Windows API for Red Team #101

Author

Joas Antonio dos Santos

https://www.linkedin.com/in/joas-antonio-dos-santos/

TABLE OF CONTENTS

WHAT IS WINDOWS API

INTRODUCTION

Windows API, also known as Windows API or WinAPI, is a set of application programming interfaces (APIs) provided by Microsoft to allow interaction between programs and the Windows operating system. These APIs provide functions that allow developers to manipulate operating system components such as windows, files, processes, graphics, networks, among others.

WinAPI is divided into several sections, each dealing with different aspects of the operating system:

1. **User Interface**: APIs for manipulating windows, menus, dialog boxes, and other user interface elements.
2. **Graphics Device Interface (GDI)**: APIs that allow the manipulation of graphics, such as drawing shapes, texts and images.
3. **System Services**: Functions that provide access to deeper operating system features such as file, event, and process management.
4. **Multimedia**: APIs dedicated to handling audio and video.
5. **Networking**: Functions for managing network communications.

These APIs are essential for developing native Windows applications and are used in a wide range of software, from simple applications to full operating systems that run on the Windows platform. They are available for several programming languages, such as C, C++, and can also be accessed through other languages via "wrappers" or specific libraries.

Functionalities

Windows APIs involve a series of features that allow developers to create applications that interact directly with the Windows operating system. These APIs cover a wide range of functionality, from creating and managing graphical user interface elements to accessing system resources such as files, networks, and devices. Here are some of the main components and functionalities that Windows APIs involve:

1. **User Interface Management**:
    - Creation and manipulation of windows and dialogs.
    - Message and event processing.
    - Interface controls such as buttons, text boxes, and lists.
2. **Graphics Device Interface (GDI)**:
    - Drawing graphics on the screen.
    - Manipulation of fonts and texts.
    - Rendering images and shapes.
3. **DirectX**:
    - APIs for developing games and applications that require high-performance graphics.
    - Support for real-time audio, video, and event processing.
4. **System Services**:
    - Access to files and directories.
    - Memory and process management.
    - Security and access control services.
5. **Communication and Networking**:
    - APIs for network protocols such as TCP/IP.
    - Functions for creating and managing network connections.
    - Inter-process communication services.
6. **Multimedia**:
    - Manipulation of audio and video files.
    - Media capture and playback.
7. **Windows Registry**:
    - Access and manipulation of the Windows registry, which stores system and application settings.
8. **COM and DCOM**:

- Component Object Model (COM) allows interaction between software components that may be in different processes or even computers.
- Distributed COM (DCOM) extends these capabilities to networks.

9. **Windows Runtime (WinRT)**:
    - A newer API that allows application development for the Windows platform with support for multiple devices, including computers, tablets and phones.

Windows APIs are designed to provide a robust platform for software development, enabling applications to leverage the resources and capabilities of hardware and the operating system efficiently and securely.

How to work with Windows API?

To access Windows APIs, you can follow some basic steps that involve choosing a suitable programming language, configuring the development environment, and using specific libraries that allow interaction with the operating system. Here are the detailed steps:

1. **Choose a Programming Language**:
    - Windows APIs are traditionally accessed using C or C++, but can also be used through other languages such as C#, Visual Basic, and Python, using appropriate bindings or wrappers.
2. **Configure your Development Environment**:
    - **For C/C++**: Install an integrated development environment (IDE) such as Microsoft Visual Studio, which offers full support for Windows development with C/C++. Visual Studio already includes the headers and libraries needed to access the Windows APIs.
    - **For C# or Visual Basic**: Visual Studio is also recommended as it offers easy access to the .NET Framework and Windows Runtime, which are more modern interfaces to Windows APIs.
    - **For Python**: Install a library like pywin32, which gives you access to many of the Windows APIs.
3. **Learn the Specific API**:
    - Consult official Microsoft documentation to understand the specific API functions you want to use. Microsoft's documentation is quite comprehensive and includes code samples.
    - Microsoft Docs (https://docs.microsoft.com/) is the recommended resource for finding detailed information about each API.

4. **Practice with Examples**:
    - Start with simple examples, like creating a window or manipulating files, and gradually move on to more complex tasks.
    - The developer community and online forums like Stack Overflow can be helpful resources for learning and solving specific problems.
5. **Use Libraries and Frameworks**:
    - For common tasks, there are many libraries and frameworks that simplify the use of Windows APIs. For example, the .NET Framework for C# and VB or Qt for C++ offer high-level abstractions that facilitate the development of robust applications.
6. **Compilation and Linking**:
    - Make sure your project is correctly configured to compile and link the libraries required to access the Windows APIs. In Visual Studio this is usually managed automatically, but in other environments it may need to be configured manually.

Minimum Hardware Requirements:

- Windows 10 latest version (here)
- 8 GB Memory Ram and 120 GB Disk Minimum
- Visual Studio 2022 Community Edition (here)

Starting with a good development environment and accessing appropriate educational resources are key to effectively leveraging Windows APIs in your software projects.

Windows API in the attack context

Windows APIs are fundamental in the context of cybersecurity, particularly in Red Team activities and adversary simulation. The ability of these APIs to interact deeply with the Windows operating system allows penetration testers and adversary simulations to perform a variety of advanced tactics, which can include reading the memory of other processes, executing code with elevated privileges, detailed system enumeration , and neutralization of antivirus (AV) solutions. Below, I detail why these capabilities are crucial to cybersecurity and the importance of caring about them:

1. **Code Execution with Elevated Privileges**:
    - Privilege control is a critical part of system security. Windows APIs allow programs to execute code with elevated privileges, which can be exploited to gain complete control over the system. This is an essential attack vector that Red Teams test to ensure systems are adequately protected against privilege escalation.
2. **Reading Memory of Other Processes**:
    - The ability to read the memory of other processes is a powerful tool for adversaries as it can reveal sensitive information, including passwords and encryption keys. Red Teams use this technique to simulate attacks that attempt to extract sensitive data that could be misused.
3. **System Enumeration**:
    - Windows APIs can be used to enumerate system resources and configurations, allowing adversaries to map the target environment. This is essential for planning more targeted and effective attacks, helping Red Teams identify potential vulnerabilities and misconfigurations.
4. **Antivirus Neutralization**:
    - Killing or disabling AV solutions is a common technique among adversaries to avoid detection and analysis. Using Windows APIs, Red Teams can test the resilience of implemented security solutions against tactics that attempt to disable or bypass these protections.
5. **Personalization and Discretion**:
    - By using custom code that directly calls the Windows API, Red Teams can create tools that are highly adapted to the target environment and less likely to be detected by conventional security solutions. This simulates advanced adversaries that use custom code and sophisticated evasion techniques.

Studying and understanding Windows APIs is essential for security teams as they offer the ability to not only protect against, but also simulate, advanced tactics used in real cyberattacks.

WHAT IS IAT TABLE?

INTRODUCTION

The Import Address Table (IAT) is a crucial component in executable files in the Portable Executable (PE) format, which is used in Windows operating systems. IAT is used to manage function calls that are imported from other modules or dynamic libraries (DLLs). When an executable or DLL needs a function that is in another DLL, that function is called through an entry in the IAT, which contains the addresses of all the imported functions that the executable or DLL needs.

What is its Importance?

The IAT table is essential for several reasons in software development and security:

1. **Dynamic Address Resolution**: IAT allows the addresses of imported functions to be resolved at runtime by the Windows loader. This means that executable code can use multiple versions of a DLL without needing to be recompiled, as long as the function interfaces remain consistent.

2. **Optimization and Maintenance**: Maintaining function calls through IAT makes application maintenance and updating easier by allowing new versions of DLLs to be simply replaced without changing the core code base.

3. **Security**: IAT is often a target in software and malware exploitation techniques. Modifying the IAT could allow an attacker to redirect function calls to malicious code. Therefore, understanding and protecting IAT is fundamental to software security.

Using PEViewer to View Imports

PEView is a tool that allows users to view and analyze the internal components of executable files in PE format, including the Import Address Table. Here is how to use PEView to examine an executable's IAT:

1. **Download and Open PEView**: First, you need to download and open PEView. Upload the PE file (executable or DLL) you want to analyze.

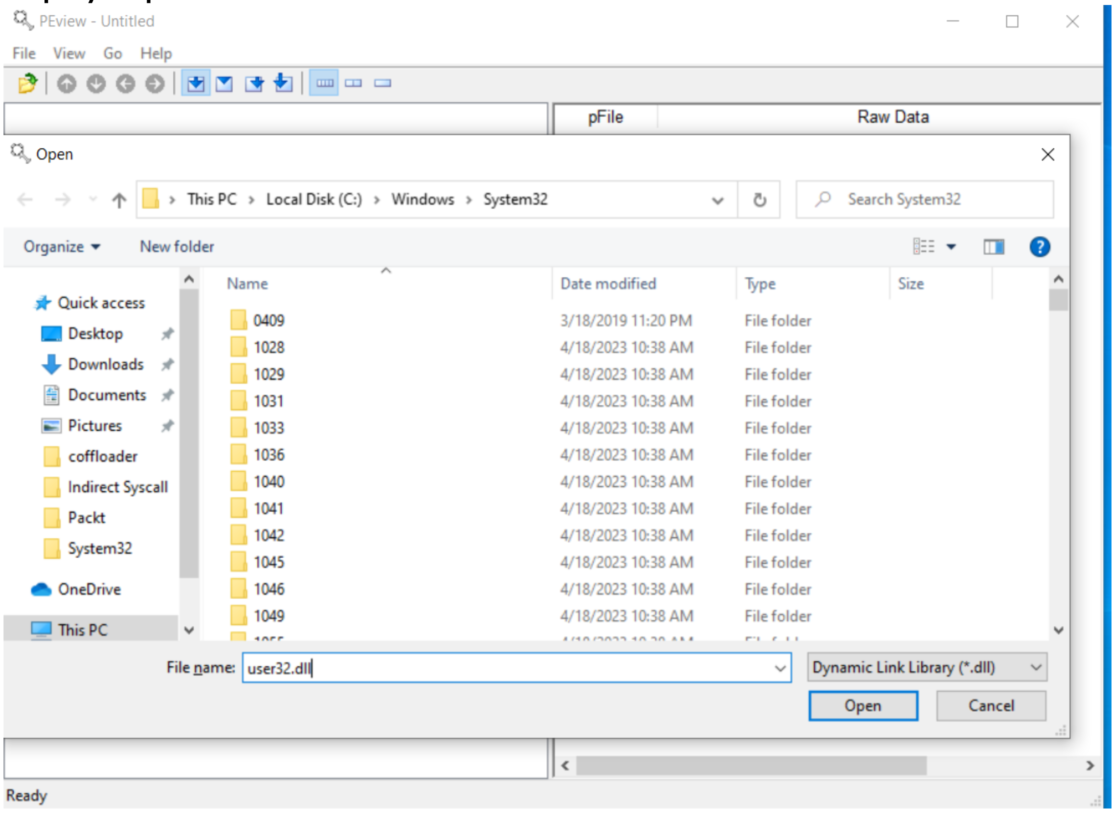
2. **Navigate to the Imports Section**: Within PEView, navigate to the section that lists imports. This section will show all the DLLs that the file depends on, as well as the specific functions that are imported from each one.

3. **Examine the IAT**: When you select a specific DLL, you can see the list of imported functions that are associated with that DLL. These entries correspond to the addresses in the IAT, where the system loader will resolve the actual addresses of the functions at run time.

4. **Security Analysis**: Use this view to understand which functions your application is importing and from which libraries. This can be crucial for identifying potential vulnerabilities, such as the use of outdated or insecure functions.

Using PEView is a common practice among developers and security analysts to better understand Windows application dependencies and to verify the integrity of software imports, making it a valuable tool for both development and security auditing.

Step by step:



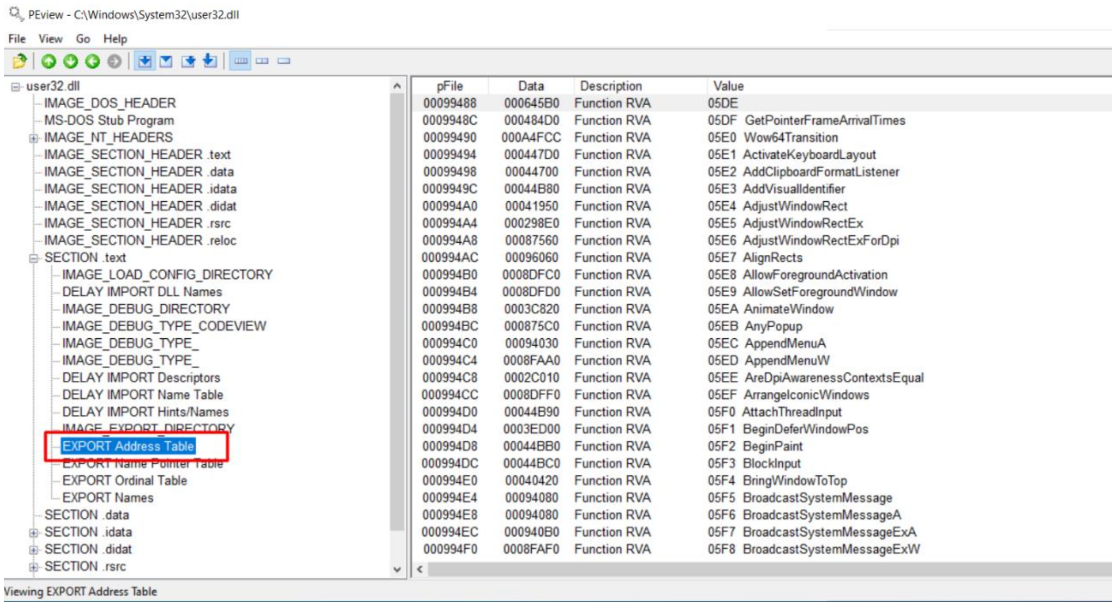
Name	Date modified	Type	Size
0409	3/18/2019 11:20 PM	File folder	
1028	4/18/2023 10:38 AM	File folder	
1029	4/18/2023 10:38 AM	File folder	
1031	4/18/2023 10:38 AM	File folder	
1033	4/18/2023 10:38 AM	File folder	
1036	4/18/2023 10:38 AM	File folder	
1040	4/18/2023 10:38 AM	File folder	
1041	4/18/2023 10:38 AM	File folder	
1042	4/18/2023 10:38 AM	File folder	
1045	4/18/2023 10:38 AM	File folder	
1046	4/18/2023 10:38 AM	File folder	
1049	4/18/2023 10:38 AM	File folder	

File name: user32.dll

Dynamic Link Library (*.dll)

Figure 1 – Selecting the DLL to analyze imports

After opening PEView, select a DLL, in this case I will import user32.dll which contains numerous API calls.



```
user32.dll
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .idata
  IMAGE_SECTION_HEADER .didat
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .reloc
  SECTION .text
    IMAGE_LOAD_CONFIG_DIRECTORY
    DELAY IMPORT DLL Names
    IMAGE_DEBUG_DIRECTORY
    IMAGE_DEBUG_TYPE_CODEVIEW
    IMAGE_DEBUG_TYPE_
    IMAGE_DEBUG_TYPE_
    DELAY IMPORT Descriptors
    DELAY IMPORT Name Table
    DELAY IMPORT Hints/Names
    IMAGE_EXPORT_DIRECTORY
    EXPORT Address Table
    EXPORT Name Pointer Table
    EXPORT Ordinal Table
    EXPORT Names
  SECTION .data
  SECTION .idata
  SECTION .didat
  SECTION .rsrc
```

pFile	Data	Description	Value
00099488	000645B0	Function RVA	05DE
0009948C	000484D0	Function RVA	05DF GetPointerFrameArrivalTimes
00099490	000A4FCC	Function RVA	05E0 Wow64Transition
00099494	000447D0	Function RVA	05E1 ActivateKeyboardLayout
00099498	00044700	Function RVA	05E2 AddClipboardFormatListener
0009949C	00044B80	Function RVA	05E3 AddVisualIdentifier
000994A0	00041950	Function RVA	05E4 AdjustWindowRect
000994A4	000298E0	Function RVA	05E5 AdjustWindowRectEx
000994A8	00087560	Function RVA	05E6 AdjustWindowRectExForDpi
000994AC	00096060	Function RVA	05E7 AlignRects
000994B0	0008DFC0	Function RVA	05E8 AllowForegroundActivation
000994B4	0008DFD0	Function RVA	05E9 AllowSetForegroundWindow
000994B8	0003C820	Function RVA	05EA AnimateWindow
000994BC	000875C0	Function RVA	05EB AnyPopup
000994C0	00094030	Function RVA	05EC AppendMenuA
000994C4	0008FAA0	Function RVA	05ED AppendMenuW
000994C8	0002C010	Function RVA	05EE AreDpiAwarenessContextsEqual
000994CC	0008DFF0	Function RVA	05EF ArrangeIconicWindows
000994D0	00044B90	Function RVA	05F0 AttachThreadInput
000994D4	0003ED00	Function RVA	05F1 BeginDeferWindowPos
000994D8	00044BB0	Function RVA	05F2 BeginPaint
000994DC	00044BC0	Function RVA	05F3 BlockInput
000994E0	00040420	Function RVA	05F4 BringWindowToTop
000994E4	00094080	Function RVA	05F5 BroadcastSystemMessage
000994E8	00094080	Function RVA	05F6 BroadcastSystemMessageA
000994EC	000940B0	Function RVA	05F7 BroadcastSystemMessageExA
000994F0	0008FAF0	Function RVA	05F8 BroadcastSystemMessageExW

Figure 2 – Analyzing the address tables

When we access the address tables, we will find numerous API functions that the dll has in its table, so we can obtain crucial information such as the memory addresses of these functions.

We can create a Python script that extracts these addresses from a PE file.

To extract the Import Address Table (IAT) from PE (Portable Executable) files in Python, you can use the pefile library. This library is widely used for analysis and manipulation of PE files in security and reverse engineering environments.

Here is a basic script that demonstrates how to extract and print the IAT table from a PE file using pefile:

```
import profile

import sys



def extract_iat(pe_file):

try:

# Load the PE file

pe = pefile.PE(pe_file)
```

```
# Check if the file has an imports table

if hasattr(pe, 'DIRECTORY_ENTRY_IMPORT'):

print(f"IAT for {pe_file}:")



# Iterate over each entry in the imports table

for entry in pe.DIRECTORY_ENTRY_IMPORT:

print(f"Imports from {entry.dll.decode()}:")

for imp in entry.imports:

address = hex(imp.address)

name = imp.name.decode() if imp.name else 'Ordinal Import'

print(f" {address} {name}")

else:

print("No imports found.")



except Exception as e:

print(f"Error processing the file: {e}")



if __name__ == '__main__':

if len(sys.argv) > 1:

extract_iat(sys.argv[1])

else:

print("Please provide the path to the PE file as an argument.")
```

Install the profile library: Before running the script, you need to install the pefile library. You can do this using pip:

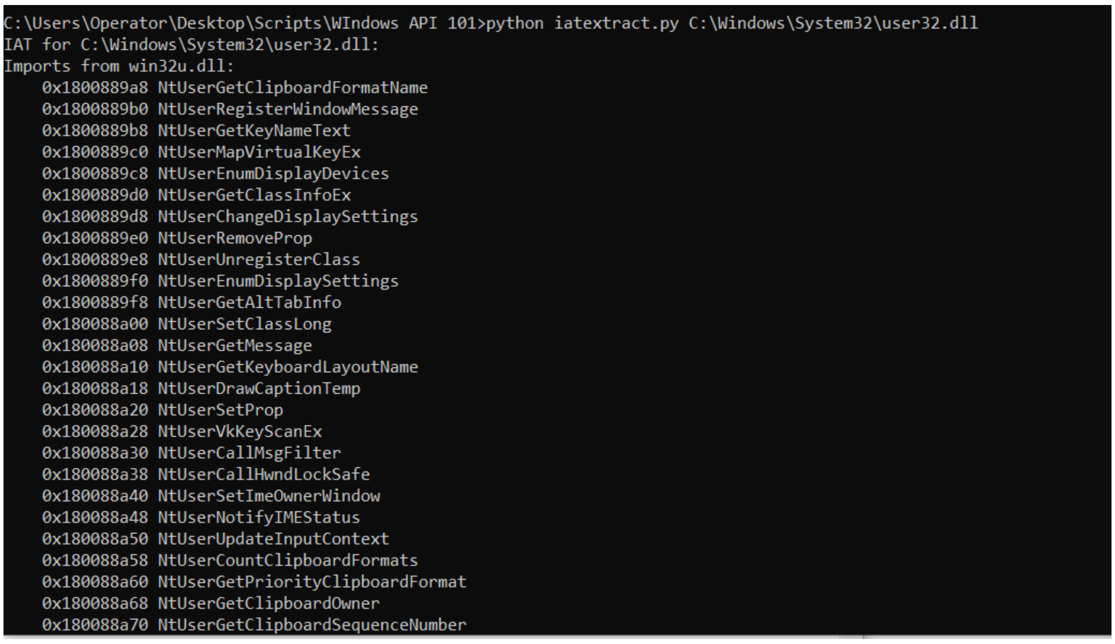
```
pip install profile

python extract_iat.py path_to_file.exe
```

Script Operation

- The script loads the PE file using pefile.PE().
- Checks if the file has an import table (DIRECTORY_ENTRY_IMPORT).
- For each DLL in the imports table, it lists the functions being imported, including their addresses.

This script is a basic introduction and can be expanded or modified to include more functionality, such as filtering for specific types of imports or manipulating the data in other ways as needed.



```
C:\Users\Operator\Desktop\Scripts\WIndows API 101>python iatextract.py C:\Windows\System32\user32.dll
IAT for C:\Windows\System32\user32.dll:
Imports from win32u.dll:
    0x1800889a8 NtUserGetClipboardFormatName
    0x1800889b0 NtUserRegisterWindowMessage
    0x1800889b8 NtUserGetKeyNameText
    0x1800889c0 NtUserMapVirtualKeyEx
    0x1800889c8 NtUserEnumDisplayDevices
    0x1800889d0 NtUserGetClassInfoEx
    0x1800889d8 NtUserChangeDisplaySettings
    0x1800889e0 NtUserRemoveProp
    0x1800889e8 NtUserUnregisterClass
    0x1800889f0 NtUserEnumDisplaySettings
    0x1800889f8 NtUserGetAltTabInfo
    0x180088a00 NtUserSetClassLong
    0x180088a08 NtUserGetMessage
    0x180088a10 NtUserGetKeyboardLayoutName
    0x180088a18 NtUserDrawCaptionTemp
    0x180088a20 NtUserSetProp
    0x180088a28 NtUserVkKeyScanEx
    0x180088a30 NtUserCallMsgFilter
    0x180088a38 NtUserCallHwndLockSafe
    0x180088a40 NtUserSetImeOwnerWindow
    0x180088a48 NtUserNotifyIMEStatus
    0x180088a50 NtUserUpdateInputContext
    0x180088a58 NtUserCountClipboardFormats
    0x180088a60 NtUserGetPriorityClipboardFormat
    0x180088a68 NtUserGetClipboardOwner
    0x180088a70 NtUserGetClipboardSequenceNumber
```

Figure 3 – Execution of the IAT table extraction script

Select the PE you want to extract the IAT addresses from, but make sure it is a valid file, first try opening it in a PE viewer to get more detailed information, after that you can use these scripts if you want to work with more advanced techniques to work with a specific function.

PEView Download:http://wjradburn.com/software/

Download extract_iat:https://github.com/CyberSecurityUP/Windows-API-for-Red-Team/blob/main/extract_iat.py

IAT table in the attack context

Techniques involving the Import Address Table (IAT) in detection evasion and malware development are sophisticated and encompass multiple methods for manipulating how programs access and execute external code. These techniques are essential to understand for both security developers and IT professionals focused on cyber defense. Here are some of the most relevant techniques:

1. Use of Ordinals

In the context of DLLs, in addition to importing functions by name, it is possible to import functions by ordinal, which is basically an index associated with each function in a DLL. Using ordinals instead of function names can complicate malware analysis and detection because external observers need to know which functions correspond to which ordinals in each DLL, which can vary between different versions of the same DLL.

2. Syscalls

Syscalls (system calls) are another low-level method used by malware to interact directly with the operating system, bypassing standard Windows APIs and, consequently, IAT. By making syscalls directly, malware can avoid detections based on monitoring common API calls, as it does not rely on IAT-listed imports.

3. Hooks

"Hooking" is a technique that involves intercepting function calls or system messages/events. Malware can use IAT hooking to redirect legitimate function calls to malicious functions without changing the host program code. This can be used to capture data, modify program behaviors, or disable security features.

4. Unhooks

To evade detection by security solutions that use hooking to monitor program behavior, some malware can execute "unhooks". This technique involves restoring the original IAT entries to their pre-hook states, temporarily during the execution of malicious activities, to appear less suspicious to monitoring tools.

5. IAT Patching

Malware can modify the IAT of a running process to point to its own malicious functions instead of the original libraries. This allows the malware to intercept and manipulate data or behavior from the infected program.

Protection and Detection

To protect against these techniques, defense systems must implement runtime behavior monitoring and analysis, IAT integrity checking, and use of anti-hooking technologies. Additionally, forensic analysis and reverse engineering remain valuable tools for understanding how malware uses these techniques and developing effective protective measures.

IAT TABLE AND ORDINALS

INTRODUCTION

Ordinals are an alternative way to reference functions or variables exported by a dynamic link library (DLL) in Windows. Instead of using function names, which is the more common and readable method, ordinals use sequential integers as identifiers for exported functions.

How Ordinals Work

When a DLL is created, the compiler or linker can assign each exported function a unique ordinal number within that DLL. This number is a zero- or one-based index that does not necessarily follow the alphabetical order of function names. The developer can also manually specify these numbers if they wish to control the ordinal numbering scheme.

The exported functions can then be imported by other modules (executables or other DLLs) using these ordinals. This eliminates the need to use name strings, which can result in a small performance gain in the application loading process, as string matching is more expensive than comparing integers.

Advantages of Using Ordinals

1. **Efficiency**: Searching by a numeric index is generally faster than searching by a function name string. This can improve software loading times in situations where loading performance is critical.

2. **Obfuscation**: Using ordinals can serve as a form of obfuscation, making it more difficult for analysts to understand the purpose of a function without deeper analysis, as the descriptive name of the function is not available.

Disadvantages and Risks

1. **Maintainability**: Using ordinals can make code more difficult to maintain, as reviewing or updating code to use or not use certain DLLs becomes more complex without function names to reference.

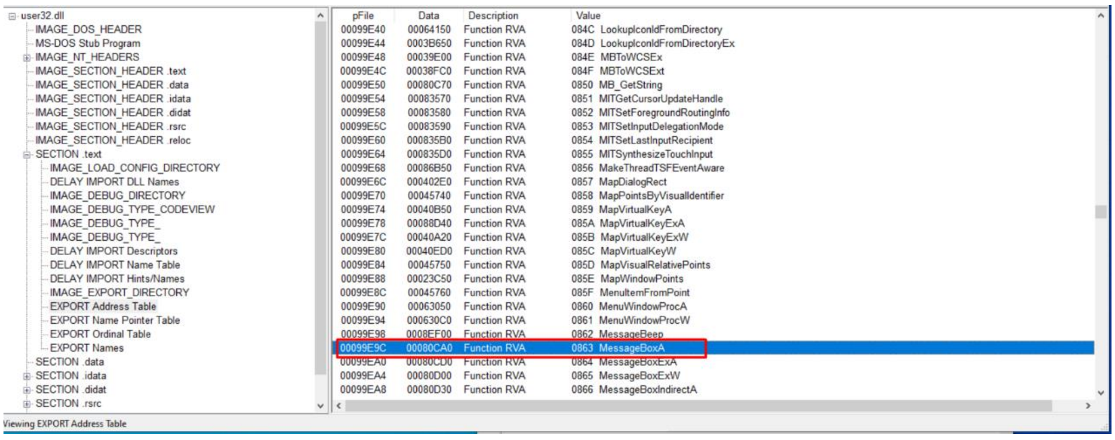
2. **Compatibility**: If a DLL is updated and ordinals are changed (for example, functions are added or removed), this may break compatibility with existing applications that expect a specific function in a specific ordinal.

Ordinals in Security and Malware

In cybersecurity, ordinals are often used in malware techniques to make analysis and detection difficult. For example, malware can import critical functions using ordinals to hide its true intentions and avoid detection based on analysis of import strings known to be malicious.

Practical example:

Let's see a simple example of how we can work with ordinals, I will use a Windows API called MessageBoxA



```
user32.dll
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .idata
  IMAGE_SECTION_HEADER .didat
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .reloc
  SECTION .text
    IMAGE_LOAD_CONFIG_DIRECTORY
    DELAY IMPORT DLL Names
    IMAGE_DEBUG_DIRECTORY
    IMAGE_DEBUG_TYPE_CODEVIEW
    IMAGE_DEBUG_TYPE_
    IMAGE_DEBUG_TYPE_
    DELAY IMPORT Descriptors
    DELAY IMPORT Name Table
    DELAY IMPORT Hints/Names
    IMAGE_EXPORT_DIRECTORY
    EXPORT Address Table
    EXPORT Name Pointer Table
    EXPORT Ordinal Table
    EXPORT Names
  SECTION .data
  SECTION .idata
  SECTION .didat
  SECTION .rsrc
Viewing EXPORT Address Table
```

```
pFile     Data      Description   Value
00099E40  00064150  Function RVA  084C  LookupIconIdFromDirectory
00099E44  0003B650  Function RVA  084D  LookupIconIdFromDirectoryEx
00099E48  00039E00  Function RVA  084E  MBToWCSEx
00099E4C  00038FC0  Function RVA  084F  MBToWCSExt
00099E50  00080C70  Function RVA  0850  MB_GetString
00099E54  00083570  Function RVA  0851  MITGetCursorUpdateHandle
00099E58  00083580  Function RVA  0852  MITSetForegroundRoutingInfo
00099E5C  00083590  Function RVA  0853  MITSetInputDelegationMode
00099E60  000835B0  Function RVA  0854  MITSetLastInputRecipient
00099E64  000835D0  Function RVA  0855  MITSynthesizeTouchInput
00099E68  00086B50  Function RVA  0856  MakeThreadTSFEventAware
00099E6C  000402E0  Function RVA  0857  MapDialogRect
00099E70  00045740  Function RVA  0858  MapPointsByVisualIdentifier
00099E74  00040B50  Function RVA  0859  MapVirtualKeyA
00099E78  00088D40  Function RVA  085A  MapVirtualKeyExA
00099E7C  00040A20  Function RVA  085B  MapVirtualKeyExW
00099E80  00040ED0  Function RVA  085C  MapVirtualKeyW
00099E84  00045750  Function RVA  085D  MapVisualRelativePoints
00099E88  00023C50  Function RVA  085E  MapWindowPoints
00099E8C  00045760  Function RVA  085F  MenuItemFromPoint
00099E90  00063050  Function RVA  0860  MenuWindowProcA
00099E94  000630C0  Function RVA  0861  MenuWindowProcW
00099E98  0008EF00  Function RVA  0862  MessageBeep
00099E9C  00080CA0  Function RVA  0863  MessageBoxA
00099EA0  00080CD0  Function RVA  0864  MessageBoxExA
00099EA4  00080D00  Function RVA  0865  MessageBoxExW
00099EA8  00080D30  Function RVA  0866  MessageBoxIndirectA
```

Figure 4 – Using the IAT table, select the desired function

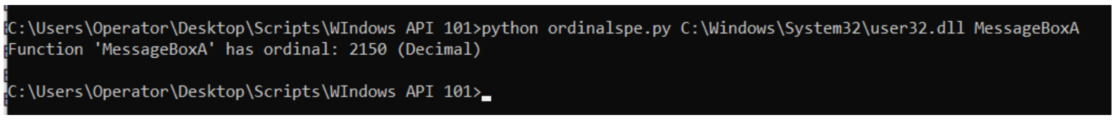
In this case I chose MessageBoxA. Because each function exported by a DLL can be identified by either a numeric ordinal or a name. Functions can also be imported from a DLL using these ordinals or names. The ordinal indicates the position of the function pointer in the DLL's export address table. It is common for internal functions to be exported only by ordinals.

After selecting the function, we will use a script to extract the ordinal of a function.

Download:

https://github.com/CyberSecurityUP/Windows-API-for-Red-Team/blob/main/ordinal_pe.py

Figure 5 – Ordinal Result of the MessageBoxA function



```
C:\Users\Operator\Desktop\Scripts\WIndows API 101>python ordinalspe.py C:\Windows\System32\user32.dll MessageBoxA
Function 'MessageBoxA' has ordinal: 2150 (Decimal)

C:\Users\Operator\Desktop\Scripts\WIndows API 101>
```

An interesting detail is that although the decimal gave 2150, it may not be the exact ordinal, so always skip 2 places up or down, for example: Result was 2150, test 2148 or 2152.

Now let's write our code:

I will use the C++ language, however I always like to adapt the same code to other languages such as Rust, C# (PInvoke and DInvoke), Python and Golang.

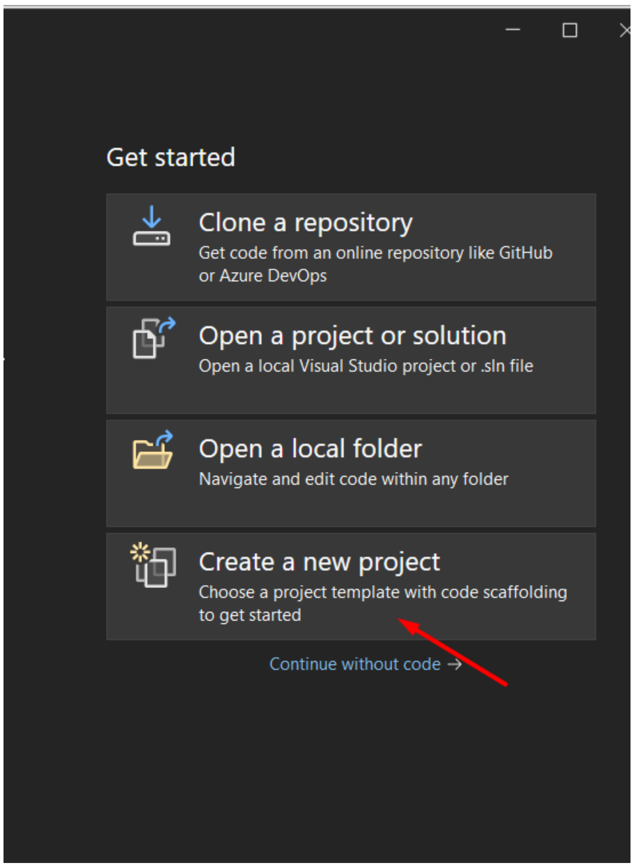
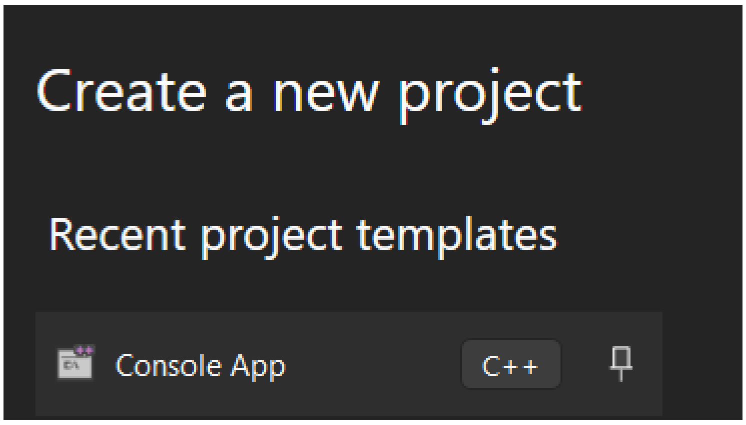

Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

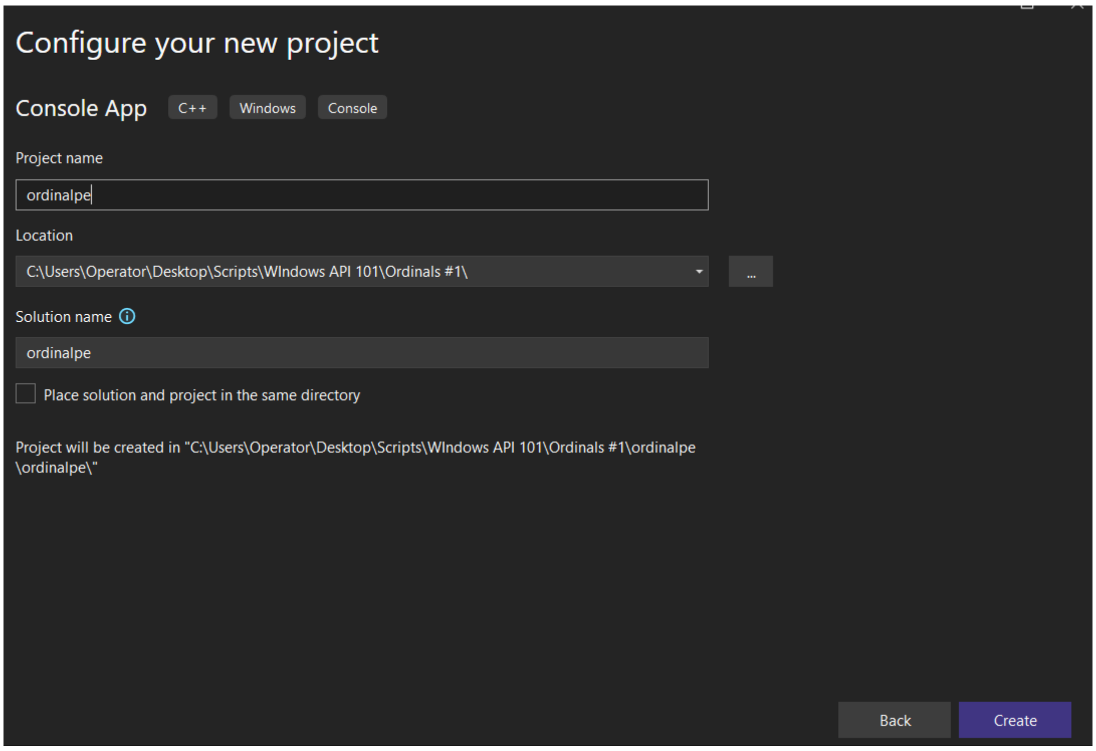
Figure 6 – Create a Project


Create a new project

Recent project templates

Console App C++

Figure 7 – Select a template called Console APP C++


Configure your new project

Console App C++ Windows Console

Project name

ordinalpe

Location

C:\Users\Operator\Desktop\Scripts\WIndows API 101\Ordinals #1\

Solution name

ordinalpe

Place solution and project in the same directory

Project will be created in "C:\Users\Operator\Desktop\Scripts\WIndows API 101\Ordinals #1\ordinalpe\ordinalpe\"

Back Create

Figure 8 – Define a name for your project

After creating the project, you can work with the example below according to the ordinal you obtained in your MessageBoxA

Writing our code:

```
#include <windows.h>
#include <iostream>
#include <stdio.h>


intmain() {

 HMODULEhModule =LoadLibrary(L"user32.dll");
 if(!hModule) {
std::cerr<< "Failed to load user32.dll!" <<std::endl;
 return1;
}
```

- **windows.h**: Includes the Windows API, required for functions such as LoadLibrary and GetProcAddress.
- **iostream**: Allows the use of data inputs and outputs in C++, in this case used to print error messages.
- **stdio.h**: Typically used for standard input and output. In the code provided, it is not explicitly used and could be removed.
- **Int main:***This is the program entry point, where execution begins.*
- **LoadLibrary:**Loads the specified DLL (user32.dll in this case), which contains many of the basic Windows user interface functions, including MessageBoxA. If LoadLibrary fails to load the DLL, an error message is printed and the program ends with error code 1.
- **HMODULE:**Type of handle used to load modules; hModule is a handle to the loaded module.

Get the function address by Ordinal

```
 typedef int(WINAPI*MsgBoxFunc)(HWND,LPCSTR,LPCSTR,UINT);
 MsgBoxFuncOrdinalBoxA = (MsgBoxFunc)GetProcAddress(hModule,
(LPCSTR)2150);

 if(!OrdinalBoxA) {
std::cerr<< "Failed to locate the function!" <<std::endl;
FreeLibrary(hModule);
 return1;
}

OrdinalBoxA(NULL,"Hello, World!","Test
MessageBoxA",MB_OK|MB_ICONINFORMATION);

FreeLibrary(hModule);
 return0;

}
```

- **GetProcAddress**: Gets the address of the function in the loaded DLL, specified by the ordinal 2150. This number is the numeric identifier of the MessageBoxA function.
- **typedef**: Defines MsgBoxFunc as a pointer to function of type MessageBoxA.
- If the function is not found (i.e., OrdinalBoxA is nullptr), it prints an error message, flushes the loaded DLL, and terminates the program in error.

- **OrdinalBoxA:**Once the address has been successfully retrieved, the MessageBoxA function is called using the pointer to the function.

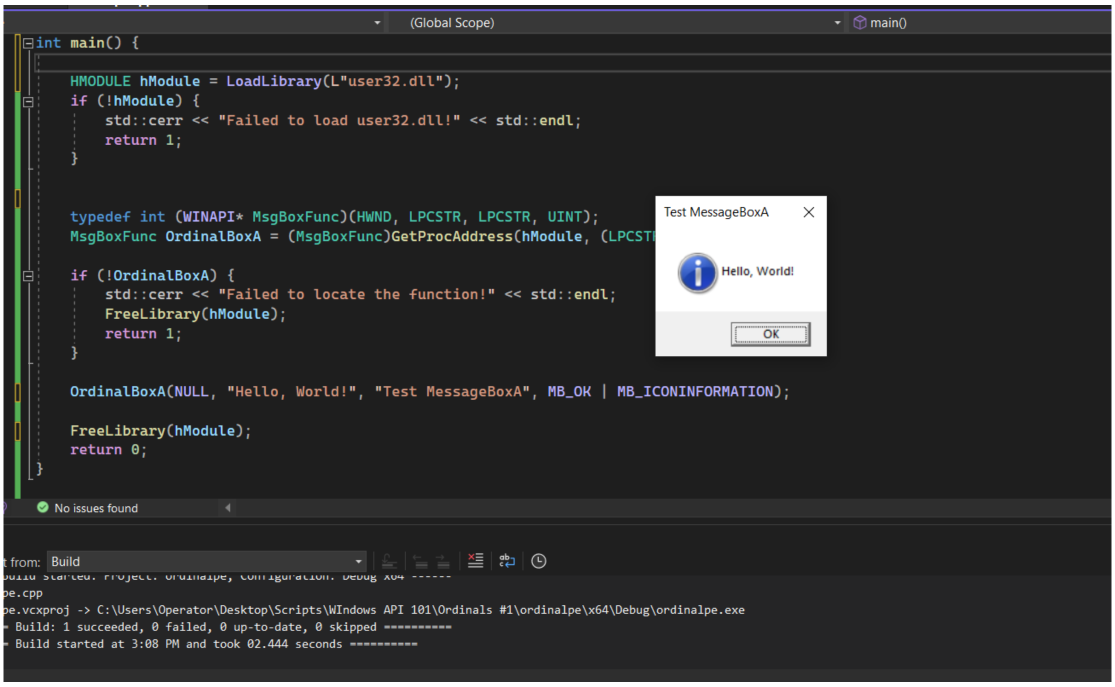
- The parameters passed are NULL for the window handle (parent window), the message "Hello, World!", the title "Test MessageBoxA" and the flags MB_OK | MB_ICONINFORMATION for the type of buttons and icon to be displayed in the message box.

- **FreeLibrary:**Releases the loaded module, in this case user32.dll.

- **return 0;:**Ends program execution normally, indicating that there were no errors.

- I could also hide the MessageBoxA and user32.dll functions so that they are not listed in the IAT table as well

Now let's see what the result of all this is in the end.



```
int main() {

    HMODULE hModule = LoadLibrary(L"user32.dll");
    if (!hModule) {
        std::cerr << "Failed to load user32.dll!" << std::endl;
        return 1;
    }


    typedef int (WINAPI* MsgBoxFunc)(HWND, LPCSTR, LPCSTR, UINT);
    MsgBoxFunc OrdinalBoxA = (MsgBoxFunc)GetProcAddress(hModule, (LPCST

    if (!OrdinalBoxA) {
        std::cerr << "Failed to locate the function!" << std::endl;
        FreeLibrary(hModule);
        return 1;
    }

    OrdinalBoxA(NULL, "Hello, World!", "Test MessageBoxA", MB_OK | MB_ICONINFORMATION);

    FreeLibrary(hModule);
    return 0;
}
```

Test MessageBoxA

Hello, World!

```
pe.cpp
pe.vcxproj -> C:\Users\Operator\Desktop\Scripts\WIndows API 101\Ordinals #1\ordinalpe\x64\Debug\ordinalpe.exe
= Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
= Build started at 3:08 PM and took 02.444 seconds ==========
```

Figure 9 – Result of code execution

We have our Hello World using an ordinal, instead of using the MessageBoxA library directly.

Full code:

```
#include <windows.h>
#include <iostream>
#include <stdio.h>


intmain() {
 HMODULEhModule =LoadLibrary(L"user32.dll");
 if(!hModule) {
std::cerr<< "Failed to load user32.dll!" <<std::endl;
 return1;
}


 typedef int(WINAPI*MsgBoxFunc)(HWND,LPCSTR,LPCSTR,UINT);
 MsgBoxFuncOrdinalBoxA = (MsgBoxFunc)GetProcAddress(hModule,
(LPCSTR)2150);

 if(!OrdinalBoxA) {
std::cerr<< "Failed to locate the function!" <<std::endl;
FreeLibrary(hModule);
 return1;
}

OrdinalBoxA(NULL,"Hello, World!","Test
MessageBoxA",MB_OK|MB_ICONINFORMATION);

FreeLibrary(hModule);
 return0;
}
```

How do I improve this code?

From a Red Team perspective, especially when considering malware development and evasion techniques, there are several improvements and techniques that can be incorporated into the code to increase stealth and make detection by security solutions more difficult. Here are some suggestions for improving the code for these purposes:

1. Streamlining DLL Loading

Avoiding directly calling well-known functions like LoadLibrary can help you dodge behavior monitoring tools that track common use of APIs to load DLLs. Instead, you can use direct memory execution techniques (Reflective DLL Injection) or load the DLL in a more unobtrusive way:

- **Load the DLL from an unconventional location or resource**: For example, you could extract the DLL from a resource embedded in the executable or from an encrypted data area.

2. Use of Direct Syscalls

For critical functions that may be monitored, such as creating MessageBoxes, considering the direct use of syscalls may be a stealthier approach. Syscalls do not pass directly through API Wrappers, so they are less likely to be caught by API monitoring-based security solutions:

- **Implement syscalls manually**: This can be done by getting the corresponding syscall number for the desired function and executing it directly via in-line assembly in the C++ code.

3. Obfuscation of Strings and Code

Obfuscating strings (such as DLL names and other string constants) and code structures can help avoid simple signature-based detections:

- **Encrypt strings**: Store strings in an encrypted manner and decrypt them at runtime.
- **Code obfuscation techniques**: Changing program logic in ways that do not affect functionality but modify the appearance of the binary.

4. Changing Ordinals and Dynamic Analysis

If possible, don't rely on a fixed ordinal, which can change between versions or configurations. A dynamic analysis of the environment can help determine the correct ordinal or even the function name if the ordinals are unreliable:

- **Discover ordinals or names at runtime**: Develop a mechanism that, at run time, reads the DLL export table and finds the correct ordinal based on heuristics or partial name comparisons.

5. Avoiding Common API Points

Replace high-level API calls with their lower-level counterparts, where applicable, or redirect these calls to other, less suspicious functions that eventually perform the required task.

6. Self-modification

Consider self-modification techniques where code changes its own execution or logic to avoid static patterns that can be detected.

7. Monitoring and Adapting to the Environment

Detect the presence of security tools and adapt code behavior accordingly. For example, if a sandbox is detected, the program may choose not to perform certain actions or simulate benign behaviors.

These techniques require in-depth knowledge of the internal mechanisms of the Windows operating system, as well as low-level programming experience. Each of these increases the complexity of the malware, but also increases its chances of evading detection and carrying out its functions effectively and discreetly.

Terms and definitions

For you to practice at home:

- Create code that calls the MessageBoxA API without ordinal

Term	Description
DLL	Dynamic Link Library (DLL) is a file containing code and data that can be used by multiple programs simultaneously.
Linker	Tool that combines multiple object files generated by a compiler into a single executable or DLL.
Compiler	Program that translates source code written in a high-level programming language into machine code.
MessageBoxA	It is a Windows API function that displays a message box with customizable text, buttons, and icon, and returns a response based on user interaction.
GetProcAddress	It is a Windows API function that returns the address of a function or variable exported by a DLL, identified by its name or ordinal.
Syscall	A syscall is a fundamental operation that programs use to request a service from the operating system kernel, such as file manipulation or network communication.

- Create code that hides the MessageBoxA function and user32.dll from appearing in the IAT table

Bibliographic references

rioasmara. (2020, November 14). Hide API Call Strings with Ordinals. Cyber Security Architect | Red/Blue Teaming | Exploit/Malware Analysis. https://rioasmara.com/2020/11/15/hide-api-call-strings-with-ordinals/#:~:text=The%20ordinal%20represents%20the%20position

GrantMeStrength. (nd). Windows API index - Win32 apps. Learn.microsoft.com.https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

NTAPI Undocumented Functions. (nd). Undocumented.ntinternals.net.http://undocumented.ntinternals.net/

Yosifovich, P. (2023). Windows Native API Programming. In leanpub.com. Leanpub.https://leanpub.com/windowsnativeapiprogramming

Santos, JA, & Pires, F. (2025). Defense Evasion Techniques: A comprehensive guide to defense evasion tactics for Red Teams and Penetration Testers. In Amazon. Packt Publishing.https://www.amazon.com.br/Defense-Evasion-Techniques-comprehensive-Penetration-ebook/dp/B0C5MRV617

Credits for customized Windows,João Paulo de Andrade