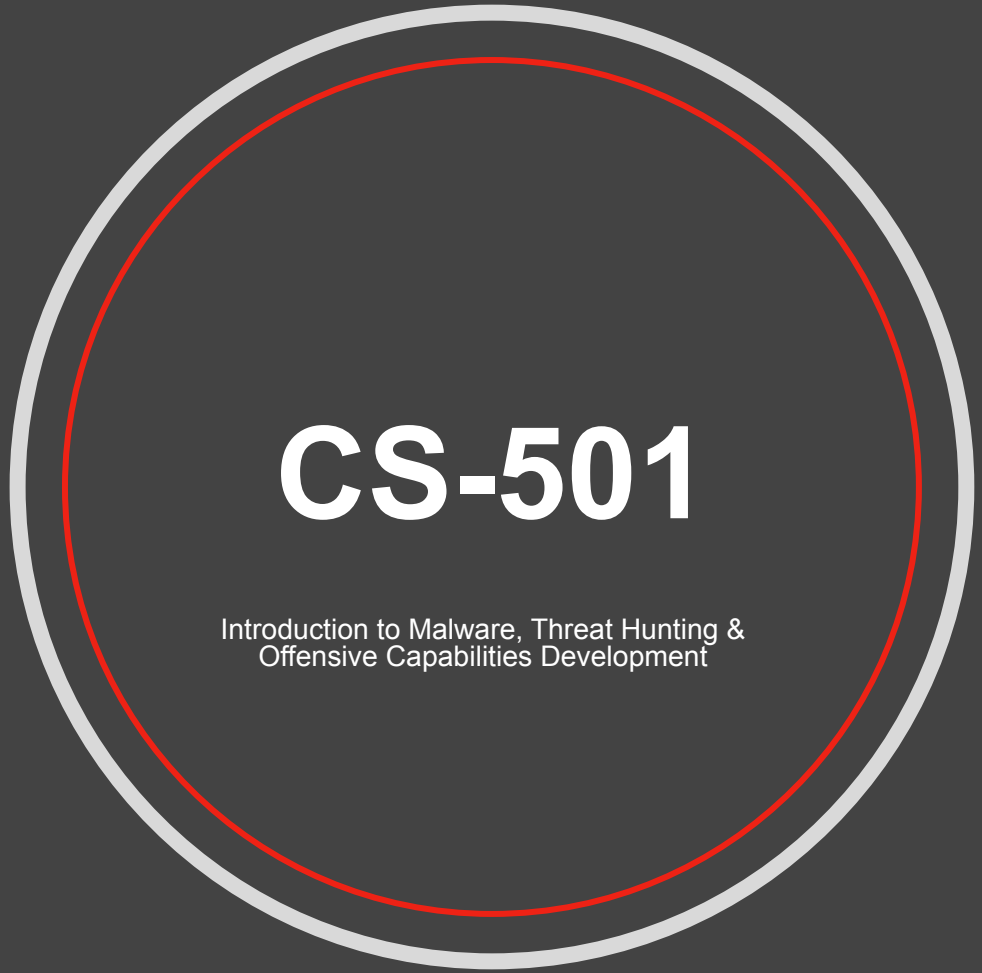


CS-501

Introduction to Malware, Threat Hunting &
Offensive Capabilities Development

Lecture 13: PEs, DLLs, Injection

PE File Format Basic Definitions and Concepts

- Portable Executables (PE) are an executable file format used by Windows NT
- file.exe, file.dll, file.obj all use the PE file format
- It is based on the Common Object File Format (COFF)
- The PE format is not architecture specific (hence “portable”)
    - Note this means the format can be used across multiple different architectures. The target architecture is still specified inside of the PE though
- Data is grouped together in blocks called sections, identified by headers

DLLs (Dynamically Linked Libraries): Goals for today

- How do we build a DLL?
- How do we execute a DLL?
- How do we export functions in a DLL?
- How do we call functions from a DLL?
- How do we force a process to load a DLL?

Building DLLs

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        printEntry();
        ::OutputDebugStringW(L"DLL_PROCESS_ATTACH");
        break;

    case DLL_THREAD_ATTACH:
        ::OutputDebugStringW(L"DLL_THREAD_ATTACH");
        break;

    case DLL_THREAD_DETACH:
        ::OutputDebugStringW(L"DLL_THREAD_DETACH");
        break;

    case DLL_PROCESS_DETACH:
        ::OutputDebugStringW(L"DLL_PROCESS_DETACH");
        break;
    }

    return TRUE;
}
```

```
PS C:\> g++ .\testdll.cpp -shared -o test.dll
```

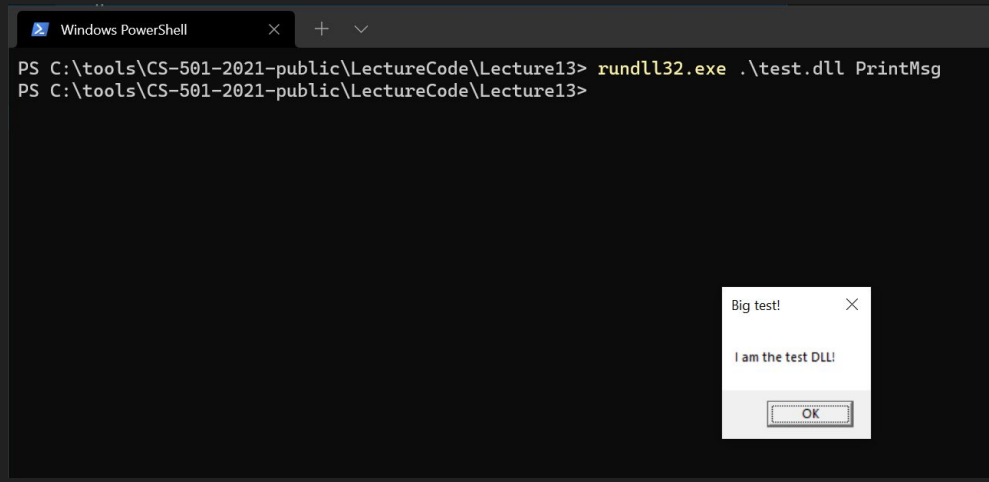
Running DLLs

Rundl32.exe (there is a 32-bit version and a 64-bit version)

Rundl32.exe can execute specific functions, or simply the DLLMain

If you give it the name of an unexported function, it will call ProcessAttach then error out

We can also write our own test runner by using LoadLibrary



Exporting Functions

```
// C++ will mangle names in exported functions. This is why we use edtern "C"
extern "C" __declspec(dllexport) int PrintMsg()
{
    ::MessageBoxW(NULL, L"I am an Exported function!", L"Big test!", MB_OK);
  return 0;
}
```

LoadLibrary Test Exe

- Loads the specified Library
- Sleeps a brief moment
- Unloads the library

PE File Format: How are PEs “loaded”

- We already talked about process creation at a high level, but have not yet discussed how Windows handles 2 main operations:

1) Resolving imports
2) Handling “Relative Values”
3) Handling “Absolute Values”

Quick Review

Static Linking

Dynamic Linking: Implicit

Dynamic Linking: Explicit

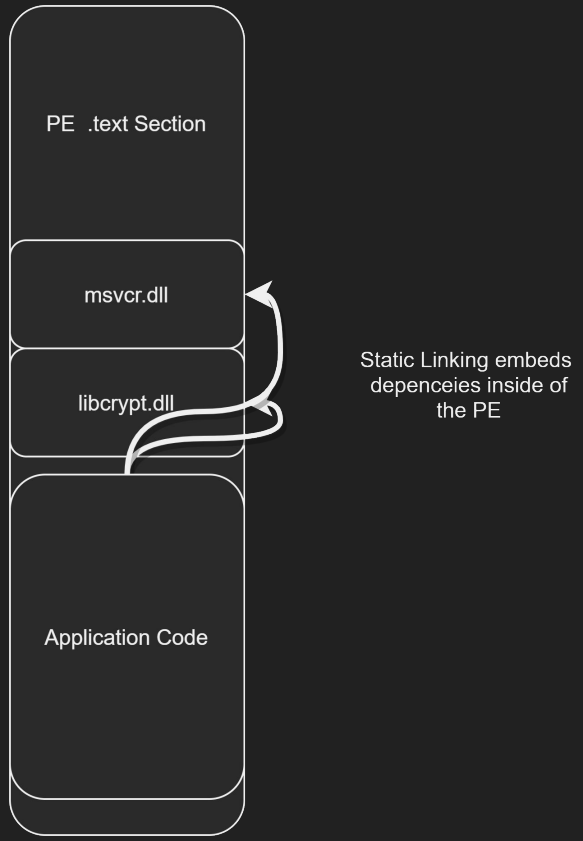
Static Linking

Statically linking against a library embeds the library inside of the PE

This is inefficient with space, but can solve many portability problems with code.

The main reason to statically link a library is if you are unsure if it will be available on the host machine

This bloats the side of the code, which may or may not be an issue.



Dynamic Linking: Implicit

Implicit Linking is declaring the required imports inside of the PE

The PE loader will then resolve these imports before execution is passed to the main thread

If any of the DLLs cannot be found, the PE loader aborts and the process exits with an error.

Most legitimate applications use implicit linking, and will ship with required DLLs packaged together with the application. These DLLs are then loaded at runtime.

Resolving Imports

When a DLL is loaded by a process, it is not guaranteed to be placed in the same place in memory, nor is it guaranteed to be placed in the same *relative* location

Resolving Imports

To handle variability in load location, the programer can can declare all imports in the *Import Address Table* (IAT)

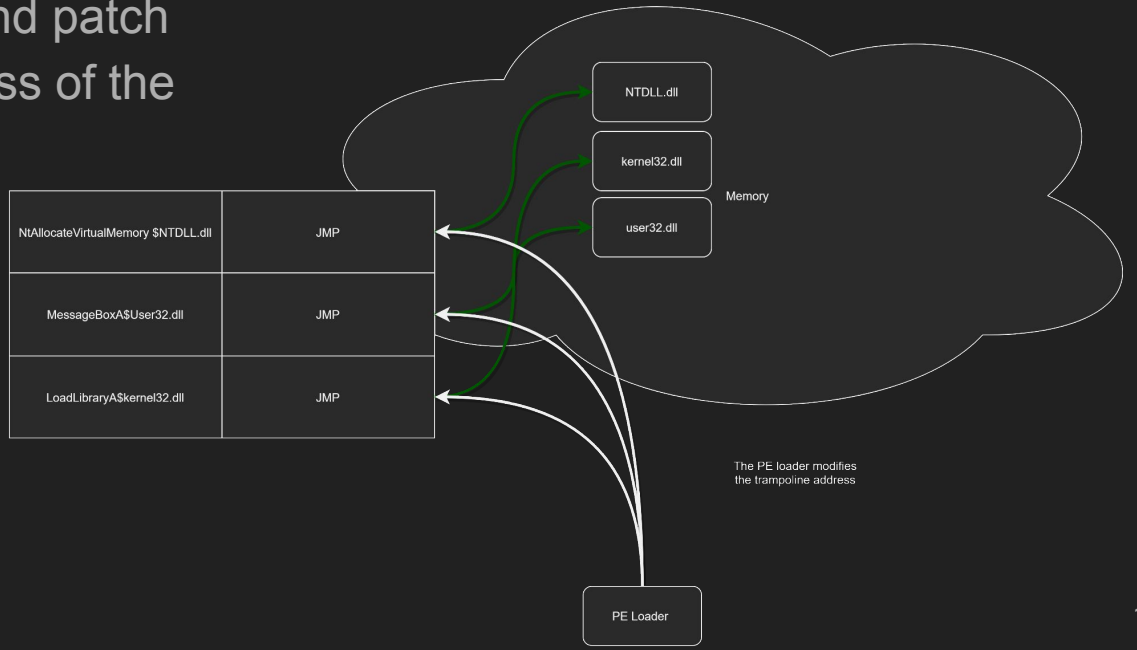
This shifts the work to the PE loader to resolve all the imports.

Resolving Imports

The IAT is effectively a collection of key value pairs, where the key is the name of an imported function, and the value is the address of a small function that jumps to the location of the real function.

Resolving Imports

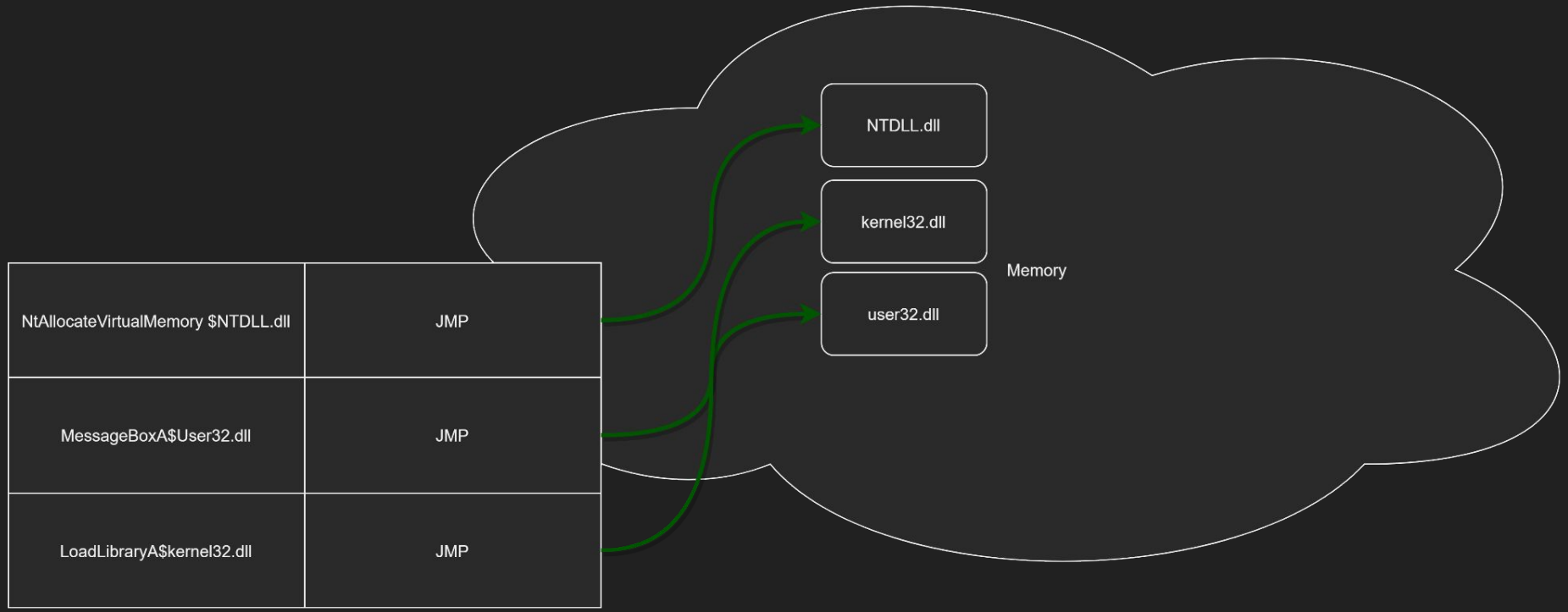
When resolving imports, the PE loader will load all required DLLs, identify the address of the loaded DLL, and patch the IAT with the correct address of the requested function



Resolving Imports

- The application code simply calls the function referenced in the IAT, which is simply a *trampoline* to the real code.
- I.e., it is a static location that application code can reference, which is simply an unconditional jump to the memory mapped location of the function.

Visual: IAT


NTDLL.dll
kernel32.dll
user32.dll
Memory
NtAllocateVirtualMemory $NTDLL.dll
JMP
MessageBoxA$User32.dll
JMP
LoadLibraryA$kernel32.dll
JMP

Dynamic Loading: Explicit

- The programer explicitly calls LoadLibrary, and has the proper function prototypes inside of their code.
- The legitimate reason to do this, is to allow the execution of the program to continue even if a DLL does not exist.
- If an import for an implicitly loaded DLL is missing, the program stops.
- The application using an explicitly loaded DLL can choose how to handle a missing DLL.

Dealing with Relative Addresses

A relative address is an address that is defined as a particular offset from a *base address* (the start of the PE's image in memory)

When the application is loaded into memory, same as the DLLs referenced in the IAT, its location might not align with the preferred location

In this case, the PE loader needs to modify the Base Location to handle relative locations.

This is especially important when exploitation mitigations such as ASLR are enabled, as the PE in this case should (probabilistically) never be loaded in the preferred address.

Applications that have only relative addresses are called Position independent. Generating Position Independent Code (PIC) is handled by the compiler, and can be enabled with -fPIC for GCC. This means you will see no unconditional jumps in application code, and instead will only see relative Jumps

Dealing With Absolute Addresses

An absolute address is a reference to a static location. Think JMP <addr>

If this address references a specific spot in memory, this could be tricky.

The base relocation table handles absolute addresses by creating a table of pointers to absolute addresses.

If the process is not loaded into its preferred address, the PE loader will modify all the absolute addresses to work with the new base address.

We generally don’t have to worry about this in our class, as all code will be PIC

DLL Injection

Forcing a remote process to load a library (DLL)

The DLL upon being loaded executes whatever code is contained in its attach process code.

There are a lot of reasons to do this, but general it is used for intercepting function calls and interacting with remote processes

Example one: Anti Malware

Many anti-malware executables have all the hallmarks of malware themselves!

An AV might inject a DLL into every process that a user spawns. It can then use this DLL to intercept commands to WinAPI functions to detect suspicious sequences of Function calls

Example 2: Cheat Engines

Running code in the context of a remote process gives unfettered access to resources used by the target process

This can be used to monitor application behavior, modify data, and inject functionality.

For example, the attach function in the injected DLL can modify values in memory associated with player’s health bar.

Example 3: Malware and Hooking

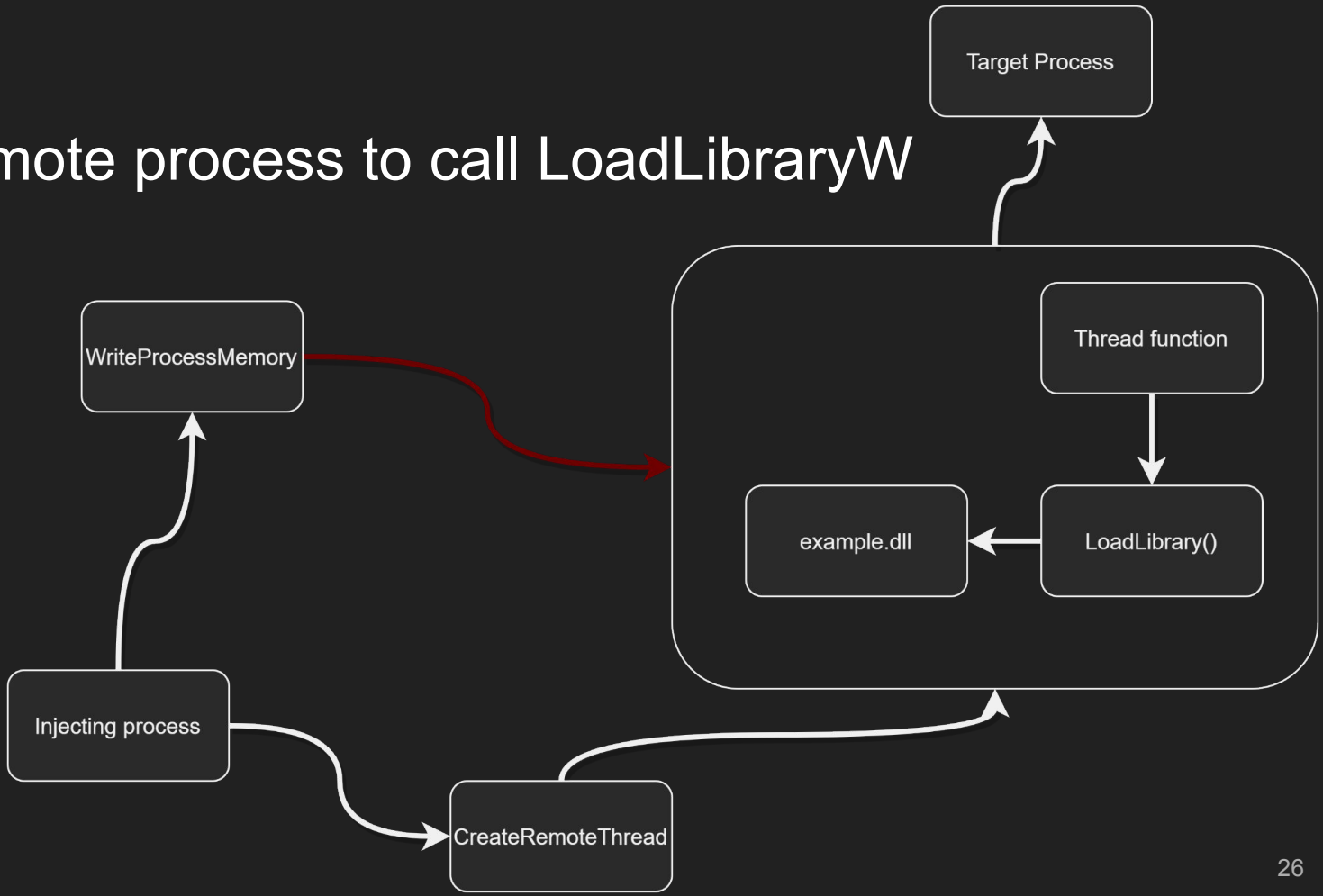
Imagine a user running Google Chrome logging into a website

If malware injects a malicious DLL into a chrome instance, it can read it’s process memory, modify functions, and steal data

For example, if the chrome function that sends an HTTP request is known, the DLL can patch references to that function with a small trampoline to code controlled by the malware

The malicious code can then inspect, and modify the arguments and forward them along.

How?
Force a remote process to call LoadLibraryW



Steps to Inject a DLL

1) Get the PID of the remote process we want to inject into
2) Get the FULL path to the DLL on disk
3) Open a handle to the remote process with the proper permissions
4) Create a LPTHREAD_START_ROUTINE function that loads the library from disk
5) Copy the contents of the function pointer’s arguments to the remote process with WriteProcessMemory
6) Call CreateRemoteThread

Creating the LPTHREAD_START_ROUTINE Function

- Open a handle to the target Process with OpenProcess
- Allocate a page of memory in the remote process with VirtualAllocEx
- Copy the required arguments (the full path to our DLL) into the remote buffer
- At this point, the thread arguments are in the remote process. We simply need to create a remote thread
- `(LPTHREAD_START_ROUTINE)::GetProcAddress(::GetModuleHandleW(L"kernel32"), "LoadLibraryW")`
- The above code creates a LPTHREAD_START_ROUTINE function pointer, which gets a handle to LoadLibraryW
- One argument for CreateRemoteThread is the Thread Arguments, which we set to the contents of the remote buffer

End Result

The remote process creates a new thread who's code consists of a single call to LoadLibraryW, and whatever code is present in `DLL_PROCESS_ATTACH`

We will talk more next week about how malware can use this to "hook" important functions

Opsec Consideration: The DLL has to be on disk!

Next time!

Hooking

Reflective DLL injection