# Introduction To The Windows API

## Introduction

The Windows API provides developers with a way for their applications to interact with the Windows operating system. For example, if the application needs to display something on the screen, modify a file or query the registry all of these actions can be done via the Windows API. The Windows API is very well documented by Microsoft and can be viewed here.

## Windows Data Types

The Windows API has many data types outside of the well-known ones (e.g. int, float). The data types are documented and can be viewed here.

Some of the common data types are listed below:

- `DWORD` - A 32-bit unsigned integer, on both 32-bit and 64-bit systems, used to represent values from 0 up to (2^32 - 1).

```
DWORD dwVariable = 42;
```

- `size_t` - Used to represent the size of an object. It's a 32-bit unsigned integer on 32-bit systems representing values from 0 up to (2^32 - 1). On the other hand, it's a 64-bit unsigned integer on 64-bit systems representing values from 0 up to (2^64 - 1).

```
SIZE_T sVariable = sizeof(int);
```

- `VOID` - Indicates the absence of a specific data type.

```
void* pVariable = NULL; // This is the same as PVOID
```

- `PVOID` - A 32-bit or 4-byte pointer of any data type on 32-bit systems. Alternatively, a 64-bit or 8-byte pointer of any data type on 64-bit systems.

```
PVOID pVariable = &SomeData;
```

- `HANDLE` - A value that specifies a particular object that the operating system is managing (e.g. file, process, thread).

```
HANDLE hFile = CreateFile(...);
```

- `HMODULE` - A handle to a module. This is the base address of the module in memory. An example of a MODULE can be a DLL or EXE file.

```
HMODULE hModule = GetModuleHandle(...);
```

- `LPCSTR/PCSTR` - A pointer to a constant null-terminated string of 8-bit Windows characters (ANSI). The "L" stands for "long" which is derived from the 16-bit Windows programming period, nowadays it doesn't affect the data type, but the naming convention still exists. The "C" stands for "constant" or read-only variable. Both these data types are equivalent to `const char*`.

```
LPCSTR  lpcString   = "Hello, world!";
PCSTR   pcString    = "Hello, world!";
```

- `LPSTR/PSTR` - The same as `LPCSTR` and `PCSTR`, the only difference is that `LPSTR` and `PSTR` do not point to a constant variable, and instead point to a readable and writable string. Both these data types are equivalent to `char*`.

```
LPSTR   lpString    = "Hello, world!";
PSTR    pString     = "Hello, world!";
```

- `LPCWSTR\PCWSTR` - A pointer to a constant null-terminated string of 16-bit Windows Unicode characters (Unicode). Both these data types are equivalent to `const wchar*`.

```
LPCWSTR     lpwcString   = L"Hello, world!";
PCWSTR      pcwString    = L"Hello, world!";
```

- `PWSTR\LPWSTR` - The same as `LPCWSTR` and `PCWSTR`, the only difference is that 'PWSTR' and 'LPWSTR' do not point to a constant variable, and instead point to a readable and writable string. Both these data types are equivalent to `wchar*`.

```
LPWSTR  lpwString   = L"Hello, world!";
PWSTR   pwString    = L"Hello, world!";
```

- `wchar_t` - The same as `wchar` which is used to represent wide characters.

```
wchar_t     wChar           = L'A';
wchar_t*    wcString        = L"Hello, world!";
```

- `ULONG_PTR` - Represents an unsigned integer that is the same size as a pointer on the specified architecture, meaning on 32-bit systems a `ULONG_PTR` will be 32 bits in size, and on 64-bit systems, it will be 64 bits in size. Throughout this course, `ULONG_PTR` will be used in the manipulation of arithmetic expressions containing pointers (e.g. PVOID). Before executing any arithmetic operation, a pointer will be subjected to type-casting to `ULONG_PTR`. This approach is used to avoid direct manipulation of pointers which can lead to compilation errors.

```
PVOID Pointer = malloc(100);
// Pointer = Pointer + 10; // not allowed
Pointer = (ULONG_PTR)Pointer + 10; // allowed
```

## Data Types Pointers

The Windows API allows a developer to declare a data type directly or a pointer to the data type. This is reflected in the data type names where the data types that start with "P" represent pointers to the actual data type while the ones that don't start with "P" represent the actual data type itself.

This will become useful later when working with Windows APIs that have parameters that are pointers to a data type. The examples below show how the "P" data type relates to its non-pointer equivalent.

- `PHANDLE` is the same as `HANDLE*`.
- `PSIZE_T` is the same as `SIZE_T*`.
- `PDWORD` is the same as `DWORD*`.

## ANSI & Unicode Functions

The majority of Windows API functions have two versions ending with either "A" or with "W". For example, there is CreateFileA and CreateFileW. The functions ending with "A" are meant to indicate "ANSI" whereas the functions ending with "W" represent Unicode or "Wide".

The main difference to keep in mind is that the ANSI functions will take in ANSI data types as parameters, where applicable, whereas the Unicode functions will take in Unicode data types. For example, the first parameter for `CreateFileA` is an `LPCSTR`, which is a pointer to a constant null-terminated string of **8-bit** Windows ANSI characters. On the other hand, the first parameter for `CreateFileW` is `LPCWSTR`, a pointer to a constant null-terminated string of **16-bit** Unicode characters.

Furthermore, the number of required bytes will differ depending on which version is used.

`char str1[] = "maldev";` // 7 bytes (maldev + null byte).

`wchar str2[] = L"maldev";` // 14 bytes, each character is 2 bytes (The null byte is also 2 bytes)

## In and Out Parameters

Windows APIs have in and out parameters. An `IN` parameter is a parameter that is passed into a function and is used for input. Whereas an `OUT` parameter is a parameter used to return a value back to the caller of the function. Output parameters are often passed in by reference through pointers.

For example, the code snippet below shows a function `HackTheWorld` which takes in an integer pointer and sets the value to `123`. This is considered an out parameter since the parameter is returning a value.

```
BOOL HackTheWorld(OUT int* num){

    // Setting the value of num to 123
    *num = 123;

    // Returning a boolean value
    return TRUE;
}

int main(){
    int a = 0;

    // 'HackTheWorld' will return true
    // 'a' will contain the value 123
    HackTheWorld(&a);
}
```

Keep in mind that the use of the `OUT` or `IN` keywords is meant to make it easier for developers to understand what the function expects and what it does with these parameters. However, it is worth mentioning that excluding these keywords does not affect whether the parameter is considered an output or input parameter.

## Windows API Example

Now that the fundamentals of the Windows API have been laid out, this section will go through the usage of the `CreateFileW` function.

## Find the API Reference

It's important to always reference the documentation if one is unsure about what the function does or what arguments it requires. Always read the description of the function and assess whether the function accomplishes the desired task. The `CreateFileW` documentation is available here.

## Analyze Return Type & Parameters

The next step would be to view the parameters of the function along with the return data type. The documentation states *If the function succeeds, the return value is an open handle to the specified file, device, named pipe, or mail slot* therefore `CreateFileW` returns a `HANDLE` data type to the specified item that's created.

Furthermore, notice that the function parameters are all `in` parameters. This means the function does not return any data from the parameters since they are all `in` parameters. Keep in mind that the keywords within the square brackets, such as `in`, `out`, and `optional`, are purely for developers' reference and do not have any actual impact.

```
HANDLE CreateFileW(
  [in]           LPCWSTR               lpFileName,
  [in]           DWORD                 dwDesiredAccess,
  [in]           DWORD                 dwShareMode,
  [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  [in]           DWORD                 dwCreationDisposition,
  [in]           DWORD                 dwFlagsAndAttributes,
  [in, optional] HANDLE                hTemplateFile
);
```

## Use The Function

The sample code below goes through an example usage of `CreateFileW`. It will create a text file with the name `maldev.txt` on the current user's Desktop.

```
// This is needed to store the handle to the file object
// the 'INVALID_HANDLE_VALUE' is just to intialize the variable
Handle hFile = INVALID_HANDLE_VALUE;

// The full path of the file to create.
// Double backslashes are required to escape the single backslash
character in C
LPCWSTR filePath = L"C:\\Users\\maldevacademy\\Desktop\\maldev.txt";
```

```
// Call CreateFileW with the file path
// The additional parameters are directly from the documentation
hFile = CreateFileW(filePath, GENERIC_ALL, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

// On failure CreateFileW returns INVALID_HANDLE_VALUE
// GetLastError() is another Windows API that retrieves the error
code of the previously executed WinAPI function
if (hFile == INVALID_HANDLE_VALUE){
    printf("[-] CreateFileW Api Function Failed With Error : %d\n",
GetLastError());
    return -1;
}
```

## Windows API Debugging Errors

When functions fail they often return a non-verbose error. For example, if `CreateFileW` fails it returns `INVALID_HANDLE_VALUE` which indicates that a file could not be created. To gain more insight as to why the file couldn't be created, the error code must be retrieved using the GetLastError function.

Once the code is retrieved, it needs to be looked up in Windows's System Error Codes List. Some common error codes are translated below:

- `5` - ERROR_ACCESS_DENIED
- `2` - ERROR_FILE_NOT_FOUND
- `87` - ERROR_INVALID_PARAMETER

## Windows Native API Debugging Errors

Recall from the *Windows Architecture* module, NTAPIs are mostly exported from `ntdll.dll`. Unlike Windows APIs, these functions cannot have their error code fetched via `GetLastError`. Instead, they return the error code directly which is represented by the `NTSTATUS` data type.

`NTSTATUS` is used to represent the status of a system call or function and is defined as a 32-bit unsigned integer value. A successful system call will return the value `STATUS_SUCCESS`, which is `0`. On the other hand, if the call failed it will return a non-zero value, to further investigate the cause of the problem, one must check Microsoft's documentation on NTSTATUS values.

The code snippet below shows how error checking for system calls is done.

```
NTSTATUS STATUS = NativeSyscallExample(...);
if (STATUS != STATUS_SUCCESS){
    // printing the error in unsigned integer hexadecimal format
    printf("[!] NativeSyscallExample Failed With Status : 0x%0.8X
\n", STATUS);
}

// NativeSyscallExample succeeded
```

## NT_SUCCESS Macro

Another way to check the return value of NTAPIs is through the `NT_SUCCESS` macro shown here. The macro returns `TRUE` if the function succeeded, and `FALSE` it fails.

```
#define NT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)
```

Below, is an example of using this macro

```
NTSTATUS STATUS = NativeSyscallExample(...);
if (!NT_SUCCESS(STATUS)){
    // printing the error in unsigned integer hexadecimal format
    printf("[!] NativeSyscallExample Failed With Status : 0x%0.8X
\n", STATUS);
}
```