COMPX508 – Malware Analysis

Week 7

Lecture 2: Windows API and DLLs

Vimal Kumar

Windows API

- Formerly called Win32 API
- A collection of functions and data structures for developing applications for the Windows Operating System
- These functions and data structures are usually defined in certain dlls
  - Help with creating and managing windows, handling user input (such as mouse clicks and keystrokes), managing memory, interacting with hardware devices, and accessing the file system, etc.

https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

Windows API

- These dlls are essentially the system-level libraries in Windows Operating Systems
- A Windows application will generally consist of a number of calls to functions in the Windows APIs
- Given an executable, even if we don’t know the source code we can find out the Windows API function calls
  - Knowing what kind of functions are being called from Windows API can tell us about what the application is doing.

https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

Common DLLS and their functions

- Kernel32.dll
  - Low-level operating system functions for process/thread management, memory management,I/O, etc.
  - Common functions used from kernel32
    - CreateProcess
    - VirtualAlloc
    - ReadFile
    - WriteFile
    - CreateMutex
    - etc.

https://github.com/7etsuo/windows-api-function-cheatsheets?tab=readme-ov-file#windows-api-function-cheatsheets

Common DLLS and their functions

- GDI32.dll
  - Functions that perform low-level drawing functions onto the displays
  - Drawing shapes such as ellipses, rectangles, etc.
  - Text Output
    - TextOutW
      - Writes a character string to the output device
- User32.dll
  - Windows management functions creating windows and their menus, message handling, timers, communications, etc.
  - These functions enable programs to create and manage GUIs
    - BeginPaint
    - EndPaint
    - CreateWindow
    - UpdateWindow
    - etc.

Common DLLS and their functions

- VCRUNTIME140.dll
  - DLL containing functions to support execution of programs written in C++
  - memcpy
  - memset

- UCRTbase.dll
  - Universal C run time library
  - Replaces MSCVRT.dll that was used before 2015
  - C string functions like strcpy, strcat etc.

- ADVAPI32.dll
  - Security functions and function to manipulate Windows registry. Also has functions to restart or shutdown the machine as well as functions to work with services.
  - RegOpenKeyExW
  - RegSetValueExW
  - CreateService
  - StartService
  - OpenEventLogW

Common DLLS and their functions

- WS2_32.dll
  - Provides TCP/IP networking functions
  - Implements the Windows socket API
    - socket
    - closesocket
    - accept
- WinInet.dll
  - Provides functions to perform HTTP communication
    - HttpSendRequest
    - InternetReadFile
    - InternetWriteFile
    - FtpGetFile

Common DLLS and their functions

- shell32.dll
  - Provides Windows shell functionality
    - ShellExecute
    - SHFileOperation
    - SHGetFolderPath
    - SHGetSetSettings

Example: Internet Operations

```cpp
bool CheckWebsiteExists(const std::wstring& url)
{
    HINTERNET hInternet = InternetOpen(L"Check Site Existence", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
    if (!hInternet) return false;

    HINTERNET hConnect = InternetOpenUrl(hInternet, url.c_str(), NULL, 0, INTERNET_FLAG_RELOAD, 0);
    if (!hConnect)
    {
        InternetCloseHandle(hInternet);
        return false;
    }

    DWORD statusCode = 0;
    DWORD length = sizeof(statusCode);
    HttpQueryInfo(hConnect, HTTP_QUERY_STATUS_CODE | HTTP_QUERY_FLAG_NUMBER, &statusCode, &length, NULL);

    InternetCloseHandle(hConnect);
    InternetCloseHandle(hInternet);

    return (statusCode == 200);
}
```

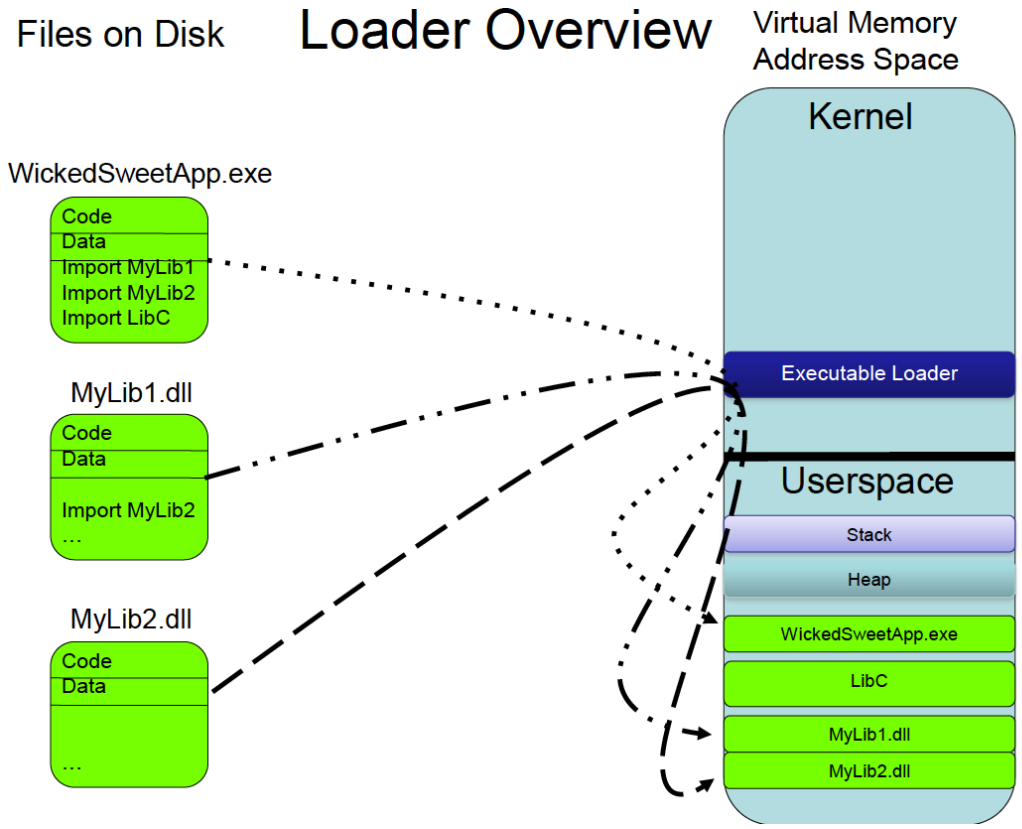
How are DLLs loaded?

- DLLs are dynamically linked libraries
  - Unlike static libraries the code of the functions that you call from DLLs is not compiled with the rest of the code.
  - Instead, you have symbolic links to the DLL functions
  - These can be seen as imports in the PE file

```
bool CheckWebsiteExists(const std::wstring& url)
{
    HINTERNET hInternet = InternetOpen(L"Check Site Existence", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
    if (!hInternet) return false;

    HINTERNET hConnect = InternetOpenUrl(hInternet, url.c_str(), NULL, 0, INTERNET_FLAG_RELOAD, 0);
    if (!hConnect)
    {
        InternetCloseHandle(hInternet);
        return false;
    }

    DWORD statusCode = 0;
    DWORD length = sizeof(statusCode);
    HttpQueryInfo(hConnect, HTTP_QUERY_STATUS_CODE | HTTP_QUERY_FLAG_NUMBER, &statusCode, &length, NULL);

    InternetCloseHandle(hConnect);
    InternetCloseHandle(hInternet);

    return (statusCode == 200);
}
```

How are DLLs loaded?

- Implicit Linking (Load-time Linking)
  - When the application is loaded in the memory, the operating system reads the import section and loads the required DLLs in the memory.
  - The application then makes calls to these functions as if they were part of the application code itself.



How are DLLs loaded?

- Explicit Linking (Run-time Linking)
  - Application calls the `LoadLibrary` function to load the DLL into memory when it is needed
  - After loading, the application uses `GetProcAddress` function to get a pointer to the function.
  - This function pointer is used to then call the desired function
    - This is also called dynamic resolution
  - Result
    - The import section doesn't show the DLLs and the imported functions in the PE file.

Explicit Linking Example

```
bool CheckWebsiteExists(const std::wstring& url)
{

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

    InternetOpenPtr pInternetOpen = (InternetOpenPtr)GetProcAddress(hWininet, "InternetOpenA");
    InternetOpenUrlPtr pInternetOpenUrl = (InternetOpenUrlPtr)GetProcAddress(hWininet, "InternetOpenUrlW");
    HttpQueryInfoPtr pHttpQueryInfo = (HttpQueryInfoPtr)GetProcAddress(hWininet, "HttpQueryInfoA");
    InternetCloseHandlePtr pInternetCloseHandle = (InternetCloseHandlePtr)GetProcAddress(hWininet, "InternetCloseHandle");


    HINTERNET hInternet = pInternetOpen("Check Site Existence", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
    if (!hInternet) return false;

    HINTERNET hConnect = pInternetOpenUrl(hInternet,url.c_str(), NULL, 0, INTERNET_FLAG_RELOAD, 0);
    if (!hConnect)
    {
```

The `ptr`s are defined with `typedef` earlier in the code

Static API imports can be obfuscated

- Dynamic Resolution
  - Instead of static imports, malware calls LoadLibrary + GetProcAddress to resolve API calls at runtime.
  - This thwarts static import table analysis.
- Manual Address Finding
  - Instead of using GetProcAddress to get the address of the function in the dll that is loaded, iterate over the process memory to find the address of the desired functions
  - Some anti-malware systems track the usage of GetProcAddress which can be defeated by this.
- API Hashing
  - Instead of storing API names in plaintext, malware stores hashes and computes them dynamically to get the address of the function in the memory.
  - Example: TrickBot
- Direct System Calls
  - System calls can be directly invoked bypassing the need for some of the API calls

API calls Monitoring

- Sequence of API calls can often reveal interesting behaviour
  - `CreateFileW` indicates a file being created.
  - `FindFirstFile → CreateFileW → WriteFile`
    - repeated across directories strongly suggests ransomware.
- While the existence of functions in the import may raise suspicion
  - More information can be found in the actual parameters used when the malware is executed.