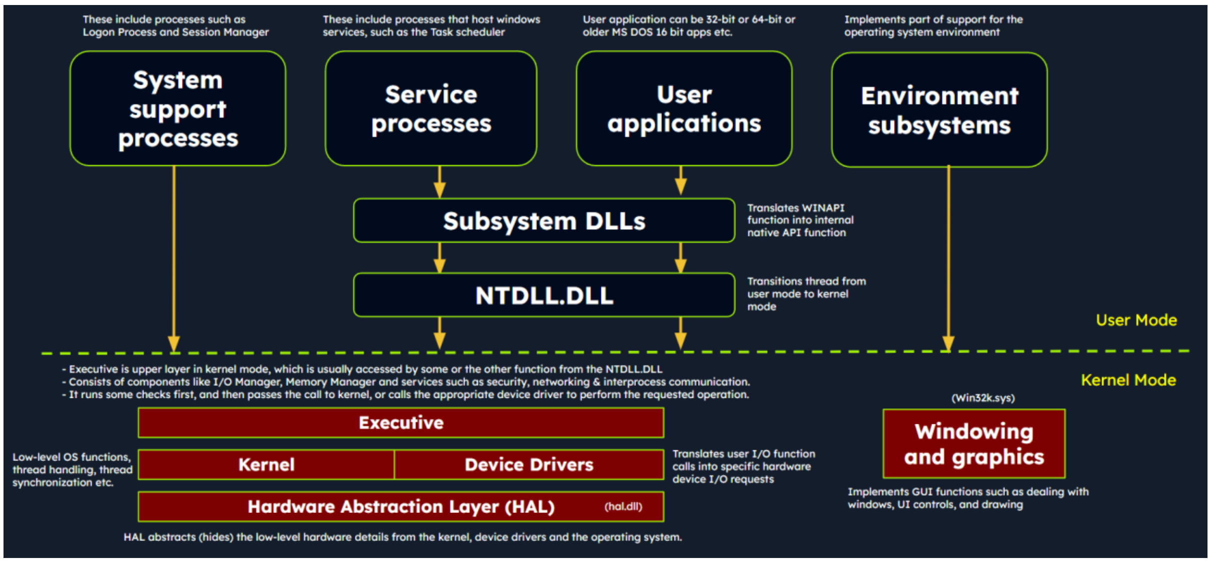
Windows Internals

To conduct effective malware analysis, a profound understanding of Windows internals is essential. Windows operating systems function in two main modes:

- User Mode: This mode is where most applications and user processes operate. Applications in user mode have limited access to system resources and must interact with the operating system through Application Programming Interfaces (APIs). These processes are isolated from each other and cannot directly access hardware or critical system functions. However, in this mode, malware can still manipulate files, registry settings, network connections, and other user-accessible resources, and it may attempt to escalate privileges to gain more control over the system.
- Kernel Mode: In contrast, kernel mode is a highly privileged mode where the Windows kernel runs. The kernel has unrestricted access to system resources, hardware, and critical functions. It provides core operating system services, manages system resources, and enforces security and stability. Device drivers, which facilitate communication with hardware devices, also run in kernel mode. If malware operates in kernel mode, it gains elevated control and can manipulate system behavior, conceal its presence, intercept system calls, and tamper with security mechanisms.

Windows Architecture At A High Level

The below image showcases a simplified version of Windows' architecture.


These include processes such as Logon Process and Session Manager
System support processes
These include processes that host windows services, such as the Task scheduler
Service processes
User application can be 32-bit or 64-bit or older MS DOS 16 bit apps etc.
User applications
Implements part of support for the operating system environment
Environment subsystems
Subsystem DLLs
Translates WINAPI function into internal native API function
NTDLL.DLL
Transitions thread from user mode to kernel mode
User Mode
Kernel Mode
- Executive is upper layer in kernel mode, which is usually accessed by some or the other function from the NTDLL.DLL
- Consists of components like I/O Manager, Memory Manager and services such as security, networking & interprocess communication.
- It runs some checks first, and then passes the call to kernel, or calls the appropriate device driver to perform the requested operation.
Executive
Low-level OS functions, thread handling, thread synchronization etc.
Kernel
Device Drivers
Translates user I/O function calls into specific hardware device I/O requests
Hardware Abstraction Layer (HAL)
(hal.dll)
HAL abstracts (hides) the low-level hardware details from the kernel, device drivers and the operating system.
(Win32k.sys)
Windowing and graphics
Implements GUI functions such as dealing with windows, UI controls, and drawing

The simplified Windows architecture comprises both user-mode and kernel-mode components, each with distinct responsibilities in the system's functioning.

User-mode Components

User-mode components are those parts of the operating system that don't have direct access to hardware or kernel data structures. They interact with system resources through APIs and system calls. Let's discuss some of them:

- System Support Processes: These are essential components that provide crucial functionalities and services such as logon processes (winlogon.exe), Session Manager (smss.exe), and Service Control Manager (services.exe). These aren't Windows services but they are necessary for the proper functioning of the system.

- Service Processes: These processes host Windows services like the Windows Update Service, Task Scheduler, and Print Spooler services. They usually run in the background, executing tasks according to their configuration and parameters.
- User Applications: These are the processes created by user programs, including both 32-bit and 64-bit applications. They interact with the operating system through **APIs** provided by Windows. These API calls get redirected to **NTDLL.DLL**, triggering a transition from user mode to kernel mode, where the system call gets executed. The result is then returned to the user-mode application, and a transition back to user mode occurs.
- Environment Subsystems: These components are responsible for providing execution environments for specific types of applications or processes. They include the **Win32 Subsystem**, **POSIX**, and **OS/2**.
- Subsystem DLLs: These dynamic-link libraries translate documented functions into appropriate internal native system calls, primarily implemented in NTDLL.DLL. Examples include kernelbase.dll, user32.dll, wininet.dll, and advapi32.dll.

Kernel-mode Components

Kernel-mode components are those parts of the operating system that have direct access to hardware and kernel data structures. These include:

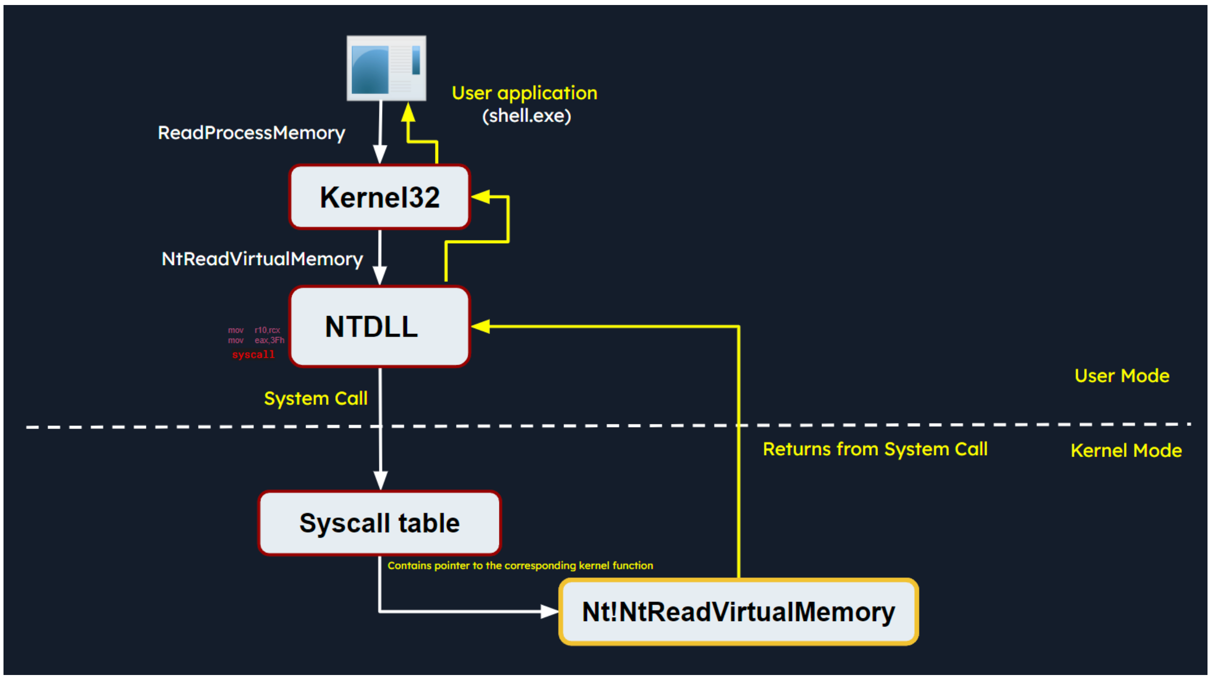
- Executive: This upper layer in kernel mode gets accessed through functions from NTDLL.DLL. It consists of components like the I/O Manager, Object Manager, Security Reference Monitor, Process Manager, and others, managing the core aspects of the operating system such as I/O operations, object management, security, and processes. It runs some checks first, and then passes the call to kernel, or calls the appropriate device driver to perform the requested operation.
- Kernel: This component manages system resources, providing low-level services like thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization.
- Device Drivers: These software components enable the OS to interact with hardware devices. They serve as intermediaries, allowing the system to manage and control hardware and software resources.
- Hardware Abstraction Layer (HAL): This component provides an abstraction layer between the hardware devices and the OS. It allows software developers to interact with hardware in a consistent and platform-independent manner.
- Windowing and Graphics System (Win32k.sys): This subsystem is responsible for managing the graphical user interface (GUI) and rendering visual elements on the screen.

Now, let's discuss in what happens behind the scenes when an user application calls a Windows API function.

Windows API Call Flow

Malware often utilize Windows API calls to interact with the system and carry out malicious operations. By understanding the internal details of API functions, their parameters, and expected behavior, analysts can identify suspicious or unauthorized API usage.

Let's consider an example of a Windows API call flow, where a user-mode application tries to access privileged operations and system resources using the [ReadProcessMemory function](). This function allows a process to read the memory of a different process.



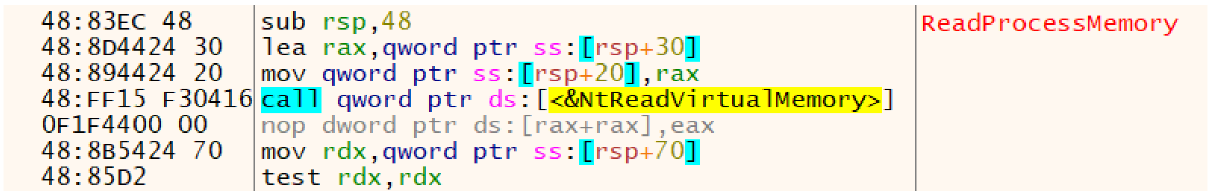
When this function is called, some required parameters are also passed to it, such as the handle to the target process, the source address to read from, a buffer in its own memory space to store the read data, and the number of bytes to read. Below is the syntax of ReadProcessMemory WINAPI function as per Microsoft documentation.

```
BOOL ReadProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPCVOID lpBaseAddress,
  [out] LPVOID  lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesRead
);
```

C++

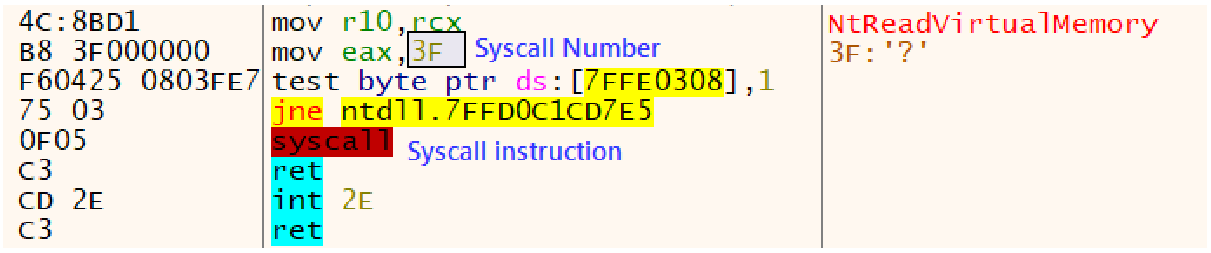
```cpp
BOOL ReadProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPCVOID lpBaseAddress,
  [out] LPVOID  lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesRead
);
```

ReadProcessMemory is a Windows API function that belongs to the kernel32.dll library. So, this call is invoked via the kernel32.dll module which serves as the user mode interface to the Windows API. Internally, the kernel32.dll module interacts with the NTDLL.DLL module, which provides a lower-level interface to the Windows kernel. Then, this function request is translated to the corresponding Native API call, which is NtReadVirtualMemory. The below screenshot from x64dbg demonstrates how this looks like in a debugger.



```
48:83EC 48       | sub rsp,48                                   | ReadProcessMemory
48:8D4424 30     | lea rax,qword ptr ss:[rsp+30]                |
48:894424 20     | mov qword ptr ss:[rsp+20],rax                |
48:FF15 F30416   | call qword ptr ds:[<&NtReadVirtualMemory>]   |
0F1F4400 00      | nop dword ptr ds:[rax+rax],eax               |
48:8B5424 70     | mov rdx,qword ptr ss:[rsp+70]                |
48:85D2          | test rdx,rdx                                 |
```

The NTDLL.DLL module utilizes system calls (syscalls).



```
4C:8BD1          | mov r10,rcx                  | NtReadVirtualMemory
B8 3F000000      | mov eax,3F                   | 3F:'?'
F60425 0803FE7   | test byte ptr ds:[7FFE0308],1 |
75 03            | jne ntdll.7FFD0C1CD7E5       |
0F05             | syscall                      |
C3               | ret                          |
CD 2E            | int 2E                       |
C3               | ret                          |
```

The syscall instruction triggers the system call using the parameters set in the previous instructions. It transfers control from user mode to kernel mode, where the kernel performs the requested operation after validating the parameters and checking the access rights of the calling process.

If the request is authorized, the thread is transitioned from user mode into the kernel mode. The kernel maintains a table known as the System Service Descriptor Table (SSDT) or the syscall table (System Call Table), which is a data structure that contains pointers to the various system service routines. These routines are responsible for handling system calls made by user-mode applications. Each entry in the syscall table corresponds to a specific system call number, and the associated pointer points to the corresponding kernel function that implements the requested operation.

The syscall responsible for ReadProcessMemory is executed in the kernel, where the Windows memory management and process isolation mechanisms are leveraged. The kernel performs necessary validations, access checks, and memory operations to read the memory from the target process. The kernel retrieves the physical memory pages corresponding to the requested virtual addresses and copies the data into the provided buffer.

Once the kernel has finished reading the memory, it transitions the thread back to user mode and control is handed back to the original user mode application. The application can then access the data that was read from the target process's memory and continue its execution.

Portable Executable

Windows operating systems employ the Portable Executable (PE) format to encapsulate executable programs, DLLs (Dynamic Link Libraries), and other integral system components. In the realm of malware analysis, an intricate understanding of the PE file format is indispensable. It allows us to gain significant insights into the executable's structure, operations, and potential malign activities embedded within the file.

PE files accommodate a wide variety of data types including executables (.exe), dynamic link libraries (.dll), kernel modules (.srv), control panel applications (.cpl), and many more. The PE file format is fundamentally a data structure containing the vital information required for the Windows OS loader to manage the executable code, effectively loading it into memory.

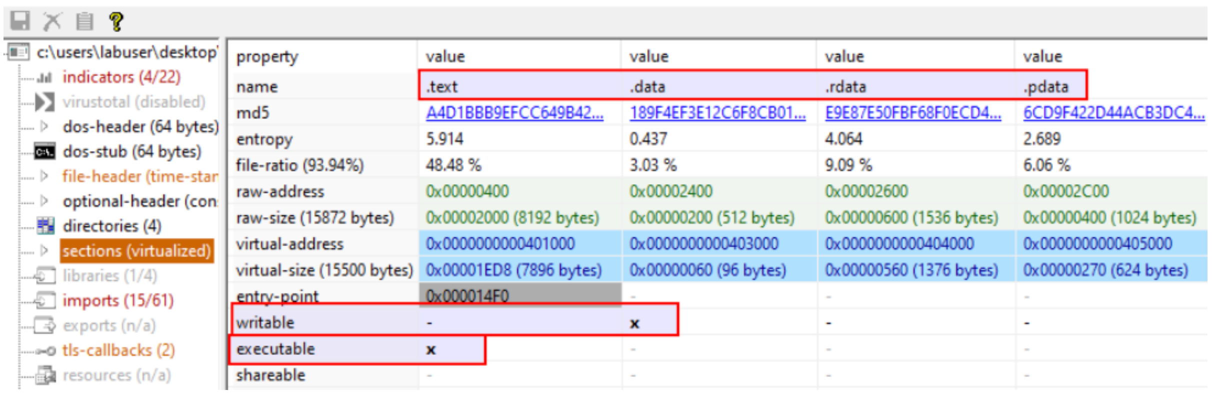
PE Sections

The PE Structure also houses a Section Table, an element comprising several sections dedicated to distinct purposes. The sections are essentially the repositories where the actual content of the file, including the data, resources utilized by the program, and the executable code, is stored. The .text section is often under scrutiny for potential artifacts related to injection attacks.

Common PE sections include:

- Text Section (.text): The hub where the executable code of the program resides.
- Data Section (.data): A storage for initialized global and static data variables.
- Read-only initialized data (.rdata): Houses read-only data such as constant values, string literals, and initialized global and static variables.
- Exception information (.pdata): A collection of function table entries utilized for exception handling.
- BSS Section (.bss): Holds uninitialized global and static data variables.
- Resource Section (.rsrc): Safeguards resources such as images, icons, strings, and version information.
- Import Section (.idata): Details about functions imported from other DLLs.
- Export Section (.edata): Information about functions exported by the executable.
- Relocation Section (.reloc): Details for relocating the executable's code and data when loaded at a different memory address.

We can visualize the sections of a portable executable using a tool like pestudio as demonstrated below.



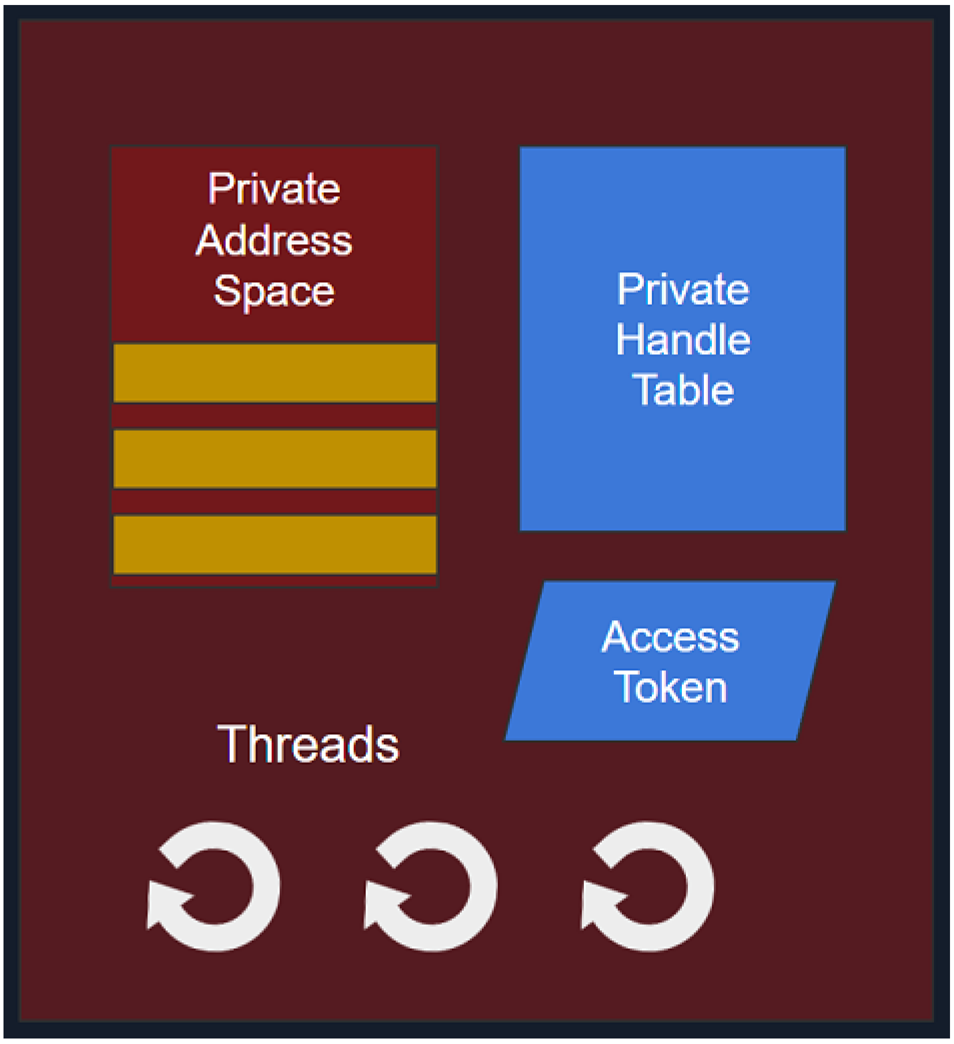
c:\users\labuser\desktop
- indicators (4/22)
- virustotal (disabled)
- dos-header (64 bytes)
- dos-stub (64 bytes)
- file-header (time-star
- optional-header (con
- directories (4)
- sections (virtualized)
- libraries (1/4)
- imports (15/61)
- exports (n/a)
- tls-callbacks (2)
- resources (n/a)

property	value	value	value	value
name	.text	.data	.rdata	.pdata
md5	A4D1BBB9EFCC649B42...	189F4EF3E12C6F8CB01...	E9E87E50FBF68F0ECD4...	6CD9F422D44ACB3DC4...
entropy	5.914	0.437	4.064	2.689
file-ratio (93.94%)	48.48 %	3.03 %	9.09 %	6.06 %
raw-address	0x00000400	0x00002400	0x00002600	0x00002C00
raw-size (15872 bytes)	0x00002000 (8192 bytes)	0x00000200 (512 bytes)	0x00000600 (1536 bytes)	0x00000400 (1024 bytes)
virtual-address	0x0000000000401000	0x0000000000403000	0x0000000000404000	0x0000000000405000
virtual-size (15500 bytes)	0x00001ED8 (7896 bytes)	0x00000060 (96 bytes)	0x00000560 (1376 bytes)	0x00000270 (624 bytes)
entry-point	0x000014F0	-	-	-
writable	-	x	-	-
executable	x	-	-	-
shareable	-	-	-	-

Delving into the Portable Executable (PE) file format is pivotal for malware analysis, offering insights into the file's structure, code analysis, import and export functions, resource analysis, anti-analysis techniques, and extraction of indicators of compromise. Our comprehension of this foundation paves the way for efficacious malware analysis.

Processes

In the simplest terms, a process is an instance of an executing program. It represents a slice of a program's execution in memory and consists of various resources, including memory, file handles, threads, and security contexts.



Each process is characterized by:

- A unique PID (Process Identifier): A unique Process Identifier (PID) is assigned to each process within the operating system. This numeric identifier facilitates the tracking and management of the process by the operating system.
- Virtual Address Space (VAS): In the Windows OS, every process is allocated its own virtual address space, offering a virtualized view of the memory for the process. The VAS is sectioned into segments, including code, data, and stack segments, allowing the process isolated memory access.
- Executable Code (Image File on Disk): The executable code, or the image file, signifies the binary executable file stored on the disk. It houses the instructions and resources necessary for the process to operate.
- Table of Handles to System Objects: Processes maintain a table of handles, a reference catalogue for various system objects. System objects can span files, devices, registry keys, synchronization objects, and other resources.
- Security Context (Access Token): Each process has a security context associated with it, embodied by an Access Token. This Access Token encapsulates information about the process's security privileges, including the user account under which the process operates and the access rights granted to the process.
- One or More Threads Running in its Context: Processes consist of one or more threads, where a thread embodies a unit of execution within the process. Threads enable concurrent execution within the process and facilitate multitasking.

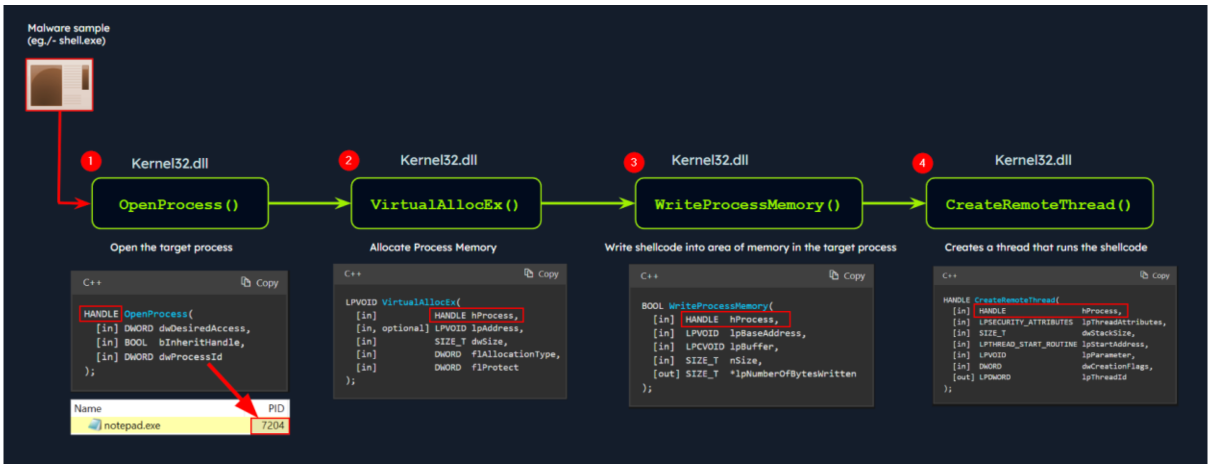
Dynamic-link library (DLL)

A Dynamic-link library (DLL) is a type of PE which represents "Microsoft's implementation of the shared library concept in the Microsoft Windows OS". DLLs expose an array of functions which can be exploited by malware, which we'll scrutinize later. First, let's unravel the import and export functions in a DLL.

Import Functions

- Import functions are functionalities that a binary dynamically links to from external libraries or modules during runtime. These functions enable the binary to leverage the functionalities offered by these libraries.
- During malware analysis, examining import functions may shed light on the external libraries or modules that the malware is dependent on. This information aids in identifying the APIs that the malware might interact with, and also the resources such as the file system, processes, registry etc.
- By identifying specific functions imported, it becomes possible to ascertain the actions the malware can perform, such as file operations, network communication, registry manipulation, and more.
- Import function names or hashes can serve as IOCs (Indicators of Compromise) that assist in identifying malware variants or related samples.

Below is an example of identifying process injection using DLL imports and function names:



```cpp
HANDLE OpenProcess(
   [in] DWORD dwDesiredAccess,
   [in] BOOL  bInheritHandle,
   [in] DWORD dwProcessId
);
```

```cpp
LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

```cpp
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

```cpp
HANDLE CreateRemoteThread(
  [in]  HANDLE                 hProcess,
  [in]  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  [in]  SIZE_T                 dwStackSize,
  [in]  LPTHREAD_START_ROUTINE lpStartAddress,
  [in]  LPVOID                 lpParameter,
  [in]  DWORD                  dwCreationFlags,
  [out] LPDWORD                lpThreadId
);
```

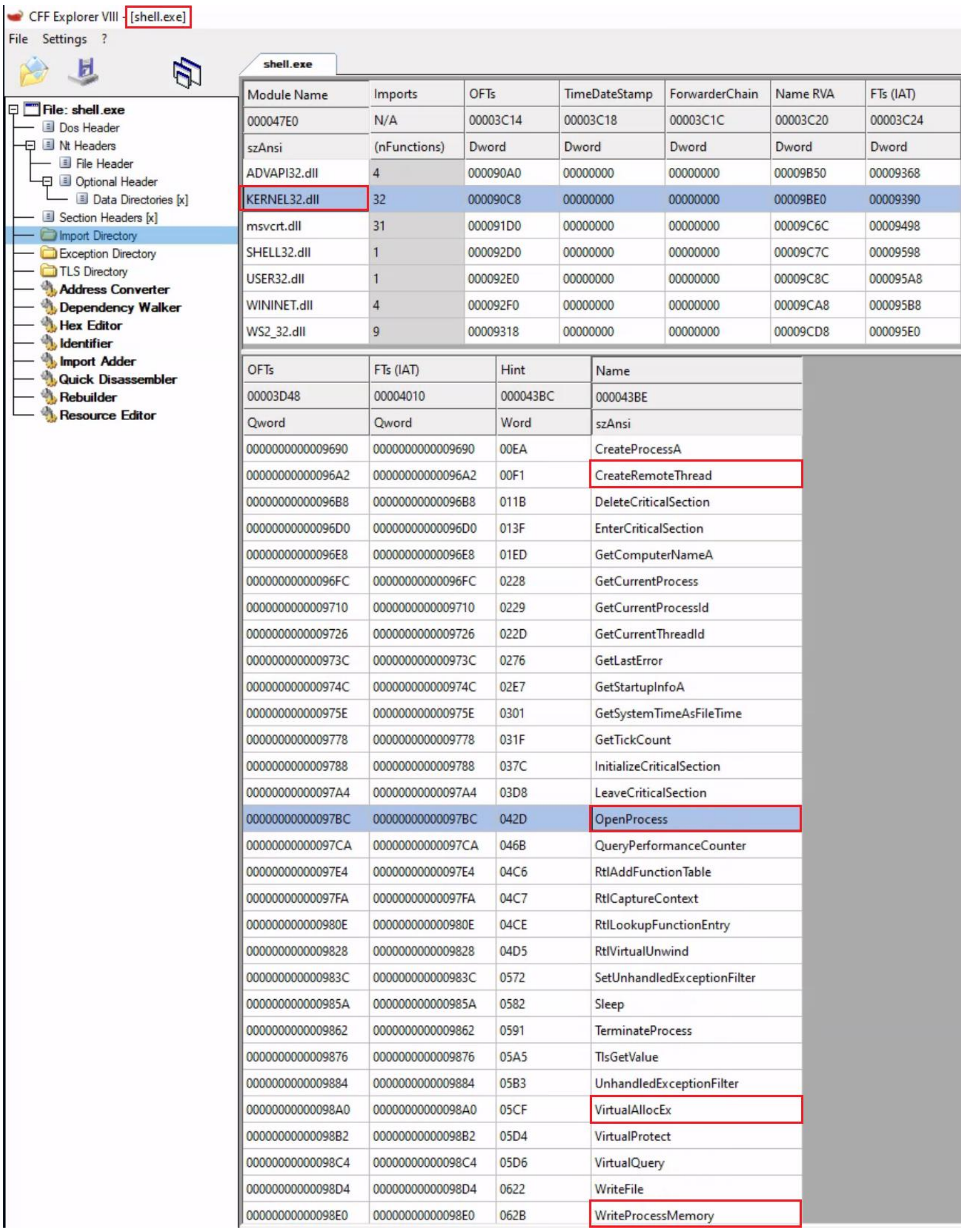
In this diagram, the malware process (shell.exe) performs process injection to inject code into a target process (notepad.exe) using the following functions imported from the DLL kernel32.exe:

- OpenProcess: Opens a handle to the target process (notepad.exe), providing the necessary access rights to manipulate its memory.
- VirtualAllocEx: Allocates a block of memory within the address space of the target process to store the injected code.
- WriteProcessMemory: Writes the desired code into the allocated memory block of the target process.
- CreateRemoteThread: Creates a new thread within the target process, specifying the entry point of the injected code as the starting point.

As a result, the injected code is executed within the context of the target process by the newly created remote thread. This technique allows the malware to run arbitrary code within the target process.

The functions above are WINAPI (Windows API) functions. Don't worry about WINAPI functions as of now. We'll discuss these in detail later.

We can examine the DLL imports of shell.exe (residing in the C:\Samples\MalwareAnalysis directory) using CFF Explorer (available at C:\Tools\Explorer Suite) as follows.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
000047E0	N/A	00003C14	00003C18	00003C1C	00003C20	00003C24
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ADVAPI32.dll	4	000090A0	00000000	00000000	00009B50	00009368
KERNEL32.dll	32	000090C8	00000000	00000000	00009BE0	00009390
msvcrt.dll	31	000091D0	00000000	00000000	00009C6C	00009498
SHELL32.dll	1	000092D0	00000000	00000000	00009C7C	00009598
USER32.dll	1	000092E0	00000000	00000000	00009C8C	000095A8
WININET.dll	4	000092F0	00000000	00000000	00009CA8	000095B8
WS2_32.dll	9	00009318	00000000	00000000	00009CD8	000095E0

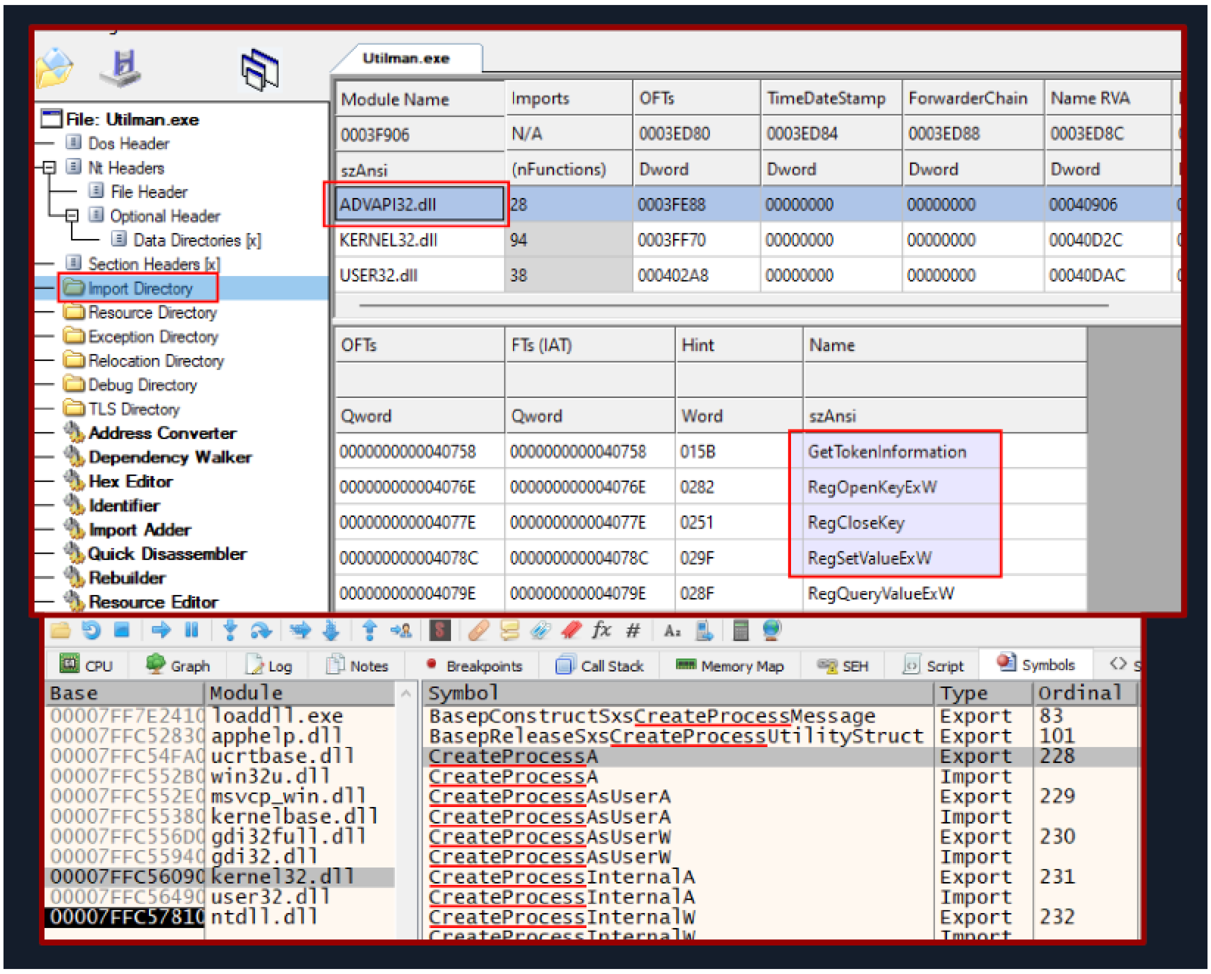
OFTs	FTs (IAT)	Hint	Name
00003D48	00004010	000043BC	000043BE
Qword	Qword	Word	szAnsi
0000000000009690	0000000000009690	00EA	CreateProcessA
00000000000096A2	00000000000096A2	00F1	CreateRemoteThread
00000000000096B8	00000000000096B8	011B	DeleteCriticalSection
00000000000096D0	00000000000096D0	013F	EnterCriticalSection
00000000000096E8	00000000000096E8	01ED	GetComputerNameA
00000000000096FC	00000000000096FC	0228	GetCurrentProcess
0000000000009710	0000000000009710	0229	GetCurrentProcessId
0000000000009726	0000000000009726	022D	GetCurrentThreadId
000000000000973C	000000000000973C	0276	GetLastError
000000000000974C	000000000000974C	02E7	GetStartupInfoA
000000000000975E	000000000000975E	0301	GetSystemTimeAsFileTime
0000000000009778	0000000000009778	031F	GetTickCount
0000000000009788	0000000000009788	037C	InitializeCriticalSection
00000000000097A4	00000000000097A4	03D8	LeaveCriticalSection
00000000000097BC	00000000000097BC	042D	OpenProcess
00000000000097CA	00000000000097CA	046B	QueryPerformanceCounter
00000000000097E4	00000000000097E4	04C6	RtlAddFunctionTable
00000000000097FA	00000000000097FA	04C7	RtlCaptureContext
000000000000980E	000000000000980E	04CE	RtlLookupFunctionEntry
0000000000009828	0000000000009828	04D5	RtlVirtualUnwind
000000000000983C	000000000000983C	0572	SetUnhandledExceptionFilter
000000000000985A	000000000000985A	0582	Sleep
0000000000009862	0000000000009862	0591	TerminateProcess
0000000000009876	0000000000009876	05A5	TlsGetValue
0000000000009884	0000000000009884	05B3	UnhandledExceptionFilter
00000000000098A0	00000000000098A0	05CF	VirtualAllocEx
00000000000098B2	00000000000098B2	05D4	VirtualProtect
00000000000098C4	00000000000098C4	05D6	VirtualQuery
00000000000098D4	00000000000098D4	0622	WriteFile
00000000000098E0	00000000000098E0	062B	WriteProcessMemory

Export Functions

- Export functions are the functions that a binary exposes for use by other modules or applications.
- These functions provide an interface for other software to interact with the binary.

In the below screenshot, we can see an example of DLL imports (using CFF Explorer) and exports (using x64dbg - Symbols tab):

- Imports: This shows the DLLs and their functions imported by an executable Utilman.exe.
- Exports: This shows the functions exported by a DLL Kernel32.dll.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA
0003F906	N/A	0003ED80	0003ED84	0003ED88	0003ED8C
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword
ADVAPI32.dll	28	0003FE88	00000000	00000000	00040906
KERNEL32.dll	94	0003FF70	00000000	00000000	00040D2C
USER32.dll	38	000402A8	00000000	00000000	00040DAC

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
0000000000040758	0000000000040758	015B	GetTokenInformation
000000000004076E	000000000004076E	0282	RegOpenKeyExW
000000000004077E	000000000004077E	0251	RegCloseKey
000000000004078C	000000000004078C	029F	RegSetValueExW
000000000004079E	000000000004079E	028F	RegQueryValueExW

Base	Module
00007FF7E2410	loaddll.exe
00007FFC52830	apphelp.dll
00007FFC54FA0	ucrtbase.dll
00007FFC552B0	win32u.dll
00007FFC552E0	msvcp_win.dll
00007FFC55380	kernelbase.dll
00007FFC556D0	gdi32full.dll
00007FFC55940	gdi32.dll
00007FFC56090	kernel32.dll
00007FFC56490	user32.dll
00007FFC57810	ntdll.dll

Symbol	Type	Ordinal
BasepConstructSxsCreateProcessMessage	Export	83
BasepReleaseSxsCreateProcessUtilityStruct	Export	101
CreateProcessA	Export	228
CreateProcessA	Import	
CreateProcessAsUserA	Export	229
CreateProcessAsUserA	Import	
CreateProcessAsUserW	Export	230
CreateProcessAsUserW	Import	
CreateProcessInternalA	Export	231
CreateProcessInternalA	Import	
CreateProcessInternalW	Export	232
CreateProcessInternalW	Import	

In the context of malware analysis, understanding import and export functions assists in discerning the behavior, capabilities, and interactions of the binary with external entities. It yields valuable information for threat detection, classification, and gauging the impact of the malware on the system.