file, we note that the batch file when run will loop to delete the original malicious executable in an attempt to cover its tracks.

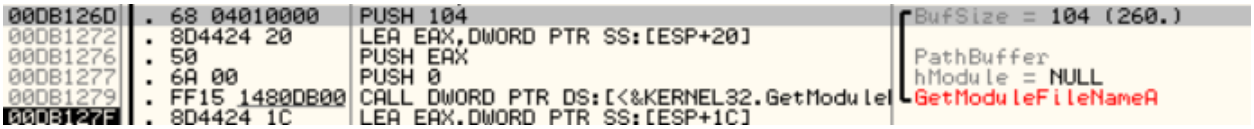


Fig 4.1.19: Call to GetModuleFileNameA

The first part of this process involves determining the filename of the malicious executable. This is achieved by a call to the GetModuleFileNameA, returning the fully qualified path for the currently running module, which in our case is “C:\Users\exetr\Desktop\Malware-sample\Active-Version\PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.exe”. This value is then stored in the stack

Fig 4.1.20: Call to wsprintfA

The second part involves constructing the contents of the batch file. This is achieved similarly using the wsprintfA function call to insert the full path to the malicious executable into the pre-formatted batch file content.

Fig 4.1.21: Parameters stored in stack

The source string is set as ":Repeat\ndel %s\nif exist %s goto Repeat\n". After wsprintfA is run, the string "C:\Users\exetr\Desktop\Malware-sample\Active-Version\PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.exe" will be set into the "%s" format specifiers. The truncated resulting string can be seen being stored in the stack at address 0x004FEBD0.

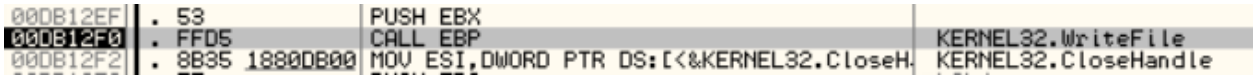


Fig 4.1.22: Call to WriteFile

The last part involves the writing of the created string into the file handler for "a.bat" previously returned by the call to CreateFileW.

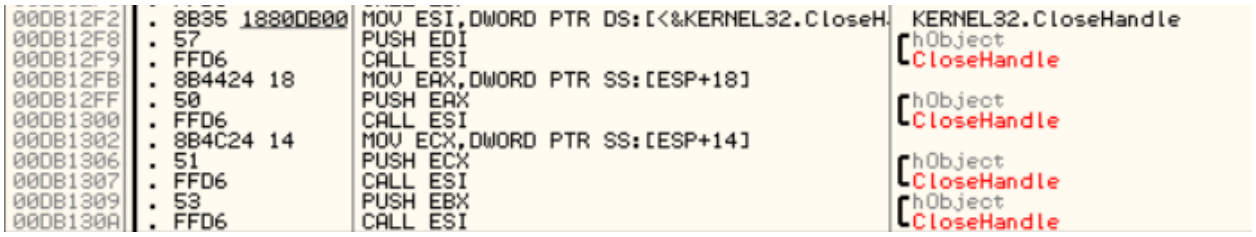


Fig 4.1.23: Calls to Close Handle

At this stage, all four files have been created and the intended contents have been written into the files. This marks the end of the process of the malware dropping additional files onto the filesystem under the PositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598.exe executable. As shown in figure 4.1.22, the CloseHandle is called four times by the program, releasing the open handles referencing each of "a.bat", "adobe.exe", "MSVCR110.dat" and "MSVCR110.dll".

```
-a----        22/4/2023   2:23 AM             53 .ses
-a----        22/4/2023  10:23 PM            303 a.bat




-a----        22/4/2023  10:23 PM          38499 MSVCR110.dat
-a----        22/4/2023  10:23 PM           9728 MSVCR110.dll
```

Fig 4.1.24: Final file sizes of dropped files

Observing the files that are dropped, we can confirm that the sizes of the files match that of the value passed to the WriteFile function call previously.

```
00DB1312  . 6A 00          PUSH 0                          IsShown = 0
00DB1314  . 6A 00          PUSH 0                          DefDir = NULL
00DB1316  . 6A 00          PUSH 0                          Parameters = NULL
00DB1318  . 8D9424 380400  LEA EDX,DWORD PTR SS:[ESP+438]
00DB131F  . 52             PUSH EDX                        FileName
00DB1320  . 68 1893DB00    PUSH Position.00DB9318          Operation = "open"
00DB1325  . 6A 00          PUSH 0                          hWnd = NULL
00DB1327  . FFD3           CALL EBX                        ShellExecuteW
```

Fig 4.1.25: Call to ShellExecuteW

Moving onto the next significant functionality of the executable, it is observed that the program makes a call to the ShellExecuteW function with the “open” operation.

```
003CEDD0  00000000  hWnd = NULL
003CEDD4  00DB9318  Operation = "open"
003CEDD8  003CF214  FileName = "C:\Users\exetr\AppData\Local\Temp\adobe.exe"
003CEDDC  00000000  Parameters = NULL
003CEDE0  00000000  DefDir = NULL
003CEDE4  00000000  IsShown = 0
```

Fig 4.1.26: Parameters passed to stack

As seen from the value of parameters being stored onto the stack, this ShellExecuteW call will result in the “open” operation being run on the dropped executable “adobe.exe”. The functionality of the “adobe.exe” executable will be elaborated in the next section.

```
00DB1329  . 6A 00          PUSH 0                          hTemplateFile = NULL
00DB132B  . 68 80000000    PUSH 80                         Attributes = NORMAL
00DB1330  . 6A 02          PUSH 2                          Mode = CREATE_ALWAYS
00DB1332  . 6A 00          PUSH 0                          pSecurity = NULL
00DB1334  . 6A 02          PUSH 2                          ShareMode = FILE_SHARE_WRITE
00DB1336  . 68 00000040    PUSH 40000000                   Access = GENERIC_WRITE
00DB133B  . 8D8424 380100  LEA EAX,DWORD PTR SS:[ESP+138]
00DB1342  . 50             PUSH EAX                        FileName
00DB1343  . FF15 0480DB00  CALL DWORD PTR DS:[<&KERNEL32.CreateFil  CreateFileA
```

Fig 4.1.27: Call to CreateFileA

After this, the program attempts to perform another call to the CreateFileA function. According to prior advanced static analysis, this is part of the program’s efforts to remove the original malicious executable and replace it with a legitimate word document.

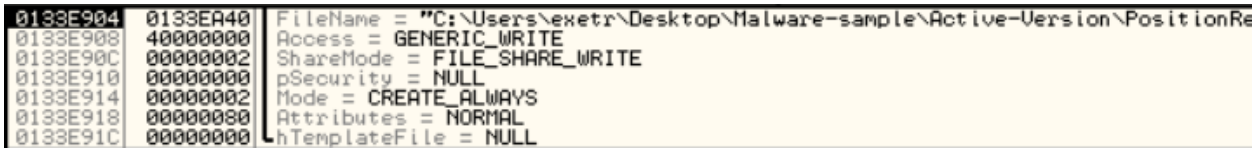


Fig 4.1.28: Parameters passed to stack for call to CreateFileA

Evident from the contents of the stack, the full path for the new file is set as "C:\Users\exetr\Desktop\Malware-sample\Active-VersionPositionRequirement-SeniorCivilEngineer.docx-53c1fd5ac99b5690b278ffcc5a49a598". The directory is the same as where the original malware was stored. The new file contains the same filename as the original, however, the ".exe" extension is now removed.

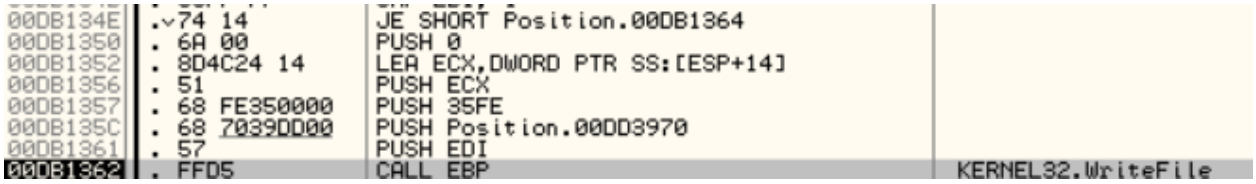


Fig 4.1.29: Call to WriteFile

After the file has been created and the handler is returned by the CreateFileA function, the program proceeds to write content to the file by calling WriteFile. The contents are not clearly seen through OllyDbg, but through analysis in IDA, it was observed that it copies contents that begin with a "PK…" file header. This is indicative of a ZIP archive, but also the format of Microsoft Word documents. Lastly, the CloseHandle function is called to release the file handler.

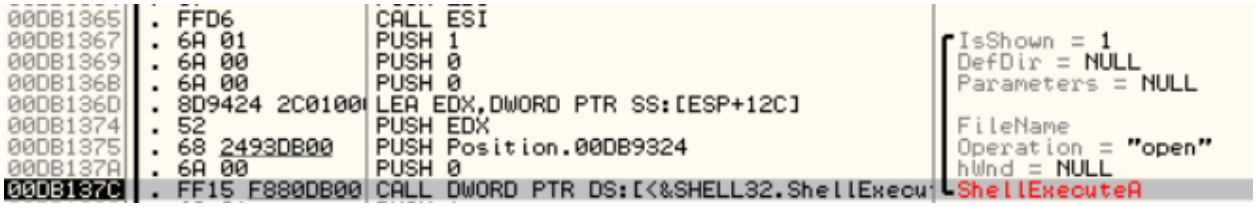


Fig 4.1.30: Call to ShellExecuteA

This is followed by another call to the ShellExecuteA function, similarly with the "open" operation.

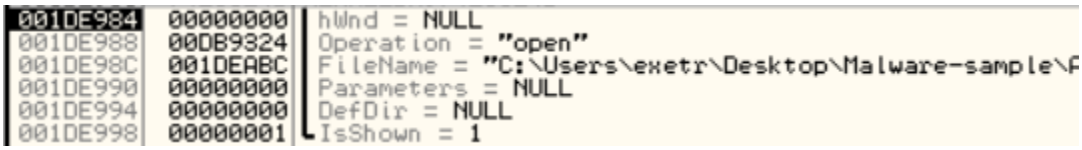


Fig 4.1.31: Contents of stack before call to ShellExecuteA

Observing the contents consisting of the parameter values pushed onto the stack, it can be seen that this ShellExecuteA call will attempt to open the newly created word document. This

can be viewed as an attempt by the malware to disguise its activities as legitimate, by opening a valid word document

Fig 4.1.32: No Association Error thrown

However, as the new file does not have any extensions, when running the program within OllyDbg, an error was thrown highlighting lack of associated program.

Fig 4.1.33: Call to ShellExecuteW

There is also another call to ShellExecuteW, similarly with the “open” operation.

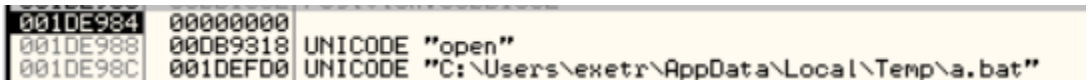


Fig 4.1.34: Contents of stack before call to ShellExecuteW

Analysing the contents of the stack shows that the program will attempt to launch the previously dropped batch file. This batch file will attempt to cover the malware’s tracks by deleting the original malicious executable.

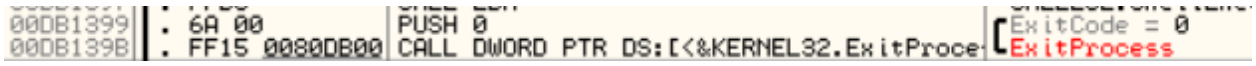


Fig 4.1.35: Call to ExitProcess

Lastly, the final instruction within this executable is a call to the ExitProcess function, with the exit code set to 0. This will cause the program to end, returning a code of 0.

4.2 Debugging adobe.exe and MSVCR110.dll

In performing advanced dynamic analysis of adobe.exe, we have also decided to include analysis into MSVCR110.dll, as the primary malicious calls from adobe.exe are directed towards functions exported by MSVCR110.dll. Firstly, adobe.exe was debugged under OllyDbg.

Fig 4.2.1: Entrypoint to adobe.exe

There was observed to be a call instruction to a function and a jump instruction to a separate function. Compared with the disassembled code from IDA, this corresponds with the instructions observed.

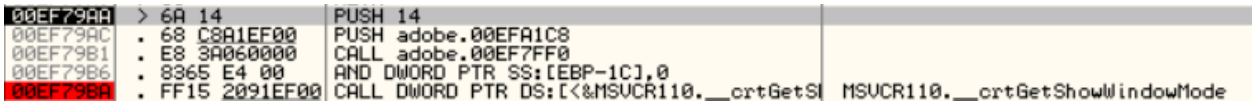


Fig 4.2.2: Call to __crtGetShowWindowMode

Stepping through the jump instruction to 0x00EF79AA, it can be observed that there is a call to a __crtGetShowWindowMode function that is exported from MSVCR110.dll.

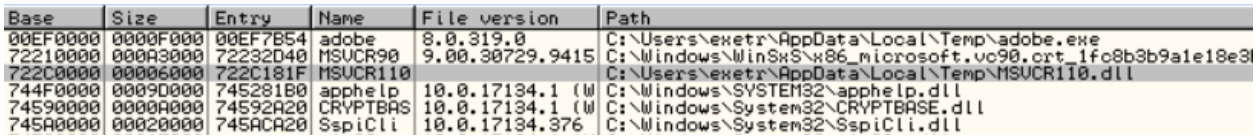


Fig 4.2.3: Path of MSVCR110.dll loaded by adobe.exe

Checking the modules loaded by adobe.exe, we can validate that the MSVCR110.dll that is loaded is the malicious DLL that was previously dropped by the malware. This verifies that the DLL load order was exploited in order to get the legitimate adobe.exe to call malicious code within a malicious DLL.

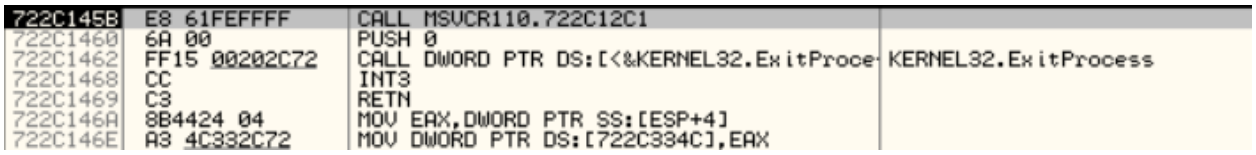


Fig 4.2.4: Entrypoint of __crtGetShowWindowMode

When the call instruction was hit, execution was stepped into the function exported by MSVCR110.dll. Figure 4.2.4 shows the entrypoint when control is handed off to the DLL.

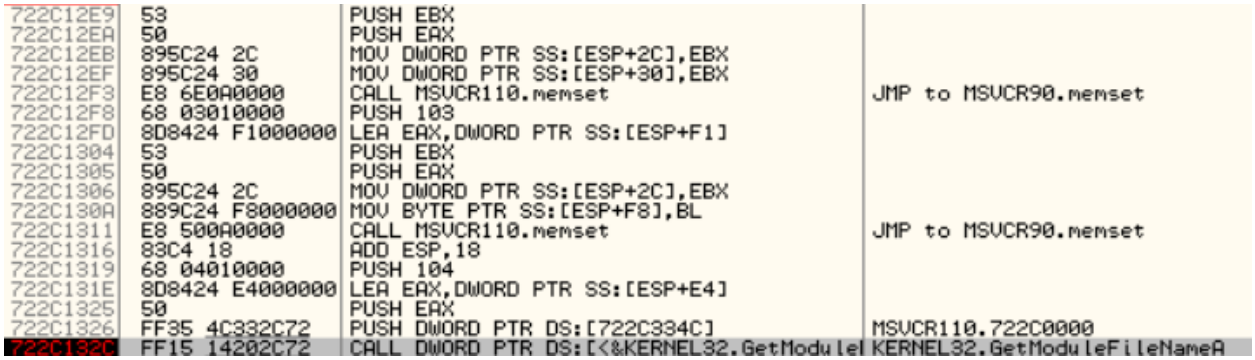


Fig 4.2.5: Call to GetModuleFileNameA

Stepping through this function, the subsequent point of interest involves the call to the GetModuleFileNameA highlighted during advanced static analysis of the DLL.

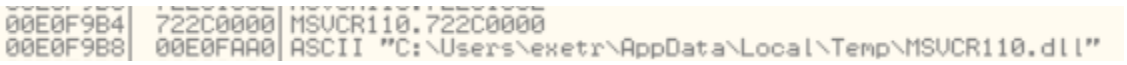


Fig 4.2.6: Value returned by GetModuleFileNameA in stack

After executing the call to function, it can be observed that the fully qualified path to the current process (MSVCR110.dll) is stored into the stack.

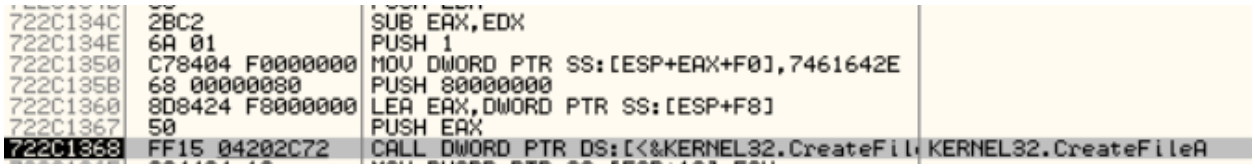


Fig 4.2.7: Call to CreateFileA

A subsequent call to CreateFileA is observed. According to prior advanced static analysis, this function is meant to return a file handler to read from MSVCR110.dat.

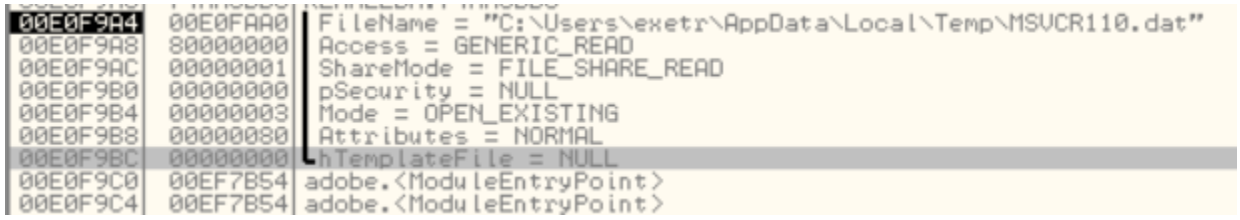


Fig 4.2.8: Parameter values stored in stack

Observing the contents of the stack which include parameters to be passed to CreateFileA, we can verify the behaviour of reading contents from the file handler pointing to MSVCR110.dat.

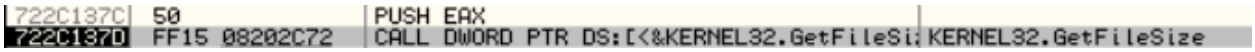


Fig 4.2.9: Call to GetFileSize

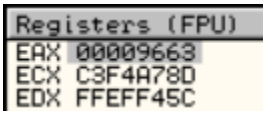


Fig 4.2.10: Results of GetFileSize

Next, there was a call to GetFileSize, which was observed to return the value of 0x9663 or 38499 in decimal. This matches the size of MSVCR110.dat, which is 38499 bytes.

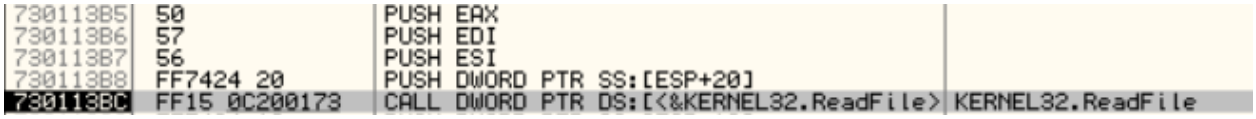


Fig 4.2.11: Call to ReadFile

The next significant function call was towards ReadFile. This will attempt to read the contents of MSVCR110.dat

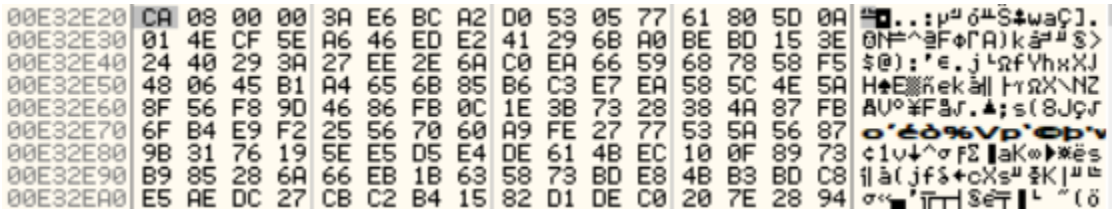


Fig 4.2.12: Contents of MSVCR.dat stored into memory

The figure above shows the contents of MSVCR110.dat being stored into the memory space of the current process.

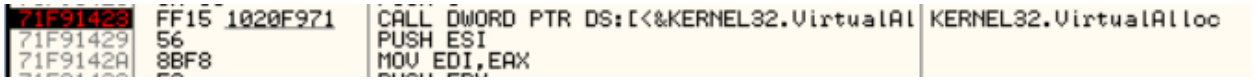


Fig 4.2.13: Call to VirtualAlloc

The next breakpoint was set at the instruction involving a call to VirtualAlloc.

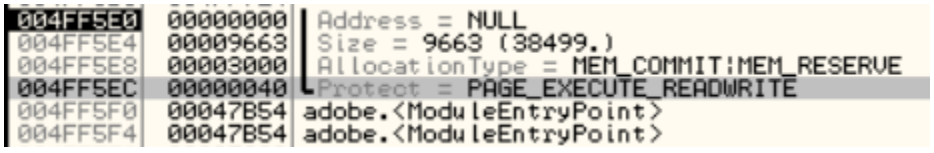


Fig 4.2.14: Parameters passed to VirtualAlloc

Observing the parameters to be passed to the function, it declares a size of 0x9663, matching the size of MSVCR110.dat, and sets the permissions to allow reading, writing and executing of data within this section of memory.

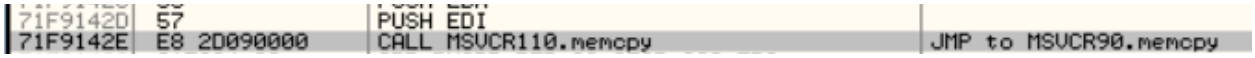


Fig 4.2.15: Call to memcpy

The next call to memcpy is designed to copy the decoded contents of MSVCR110.dat into the current process' memory space. Along with the executable permission set, it is reasonable to assume that this section of memory is meant to host shellcode.

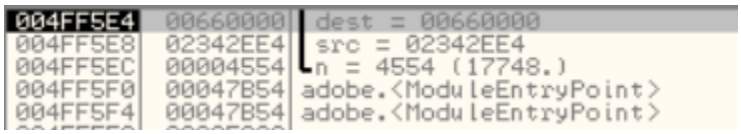


Fig 4.2.16: Parameters passed to memcpy, noting different value of size (0x4554)

Observing the parameters to be passed to memcpy, notably it is observed that the size of contents copied to the memory space (0x4554) is lesser than that of the declared size of the space (0x9663).

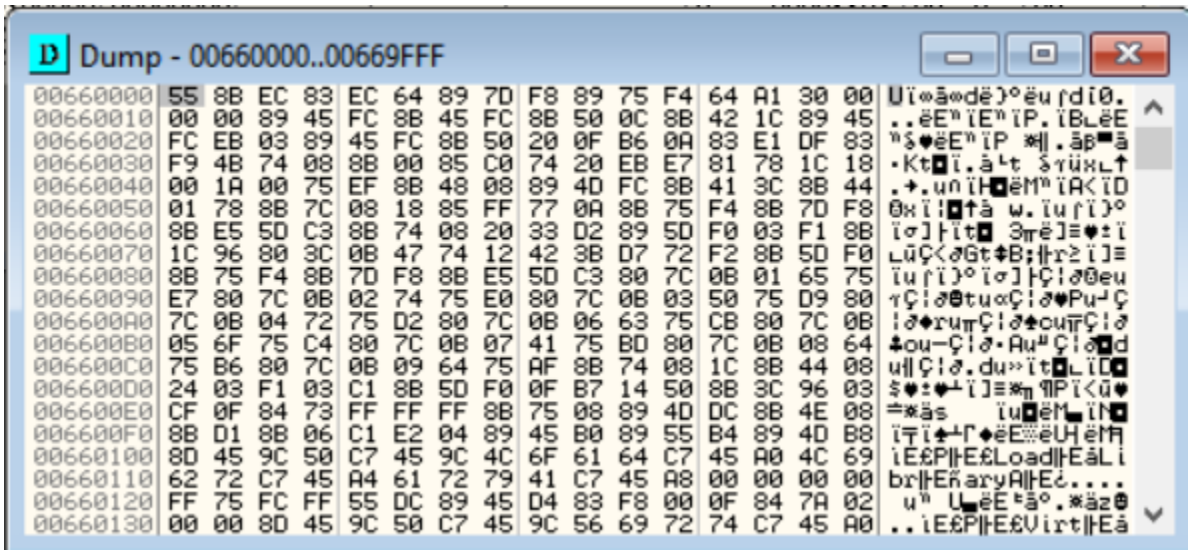


Fig 4.2.17: Contents copied to memory space at 0x00660000

After executing the memcpy call, analysing the memory space previously allocated at 0x00660000, we are able to observe what seems like shellcode stored in a section of memory that is marked as executable.

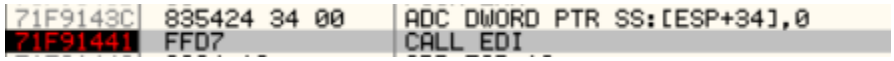


Fig 4.2.18: Call to address stored in EDI

The final significant function call within MSVCR110.dll involves a call to an address stored in the EDI register.

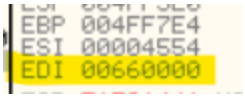


Fig 4.2.19: Value stored in EDI register

Before the call instruction is taken, it was observed that the value in EDI is currently 0x00660000. Thus, executing the call instruction would pass control of execution to the instructions located in the previously allocated memory space.

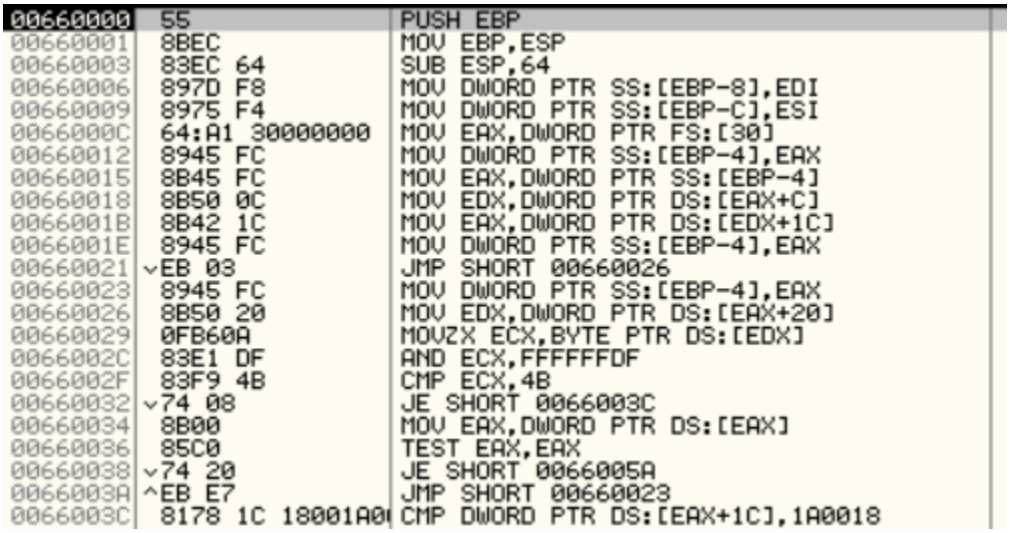


Fig 4.2.20: Disassembled opcodes stored at 0x00660000

When stepping into this region of memory, it can be observed that the bytes stored within the memory are shellcode with valid operations and instructions.

4.3 Debugging Shellcode Contained in MSVCR110.dat

We were unable to effectively debug the full contents and functionalities of the shellcode contained in MSVCR110.dat due to multiple difficulties faced during the process. However, our efforts were able to uncover a number of potential functionalities contained within the shellcode.

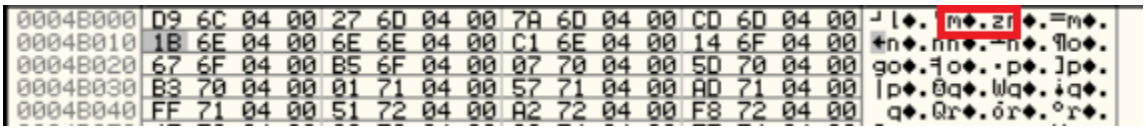


Fig 4.3.1: Existence of MZ header within .data section

Firstly, within the .data section of the shellcode when executing, it was observed that it stores bytes that begin with “MZ”. This hints at the existence of a DOS executable being stored within the .data section.

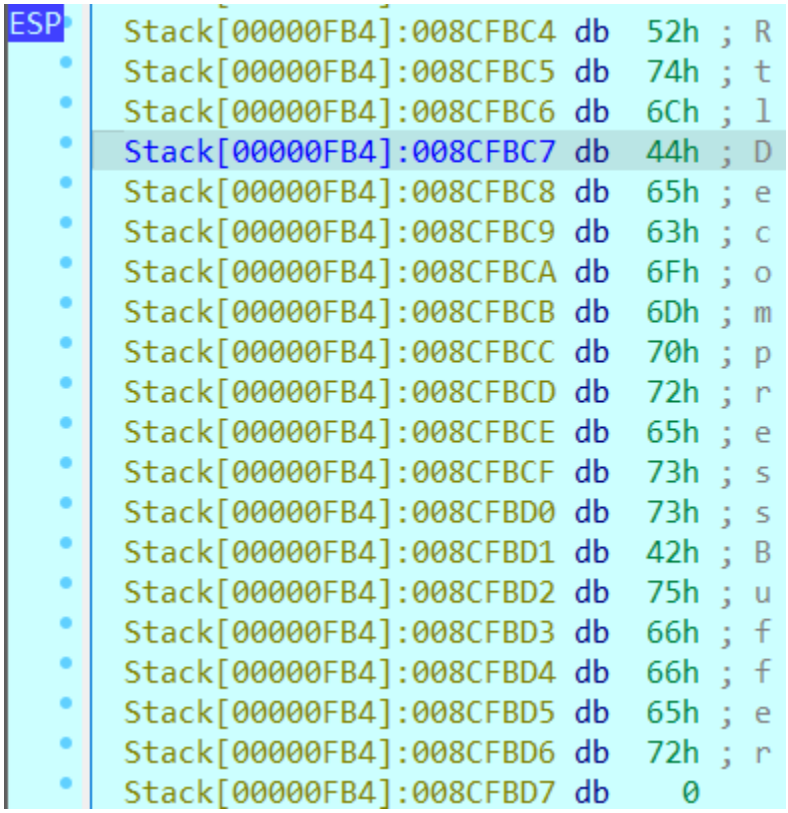


Fig 4.3.2: Function names being stored in the stack

Furthermore, initial stepping through of the instructions within the shellcode show that there were multiple instances of function names being pushed onto the stack. Figure 4.3.2 above depicts one example, showing the value “RtlDecompressBuffer” being pushed onto the stack. Firstly, this indicates that the shellcode contains capabilities to dynamically load additional libraries and functions.

Secondly, the specific mention of the RtlDecompressBuffer function is also significant. This is because it has been previously documented that PlugX malware variants utilise this function to decompress PE files that are stored within the memory with the LZ decompression format. It is highly likely that from this decompressed PE file that the malware performs several actions.

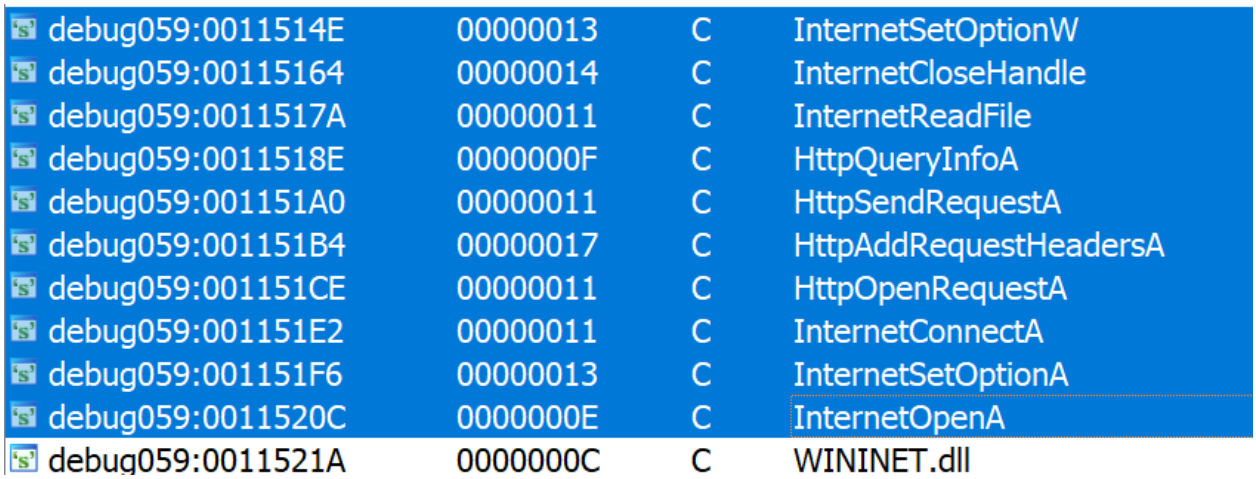


Fig 4.3.3: Internet related functions imports

Furthermore, observing the functions imported by the shellcode also supports the malware's capability of performing actions over the network.

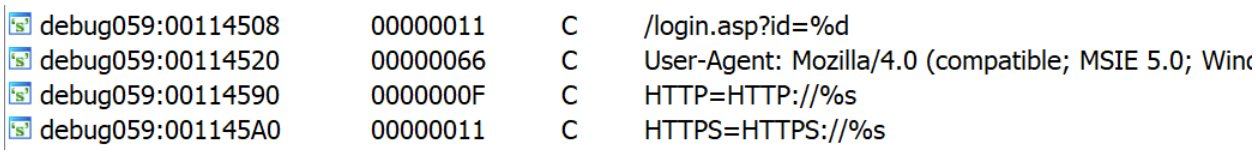


Fig 4.3.4: Content in stack relating to HTTP requests

While we were able to detect outgoing DNS requests during basic static analysis, we were not able to detect any outgoing HTTP requests that could have been sent by the malware back to a potential command and control server. However, observing the contents that are eventually pushed onto the stack by the shellcode, we can further ascertain certain behaviour:

- A HTTP or HTTPS request will be sent to the domain news.singmicrosoft.ga
- The requested path will be /login.asp
- The parameter id with a value of a decimal will be appended to the request URL
- The user agent will be set as Mozilla/4.0 from a Windows device

5. Other Observations and Inferences

Although we were unable to make observations relating to the spawning of the eeclnt.exe and the running of the executables with certain arguments through the process of advanced dynamic analysis, we are fairly confident that the suspicious behaviour originates from instructions held within the decompressed PE executable as stored within the shellcode.

Additionally, we were also unable to pinpoint the exact forms of information exfiltrated by the malware, as we similarly feel that the relevant logic and instructions are contained within the shellcode. Furthermore, we are unable to verify any outgoing HTTP requests generated by the malware.

We also believe that further analysis of the decompressed PE executable could provide further insights into how the malware handles persistence of the different executables that are dropped by the program.

6. Conclusion

Overall, we have ascertained the malware to possess, among others, these notable functionalities:

1. Double file extension (.docx.exe), coupled with the Microsoft Word Document icon to trick users into thinking it is a legitimate file, resulting in users clicking and unintentionally running the malware
2. The .docx.exe file essentially serves as a dropper as it creates and drops malicious files such as a.bat, adobe.exe, MSVCR110.dll and MSVCR110.dat in specified filepaths
3. This is characteristic of DLL side-loading, as it results in the malicious MSVCR110.dll being invoked as opposed to the actual and benign MSVCR110.dll which is found under the \Windows\System32 directory
4. To mask this malicious activity, after the primary executable is run, a new file is created, with the exact same name except for the .exe extension, while the .docx.exe file is deleted. Additionally, this new file created is a legitimate word document and it will also be opened so as not to raise undue suspicion
5. Persistence is gained after running eeclnt.exe with command-line argument 260, by registering the eeclnt program under the registry key Software\Microsoft\Windows\CurrentVersion\Run
6. Hiding adobe.exe as eeclnt.exe along with a copy of MSVCR110.dll and MSVCR110.dat in a hidden folder
7. Execution of eeclnt.exe with several different arguments
8. Attempting to connect to news.singmicrosoft.ga to exfiltrate information collected from the system