Challenge: MalaCrypt Lab

Platform: CyberDefenders

Category: Malware Analysis

Difficulty: Medium

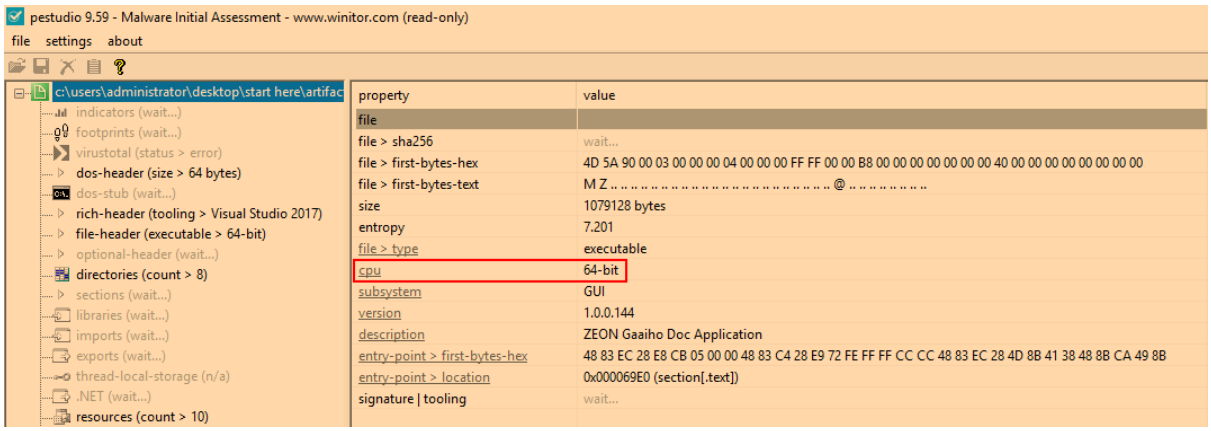
Tools Used: PEStudio, Floss, Strings, CyberChef, ProcMon, Cutter, Capa, VirusTotal

Summary: This lab involved investigating a malware sample and focused on practising basic to intermediate static and dynamic analysis of a PE file. It required collecting basic information about the sample, including its architecture, original filename, suspicious libraries, and strings. The dynamic analysis component involved executing the sample and using tools such as ProcMon to observe malware behaviour, including registry activity and image load events. Basic reverse engineering was also performed on the sample through the use of a disassembler; in this case, a combination of Capa and Cutter was used to effectively analyse suspicious code within the file. I recommend giving this lab a shot if you are relatively new to learning malware analysis and wish to practice.

Scenario: During routine monitoring at MalaCrypt company, a suspicious binary named "malware.exe" was found on a device. Initial checks of its hash values against threat intelligence platforms yielded no results, suggesting the attacker may have altered the malware to evade detection. As a security analyst, the next action is to investigate further, using alternative methods beyond hash-based detection, considering that attackers often modify hashes to bypass initial security checks.

Understanding the type of binary architecture allows us to determine the types of registers being used. What architecture is this binary built for?

Each CPU has its own instruction set architecture (ISA) which is the language a CPU understands. To find the architecture of this binary, we can use a static-analysis tool called PEStudio:



We can see that this binary is for 64-bit CPUs, i.e., CPUs which support a x64 architecture.

Answer: x64

Executables are sometimes renamed or altered to evade detection or disguise their true purpose. What is the original name of the executable?

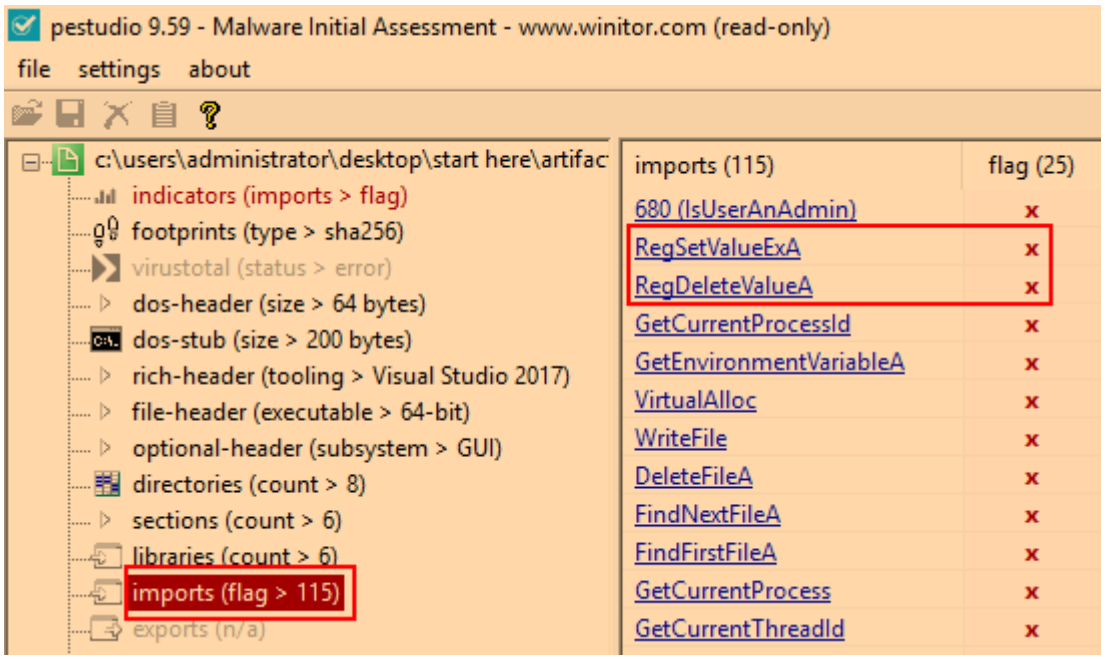
Within Portable Executable (PE) files is a specific field in the file's version information resource called the original filename. As the name implies, this stores the file's name as it was at compile time. Using PEStudio, we can see that the original filename was "DefaultViewer.exe":

version > original-file-name	DefaultViewer.exe

Answer: DefaultViewer.exe

Some DLL files are responsible for accessing Windows registries. Which DLL file is utilized to manipulate the Windows Registry?

One incredible feature of PEStudio is its ability to flag suspicious imports and strings. If you navigate to the imports section and look at the flagged imports, we can see references to RegSetValueExA and RegDeleteValueA:



Upon viewing the Microsoft documentation, we can determine that RegSetValueExA is used to create registry keys, and RegDeleteValueA is used to remove a value from registry. If you look at the library column, we can see that both functions are imported from the ADVAPI32.dll library:

imports (115)	flag (25)	first-thunk-original (INT)	first-thunk (IAT)	hint	group (0)	technique (12)	type (5)	ordinal (1)	library (2)
680 (IsUserAnAdmin)	x	0x80000000000002A8	0x80000000000002A8	0 (0x0000)	security	-	implicit	x	SHELL32.dll
RegSetValueExA	x	0x000000000003C64A	0x000000000003C64A	680 (0x02A8)	registry	T1112 | Modify Registry	implicit	-	ADVAPI32.dll
RegDeleteValueA	x	0x000000000003C624	0x000000000003C624	626 (0x0272)	registry	T1485 | Data Destruction	implicit	-	ADVAPI32.dll

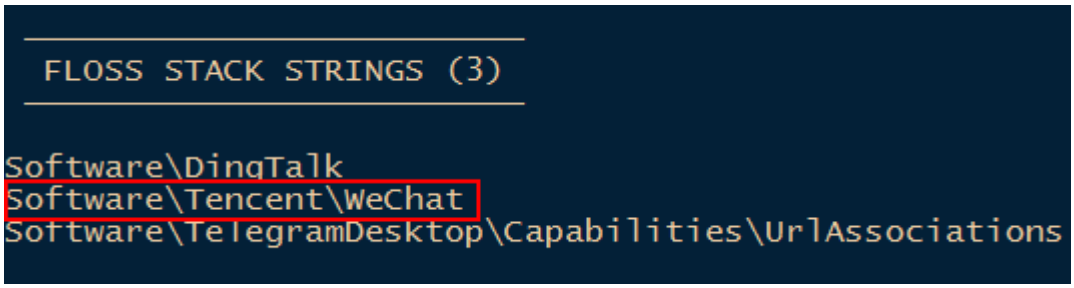
Answer: ADVAPI32.dll

Certain strings may reveal specific information. What is the name of the Chinese messaging app discovered in the basic static analysis?

Whilst there are many tools you can use to extract strings, a popular one is called FLOSS, which is a tool developed by Mandiant that extracts and deobfuscates all strings from binaries. Basic usage is simple:

- `floss <binary_to_analyse>`

After Floss has completed analysing the binary, we can find an interesting stack string:



Stack strings are created during the program's run-time and are stored temporarily on the stack. In this case, we can see a stack string form "WeChat", which is a popular Chinese multi-purpose app, with features like messaging, social media, and much more.

Answer: WeChat

The Windows API can be used for malicious purposes. Which Windows API is used to destroy previously generated encryption keys?

Navigating back to flagged functions in PEStudio, we can find a series of cryptographic related functions all imported from ADVAPI32.dll:

CryptDestroyKey	x	0x000000000003C6B8	0x000000000003C6B8	200 (0x00C8)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll
CryptDecrypt	x	0x000000000003C690	0x000000000003C690	197 (0x00C5)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll
CryptCreateHash	x	0x000000000003C67E	0x000000000003C67E	196 (0x00C4)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll
CryptDeriveKey	x	0x000000000003C66C	0x000000000003C66C	198 (0x00C6)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll
CryptHashData	x	0x000000000003C65C	0x000000000003C65C	217 (0x00D9)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll
CryptDestroyHash	x	0x000000000003C636	0x000000000003C636	199 (0x00C7)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll
CryptReleaseContext	x	0x000000000003C60E	0x000000000003C60E	220 (0x00DC)	crypto	T1027 | Obfuscated Files or Information	implicit	-	ADVAPI32.dll

The one that stands out is CryptDestroyKey, which frees a cryptographic key handle, making it invalid for further use.

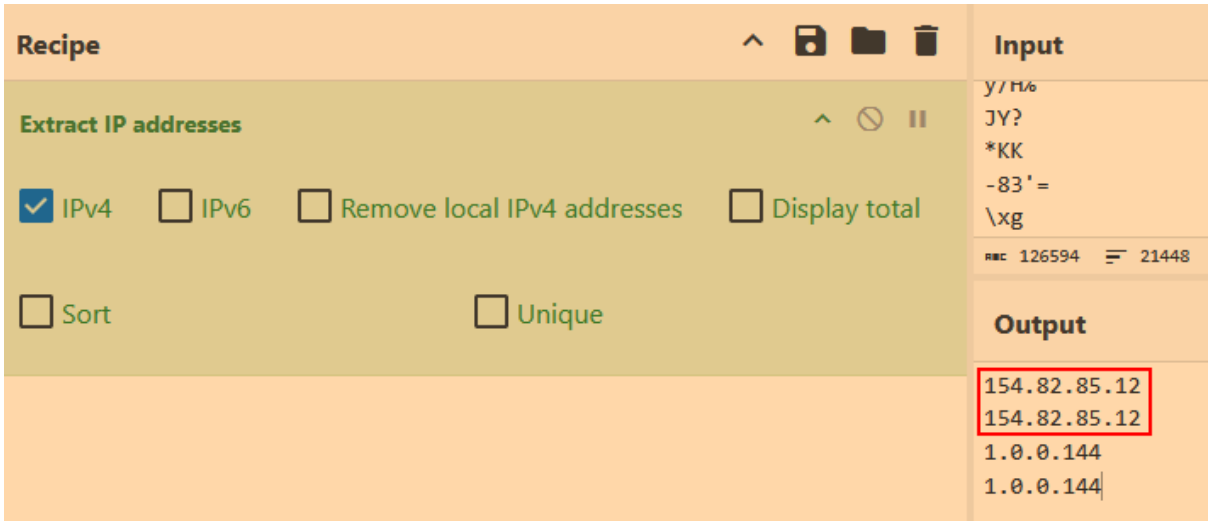
Answer: CryptDestroyKey

Knowing the attacker's IP can help trace the source of the attack and gather information about their location and network. What IP address is found in the executable that belongs to Hong Kong?

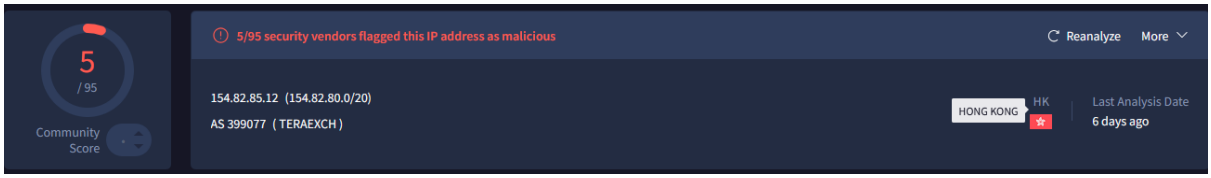
To extract IP addresses from this binary, I am going to use a combination of the strings command and CyberChef:

- `strings .\malware.exe > strings.txt`

Using the Extract IP addresses recipe in CyberChef, we can find one IP address:



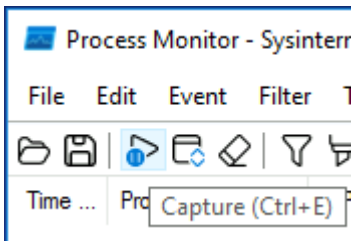
Upon submitting this IP on VirusTotal, we can see that it geolocates to Hong Kong:



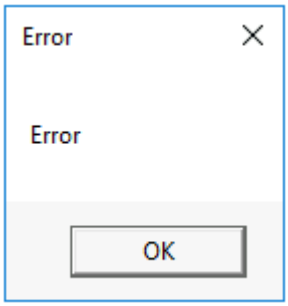
Answer: 154.82.85.12

In dynamic analysis, we examine the behavior of the malware and identify any suspicious activities, What message is displayed on the screen when the binary is executed?

To start dynamical analysis on this sample, I am going to use a Sysinternals tool called Process Monitor (ProcMon). Make sure that the capture is enabled before executing the binary:



When executing this binary, we can immediately see a popup that displays “Error”:

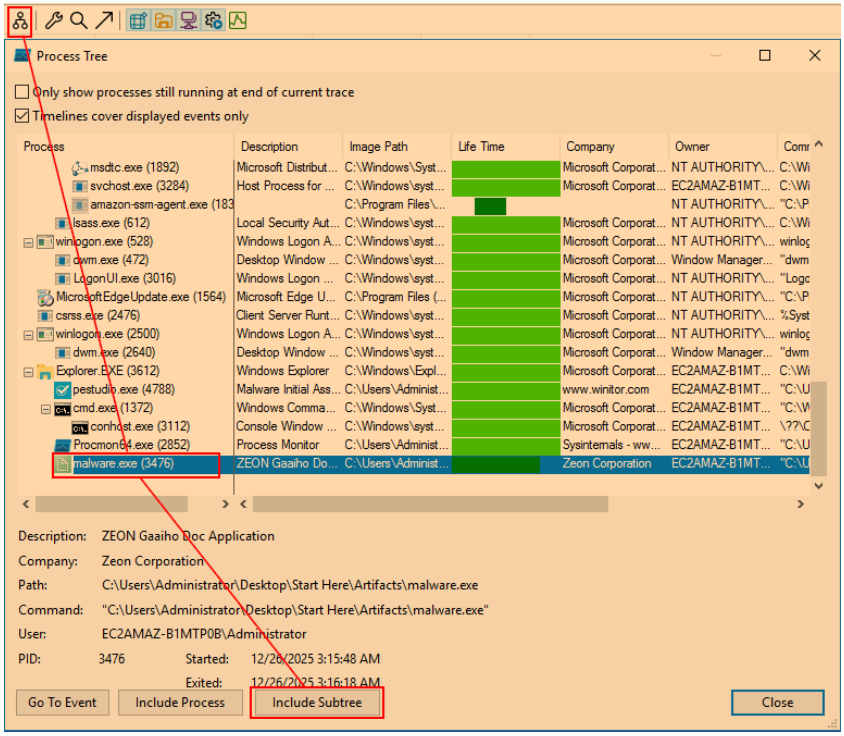


After the binary has finished execution, make sure to stop the capture in ProcMon.

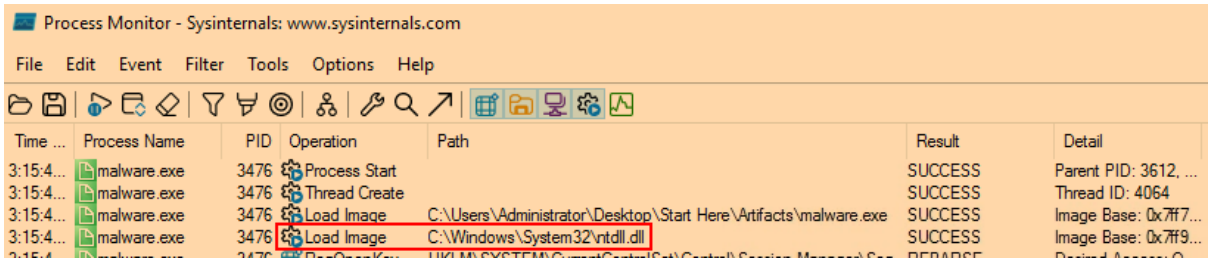
Answer: Error

Identifying the executed DLLs gives us insight into the attacker's strategies and goals. What is the name of the first DLL file that is loaded after the binary is executed?

To reduce the noise from other processes running on the system, we can filter for events related to this malware sample by clicking the Process Tree button, Selecting the binary, and clicking Include Subtree:



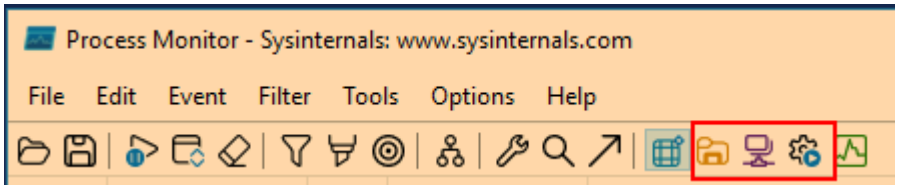
Immediately you will find Load Image events, the first being ntdll.dll:



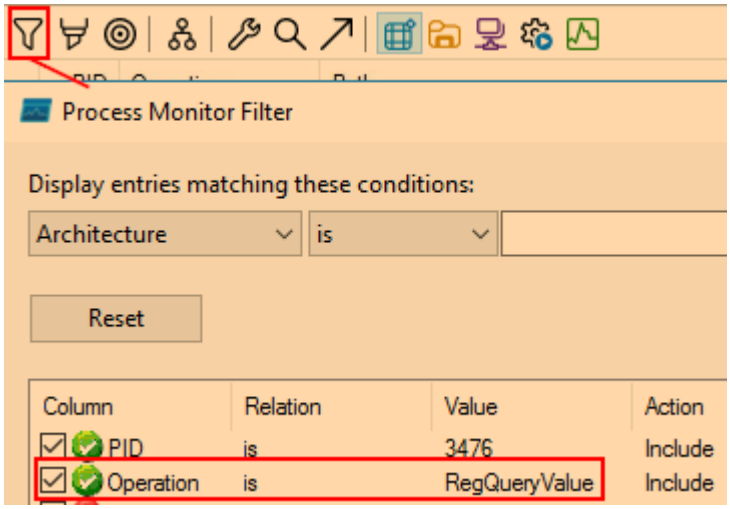
Answer: ntdll.dll

Registry enumeration involves listing all the keys and values in the Windows Registry that a process has accessed to understand its structure and contents. What is the full path of the registry key associated with fallback handling in language packs that was successfully enumerated?

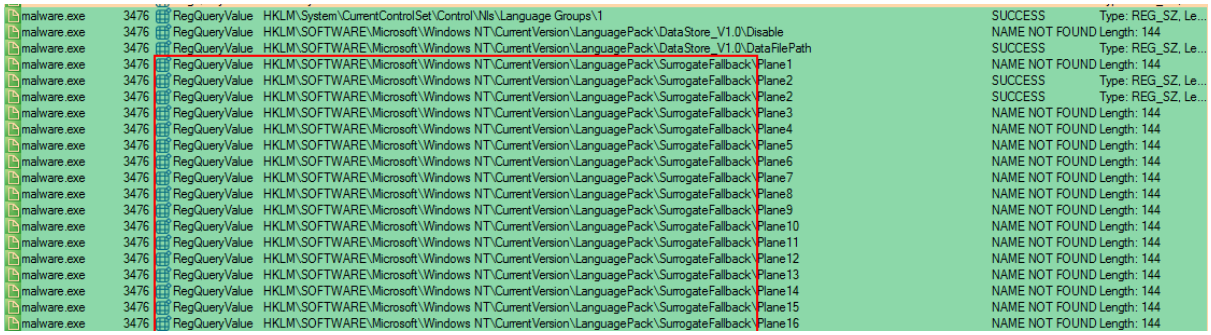
Let's start by limiting the results in ProcMon to only show registry events, we can do so by unselecting these three boxes and making sure the only option enabled is for registry events:



To reduce the noise even further, select the filter button and filter for the RegQueryValue operation:



Scrolling through the results, we can see that the malware queried HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\SurrogateFallback multiple times:

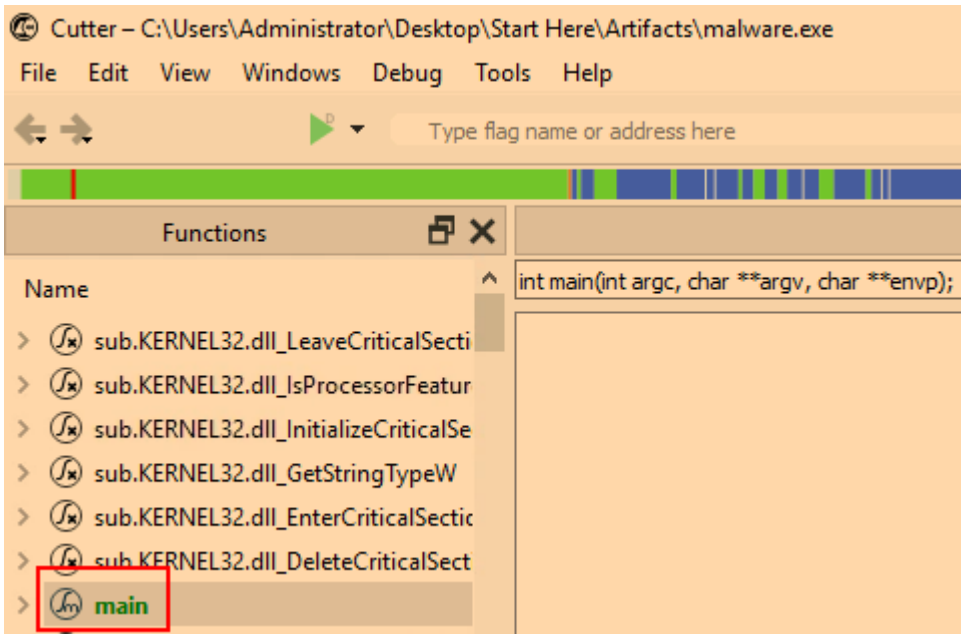


This registry key holds settings for how Windows handles surrogate characters when a specific font isn't available.

Answer: HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\SurrogateFallback

Tracing Windows API calls helps understand what the malware is intended to do by analyzing specific patterns or arguments used in these calls. What is the first Windows API call made from the function that is called from the main function?

To determine the first API call in the main function, we need to disassemble the malware. Cutter is a free and open-source reverse engineering tool that we can use for this purpose. Once cutter has loaded the binary, locate the main function on the left-hand side pane:



```
[0x140004690]
int main(int argc, char **argv, char **envp);
0x140004690      sub     rsp, 0x28
0x140004694      call    fcn.140003d9c ; fcn.140003d9c
0x140004699      int3
```

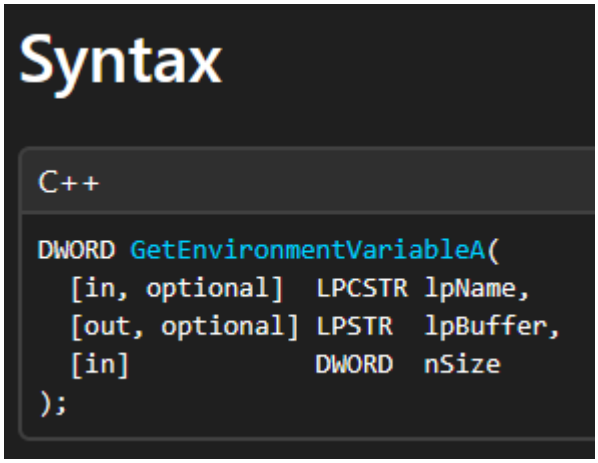
This is a very small function that calls another function, meaning it will execute the code within the address supplied to call. If you double click "0x140004694", it will take us to the function that gets executed. Within this function, we can see that the first API call made is GetEnvironmentVariableA:

```
call    qword [GetEnvironmentVariableA]
```

Answer: GetEnvironmentVariableA

Understanding the number of arguments passed helps identify the data type being transmitted or processed. How many arguments does the API mentioned in the previous question accept?

Viewing the documentation for "GetEnvironmentVariableA", we can see that it accepts three arguments:



Those being lpName, lpBuffer, and nSize:

Parameters

`[in, optional] lpName`

The name of the environment variable.

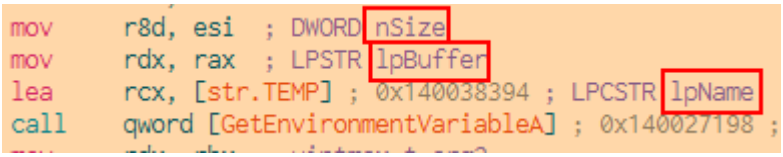
`[out, optional] lpBuffer`

A pointer to a buffer that receives the contents of the specified environment variable as a null-terminated string. An environment variable has a maximum size limit of 32,767 characters, including the null-terminating character.

`[in] nSize`

The size of the buffer pointed to by the *lpBuffer* parameter, including the null-terminating character, in characters.

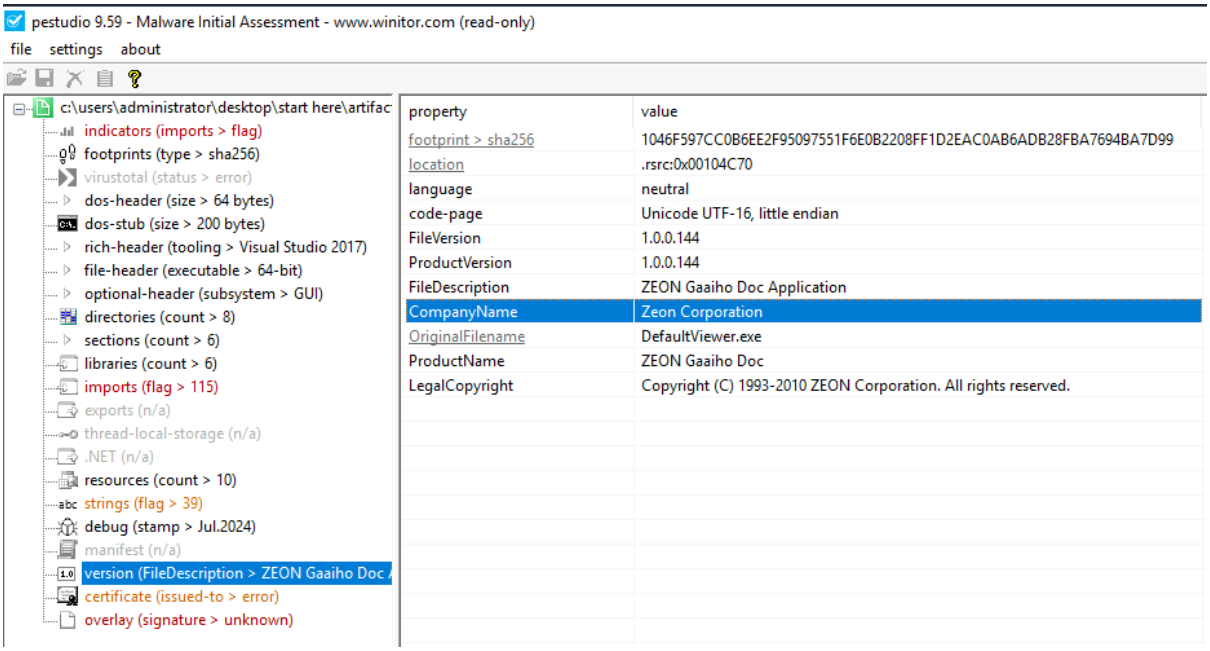
We can see this within the disassembled code in Cutter:



Answer: 3

Identifying specific details within a binary can provide insights into the target's identity or the attacker's intent. What is the name of the company embedded in the binary?

Navigating back to PEStudio, we can find the company name in the version header:



Answer: Zeon Corporation

Injecting shellcode through a Windows callback function involves placing the shellcode into a process and using a callback mechanism. What is the memory location of VirtualAlloc as identified in the Capa rules related to this technique?

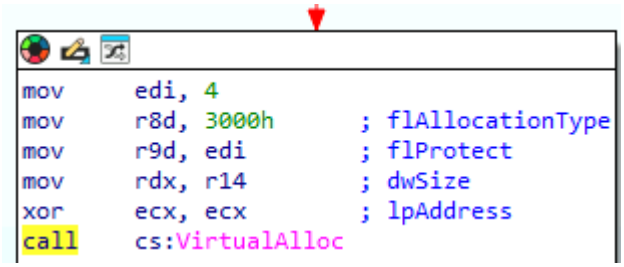
Capa is a tool developed by Mandiant that detects capabilities in executable files. To run Capa and display the virtual location of detected capabilities, execute the following command (make sure to use the -vv option):

- `capa -vv malware.exe`

The -vv flag (very verbose) reports exactly where Capa finds capabilities, which is useful to verify the output and analyse the sample further. Within the output, we can find the location of VirtualAlloc as identified by Capa:

```
execute shellcode via indirect call
namespace  load-code/shellcode
author     ronnie.salomonsen@mandiant.com
scope      function
mbc        Memory::Allocate Memory [C0007]
function @ 0x14000469C
  and:
    match: allocate RWX memory @ 0x140004A1E
      and:
        match: allocate memory @ 0x140004A1E
          or:
            api: kernel32.VirtualAlloc @ 0x140004A31
        number: 0x40 = PAGE_EXECUTE_READWRITE @ 0x140004A92
    or:
      characteristic: indirect call @ 0x140004730, 0x140004732, 0x140004754, 0x140004764, and 4 more...
```

This is extremely helpful as we can take the base address from capa, rebase our image and navigate to this address to analyse suspicious behaviour further. For example, I loaded the binary into IDA, navigated to Edit > Segments > Rebase program, supplied the base address given by Capa and navigated to the address identified in the previous image:



Answer: 0x140004A31