Introduction to Windows API

Introduction

The Windows API provides native functionality to interact with key components of the Windows operating system. The API is widely used by many, including red teamers, threat actors, blue teamers, software developers, and solution providers.

The API can integrate seamlessly with the Windows system, offering its range of use cases. You may see the Win32 API being used for offensive tool and malware development, EDR (Endpoint Detection & Response) engineering, and general software applications. For more information about all of the use cases for the API, check out the Windows API Index.

Learning Objectives

- Understand what the Windows API is, its use cases, and how it interacts with the OS subsystems
- Learn how to implement the Windows API in different languages
- Understand how the Windows API can be used from a malicious perspective and break down several practical case studies

Before beginning this room, we recommend general familiarity with operating system architecture. Basic programming knowledge is also recommended but not required.

This room aims to teach the Windows API at a fundamental level. We will briefly cover implementations of the Win32 API, but we will focus on why and where API calls are used.

Subsystems and Hardware Interactions

Programs often need to access or modify Windows subsystems or hardware but are restricted to maintain machine stability. To solve this problem, Microsoft released the Win32 API, a library to interface between user-mode applications and the kernel.

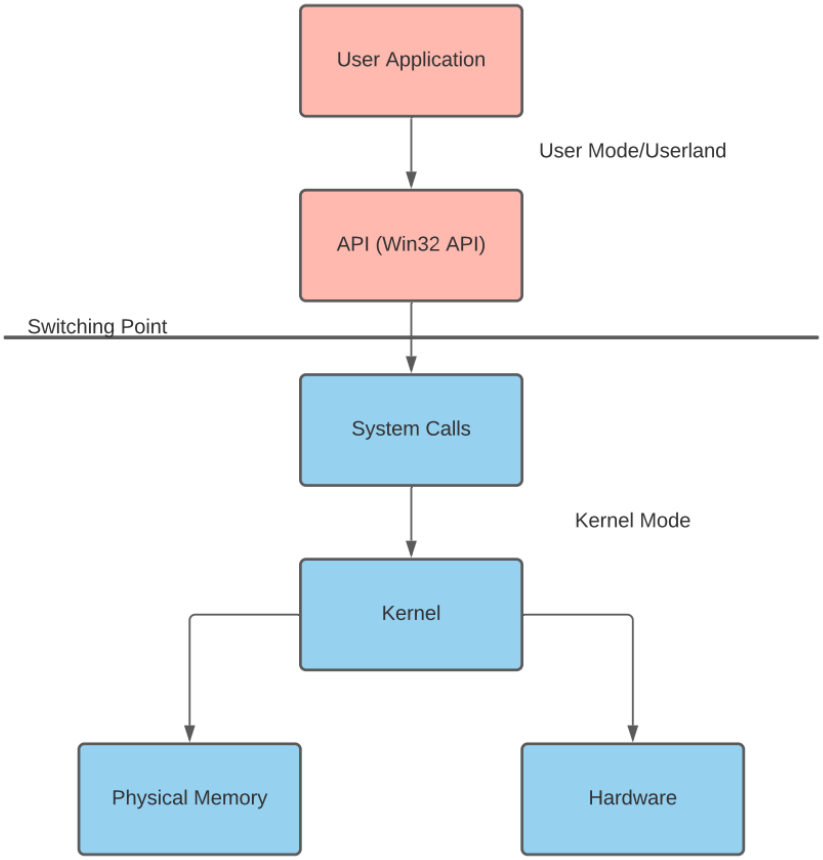
Windows distinguishes hardware access by two distinct modes: user and kernel mode. These modes determine the hardware, kernel, and memory access an application or driver is permitted. API or system calls interface between each mode, sending information to the system to be processed in kernel mode.

User Mode	Kernel Mode

No direct hardware access	Direct hardware access
Access to "owned" memory locations	Access to entire physical memory

For more information about memory management, check out Windows Internals.

Below is a visual representation of how a user application can use API calls to modify kernel components.



When looking at how languages interact with the Win32 API, this process can become further warped; the application will go through the language runtime before going through the API.

For more information about the runtime, check out Runtime Detection Evasion.

****

Answer the questions below:

Does a process in the user mode have direct hardware access? (Y/N)
Answer: N

Does launching an application as an administrator open the process in kernel mode? (Y/N)
Answer: N

Components of the Windows API

The Win32 API, more commonly known as the Windows API, has several dependent components that are used to define the structure and organization of the API.

Let's break the Win32 API up via a top-down approach. We'll assume the API is the top layer and the parameters that make up a specific call are the bottom layer. In the table below, we will describe the top-down structure at a high level and dive into more detail later.

Layer	**Explanation**
API	A top-level/general term or theory used to describe any call found in the win32 API structure.
Header Files or Imports	Defines libraries to be imported at run-time, defined by header files or library imports. Uses pointers to obtain the function address.
Core DLLs	A group of four DLLs that define call structures. (KERNEL32, USER32, and ADVAPI32). These DLLs define kernel and user services that are not contained in a single subsystem.
Supplemental DLLs	Other DLLs defined as part of the Windows API. Controls separate subsystems of the Windows OS. ~36 other defined DLLs. (NTDLL, COM, FVEAPI, etc.)
Call Structures	Defines the API call itself and parameters of the call.
API Calls	The API call used within a program, with function addresses obtained from pointers.
In/Out Parameters	The parameter values that are defined by the call structures.

Let's expand these definitions; in the next task, we will discuss importing libraries, the core header file, and the call structure. In task 4, we will dive deeper into the calls, understanding where and how to digest call parameters and variants.

****

Answer the questions below:

What header file imports and defines the User32 DLL and structure?

winuser.h – user32.dll: user services, inline resource macro(e.g. MAKEINTRESOURCE macro [8]), inline dialog macro(e.g. DialogBox function [9]). [10]

Answer: winuser.h

What parent header file contains all other required child and core header files?

Answer: windows.h

OS Libraries

Each API call of the Win32 library resides in memory and requires a pointer to a memory address. The process of obtaining pointers to these functions is obscured because of ASLR (Address Space Layout Randomization) implementations; each language or package has a unique procedure to overcome ASLR. Throughout this room, we will discuss the two most popular implementations: P/Invoke and the Windows header file.

In this task, we will take a deep dive into the theory of how both of these implementations work, and in future tasks, we will put them to practical use.

Windows Header File

Microsoft has released the Windows header file, also known as the Windows loader, as a direct solution to the problems associated with ASLR's implementation. Keeping the concept at a high level, at runtime, the loader will determine what calls are being made and create a thunk table to obtain function addresses or pointers.

Luckily, we do not have to dive deeper than that to continue working with API calls if we do not desire to do so.

Once the windows.h file is included at the top of an unmanaged program; any Win32 function can be called.

We will cover this concept at a more practical level in task 6.

P/Invoke

Microsoft describes P/Invoke or platform invoke as “a technology that allows you to access structs, callbacks, and functions in unmanaged libraries from your managed code.”

P/invoke provides tools to handle the entire process of invoking an unmanaged function from managed code or, in other words, calling the Win32 API. P/invoke will kick off by importing the desired DLL that contains the unmanaged function or Win32 API call. Below is an example of importing a DLL with options.

```
using System;
using System.Runtime.InteropServices;

public class Program
{
[DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
...
}
```

In the above code, we are importing the DLL user32 using the attribute: DLLImport.

Note: a semicolon is not included because the p/invoke function is not yet complete. In the second step, we must define a managed method as an external one. The extern keyword will inform the runtime of the specific DLL that was previously imported. Below is an example of creating the external method.

```
using System;
using System.Runtime.InteropServices;

public class Program
{
...
private static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption, uint uType);
}
```

Now we can invoke the function as a managed method, but we are calling the unmanaged function!

We will cover this concept at a more practical level in task 7.

****

Answer the questions below:

What overarching namespace provides P/Invoke to .NET?

Answer: System

What memory protection solution obscures the process of importing API calls?
Answer: ASLR

API Call Structure

API calls are the second main component of the Win32 library. These calls offer extensibility and flexibility that can be used to meet a plethora of use cases. Most Win32 API calls are well documented under the Windows API documentation and pinvoke.net.

In this task, we will take an introductory look at naming schemes and in/out parameters of API calls.

API call functionality can be extended by modifying the naming scheme and appending a representational character. Below is a table of the characters Microsoft supports for its naming scheme.

Character	Explanation
A	Represents an 8-bit character set with ANSI encoding
W	Represents a Unicode encoding
Ex	Provides extended functionality or in/out parameters to the API call

For more information about this concept, check out the Microsoft documentation.

Each API call also has a pre-defined structure to define its in/out parameters. You can find most of these structures on the corresponding API call document page of the Windows documentation, along with explanations of each I/O parameter.

Let's take a look at the WriteProcessMemory API call as an example. Below is the I/O structure for the call obtained here.

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

For each I/O parameter, Microsoft also explains its use, expected input or output, and accepted values.

Even with an explanation, determining these values can sometimes be challenging for particular calls. We suggest always researching and finding examples of API call usage before using a call in your code.

****

Answer the questions below:

Which character appended to an API call represents an ANSI encoding?
Answer: A

Which character appended to an API call represents extended functionality?
Answer: Ex

What is the memory allocation type of 0x00080000 in the VirtualAlloc API call?

MEM_RESET 0x00080000	Indicates that data in the memory range specified by *lpAddress* and *dwSize* is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block wil be used again later, so it should not be decommitted. This value cannot be used with any other value. Using this value does not guarantee that the range operated on with **MEM_RESET** will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it. When you specify **MEM_RESET**, the **VirtualAlloc** function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**. **VirtualAlloc** returns an error if you use **MEM_RESET** and the rang of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

Answer: MEM_RESET

C API Implementations

Microsoft provides low-level programming languages such as C and C++ with a pre-configured set of libraries that we can use to access needed API calls.

The windows.h header file, as discussed in task 4, is used to define call structures and obtain function pointers. To include the windows header, prepend the line below to any C or C++ program.

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

Let's jump right into creating our first API call. As our first objective, we aim to create a pop-up window with the title: "Hello THM!" using CreateWindowExA. To reiterate what was covered in task 5, let's observe the in/out parameters of the call.

```
HWND CreateWindowExA(
  [in]           DWORD     dwExStyle, // Optional windows styles
  [in, optional] LPCSTR    lpClassName, // Windows class
  [in, optional] LPCSTR    lpWindowName, // Windows text
  [in]           DWORD     dwStyle, // Windows style
  [in]           int       X, // X position
  [in]           int       Y, // Y position
  [in]           int       nWidth, // Width size
  [in]           int       nHeight, // Height size
  [in, optional] HWND      hWndParent, // Parent windows
  [in, optional] HMENU     hMenu, // Menu
  [in, optional] HINSTANCE hInstance, // Instance handle
  [in, optional] LPVOID    lpParam // Additional application data
);
```

Let's take these pre-defined parameters and assign values to them. As mentioned in task 5, each parameter for an API call has an explanation of its purpose and potential values. Below is an example of a complete call to CreateWindowsExA.

```
HWND hwnd = CreateWindowsEx(
    0,
    CLASS_NAME,
    L"Hello THM!",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
    );
```

We've defined our first API call in C! Now we can implement it into an application and use the functionality of the API call. Below is an example application that uses the API to create a small blank window.

```
BOOL Create(
        PCWSTR lpWindowName,
        DWORD dwStyle,
        DWORD dwExStyle = 0,
        int x = CW_USEDEFAULT,
        int y = CW_USEDEFAULT,
        int nWidth = CW_USEDEFAULT,
        int nHeight = CW_USEDEFAULT,
        HWND hWndParent = 0,
        HMENU hMenu = 0
        )
    {
        WNDCLASS wc = {0};

        wc.lpfnWndProc   = DERIVED_TYPE::WindowProc;
        wc.hInstance     = GetModuleHandle(NULL);
        wc.lpszClassName = ClassName();

        RegisterClass(&wc);

        m_hwnd = CreateWindowEx(
            dwExStyle, ClassName(), lpWindowName, dwStyle, x, y,
            nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this
            );

        return (m_hwnd ? TRUE : FALSE);
    }
```

If successful, we should see a window with the title “Hello THM!”.

As demonstrated throughout this task, low-level languages make it very easy to define an API call quickly. Because of the ease of use and extensibility, C-based languages are the most popular among threat actors and vendors alike.

****

Answer the questions below:

Do you need to define a structure to use API calls in C? (Y/N)
Answer: N

.NET and PowerShell Implementations

As discussed in task 4, P/Invoke allows us to import DLLs and assign pointers to API calls.

To understand how P/Invoke is implemented, let's jump right into it with an example below and discuss individual components afterward.

```
class Win32 {
    [DllImport("kernel32")]
    public static extern IntPtr GetComputerNameA(StringBuilder lpBuffer, ref uint lpnSize);
}
```

The class function stores defined API calls and a definition to reference in all future methods.

The library in which the API call structure is stored must now be imported using DllImport. The imported DLLs act similar to the header packages but require that you import a specific DLL with the API call you are looking for. You can reference the API index or pinvoke.net to determine where a particular API call is located in a DLL.

From the DLL import, we can create a new pointer to the API call we want to use, notably defined by intPtr. Unlike other low-level languages, you must specify the in/out parameter structure in the pointer. As discussed in task 5, we can find the in/out parameters for the required API call from the Windows documentation.

Now we can implement the defined API call into an application and use its functionality. Below is an example application that uses the API to get the computer name and other information of the device it is run on.

```
class Win32 {
    [DllImport("kernel32")]
    public static extern IntPtr GetComputerNameA(StringBuilder lpBuffer, ref uint lpnSize);
}

static void Main(string[] args) {
    bool success;
    StringBuilder name = new StringBuilder(260);
    uint size = 260;
    success = GetComputerNameA(name, ref size);
    Console.WriteLine(name.ToString());
}
```

If successful, the program should return the computer name of the current device.

Now that we've covered how it can be accomplished in .NET let's look at how we can adapt the same syntax to work in PowerShell.

Defining the API call is almost identical to .NET's implementation, but we will need to create a method instead of a class and add a few additional operators.

```
$MethodDefinition = @"
    [DllImport("kernel32")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
    [DllImport("kernel32")]
    public static extern IntPtr GetModuleHandle(string lpModuleName);
    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
"@;
```

The calls are now defined, but PowerShell requires one further step before they can be initialized. We must create a new type for the pointer of each Win32 DLL within the method definition. The function Add-Type will drop a temporary file in the /temp directory and compile needed functions using csc.exe. Below is an example of the function being used.

```
$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -NameSpace 'Win32' -PassThru;
```

We can now use the required API calls with the syntax below.

[Win32.Kernel32]::<Imported Call>()

****

Answer the questions below:

What method is used to import a required DLL?
Answer: DllImport

What type of method is used to reference the API call to obtain a struct?
Answer: External

Commonly Abused API Calls

Several API calls within the Win32 library lend themselves to be easily leveraged for malicious activity.

Several entities have attempted to document and organize all available API calls with malicious vectors, including SANs and MalAPI.io.

While many calls are abused, some are seen in the wild more than others. Below is a table of the most commonly abused API organized by frequency in a collection of samples.

API	Explanation
LoadLibraryA	Maps a specified DLL into the address space of the calling process
GetUserNameA	Retrieves the name of the user associated with the current thread
GetComputerNameA	Retrieves a NetBIOS or DNS name of the local computer
GetVersionExA	Obtains information about the version of the operating system currently running
GetModuleFileNameA	Retrieves the fully qualified path for the file of the specified module and process
GetStartupInfoA	Retrieves contents of STARTUPINFO structure (window station, desktop, standard handles, and appearance of a process)
GetModuleHandle	Returns a module handle for the specified module if mapped into the calling process's address space
GetProcAddress	Returns the address of a specified exported DLL function
VirtualProtect	Changes the protection on a region of memory in the virtual address space of the calling process

In the next task, we will take a deep dive into how these calls are abused in a case study of two malware samples.

****

Answer the questions below:

In the next task, we will take a deep dive into how these calls are abused in a case study of two malware samples.
Answer: GetProcAddress

Which API call imports a specified DLL into the address space of the calling process?
Answer: LoadLibrary

Malware Case Study

Now that we understand the underlying implementations of the Win32 library and commonly abused API calls, let's break down two malware samples and observe how their calls interact.

In this task, we will be breaking down a C# keylogger and shellcode launcher.

KeyLogger

To begin analyzing the keylogger, we need to collect which API calls and hooks it is implementing. Because the keylogger is written in C#, it must use P/Invoke to obtain pointers for each call. Below is a snippet of the p/invoke definitions of the malware sample source code.

```
[DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
private static extern IntPtr SetWindowsHookEx(int idHook, LowLevelKeyboardProc lpfn, IntPtr hMod, uint dwThreadId);
[DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern bool UnhookWindowsHookEx(IntPtr hhk);
[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
private static extern IntPtr GetModuleHandle(string lpModuleName);
private static int WHKEYBOARDLL = 13;
[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
private static extern IntPtr GetCurrentProcess();
```

Below is an explanation of each API call and its respective use.

API Call	Explanation
SetWindowsHookEx	Installs a memory hook into a hook chain to monitor for certain events
UnhookWindowsHookEx	Removes an installed hook from the hook chain
GetModuleHandle	Returns a module handle for the specified module if mapped into the calling process's address space
GetCurrentProcess	Retrieves a pseudo handle for the current process.

To maintain the ethical integrity of this case study, we will not cover how the sample collects each keystroke. We will analyze how the sample sets a hook on the current process. Below is a snippet of the hooking section of the malware sample source code.

```
public static void Main() {
    _hookID = SetHook(_proc);
    Application.Run();
    UnhookWindowsHookEx(_hookID);
    Application.Exit();
}
private static IntPtr SetHook(LowLevelKeyboardProc proc) {
    using (Process curProcess = Process.GetCurrentProcess()) {
        return SetWindowsHookEx(WHKEYBOARDLL, proc, GetModuleHandle(curProcess.ProcessName), 0);
    }
}
```

Let's understand the objective and procedure of the keylogger, then assign their respective API call from the above snippet.

Using the [Windows API documentation] and the context of the above snippet, begin analyzing the keylogger, using questions 1 - 4 as a guide to work through the sample.

Shellcode Launcher

To begin analyzing the shellcode launcher, we once again need to collect which API calls it is implementing. This process should look identical to the previous case study. Below is a snippet of the p/invoke definitions of the malware sample source code.

```
private static UInt32 MEM_COMMIT = 0x1000;
private static UInt32 PAGE_EXECUTE_READWRITE = 0x40;
[DllImport("kernel32")]
private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocationType, UInt32 flProtect);
[DllImport("kernel32")]
private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);
[DllImport("kernel32")]
private static extern IntPtr CreateThread(UInt32 lpThreadAttributes, UInt32 dwStackSize, UInt32 lpStartAddress, IntPtr param, UInt32 dwCreationFlags, ref UInt32 lpThreadId);
```

Below is an explanation of each API call and its respective use.

API Call	**Explanation**
VirtualAlloc	Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process.
WaitForSingleObject	Waits until the specified object is in the signaled state or the time-out interval elapses
CreateThread	Creates a thread to execute within the virtual address space of the calling process

We will now analyze how the shellcode is written to and executed from memory.

```
UInt32 funcAddr = VirtualAlloc(0, (UInt32)shellcode.Length, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
Marshal.Copy(shellcode, 0, (IntPtr)(funcAddr), shellcode.Length);
IntPtr hThread = IntPtr.Zero;
UInt32 threadId = 0;
IntPtr pinfo = IntPtr.Zero;
hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
WaitForSingleObject(hThread, 0xFFFFFFFF);
return;
```

Let's understand the objective and procedure of shellcode execution, then assign their respective API call from the above snippet.

Using the Windows API documentation and the context of the above snippet, begin analyzing the shellcode launcher, using questions 5 - 8 as a guide to work through the sample.

****

Answer the questions below:

What Win32 API call is used to obtain a pseudo handle of our current process in the keylogger sample?
Answer: GetCurrentProcess

What Win32 API call is used to set a hook on our current process in the keylogger sample?
Answer: SetWindowsHookEx

What Win32 API call is used to obtain a handle from the pseudo handle in the keylogger sample?
Answer: GetModuleHandle

What Win32 API call is used to unset the hook on our current process in the keylogger sample?
Answer: UnhookWindowsHookEx

What Win32 API call is used to allocate memory for the size of the shellcode in the shellcode launcher sample?
Answer: VirtualAlloc

What native method is used to write shellcode to an allocated section of memory in the shellcode launcher sample?
Answer: Marshal.copy

What Win32 API call is used to create a new execution thread in the shellcode launcher sample?
Answer: CreateThread

What Win32 API call is used to wait for the thread to exit in the shellcode launcher sample?
Answer: WaitForSingleObject

Conclusion

That is the end of the basics of the Windows API. The functionality and extensibility of the API have many other possibilities and applications that we did not mention in this room.

The Windows API can serve you for various use cases as we will continue to discuss and explore throughout the red team pathway and other rooms on TryHackMe.