COMPX508 – Malware Analysis

Week 10

Lecture 2: Code Injection - 2

Vimal Kumar

DLLs

```
void func1(param1, param2)
{
      …
      return value;
}

bool func2(param1, param2, ….., paramN)
{
      …
      return value;
}
DWORD func3(param1)
{
      …
      return value;
}
```

Exports

- Any function that is created needs to be exported

```
// File: HelloDll.h //

#ifndef HELLODLL_H
#define HELLODLL_H
      extern __declspec(dllexport) void func1(param1, param2) ;
      extern __declspec(dllexport) void func2(param1, param2, paramN) ;
      extern __declspec(dllexport) void func2(param1) ;
#endif
```

DllMain

- Optionally you can add a DllMain function in your library
- If defined this becomes the entry point of DLL and is executed when the library is loaded.
  - Any code in DllMain will be executed when the library is loaded
  - Malware authors often put code in DllMain and then get the library loaded, to execute the code.
    - But this is not necessary, they can also just write code in a function and then call the function
- Should be kept small as any execution here will mean the rest of the code waits for this execution to finish
  - You can create a thread so the execution takes place in a separate thread.

```
BOOL APIENTRY DllMain( HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved )
{
        ….Your code here….
        return TRUE;
 }
```

Testing DLLs with rundll32

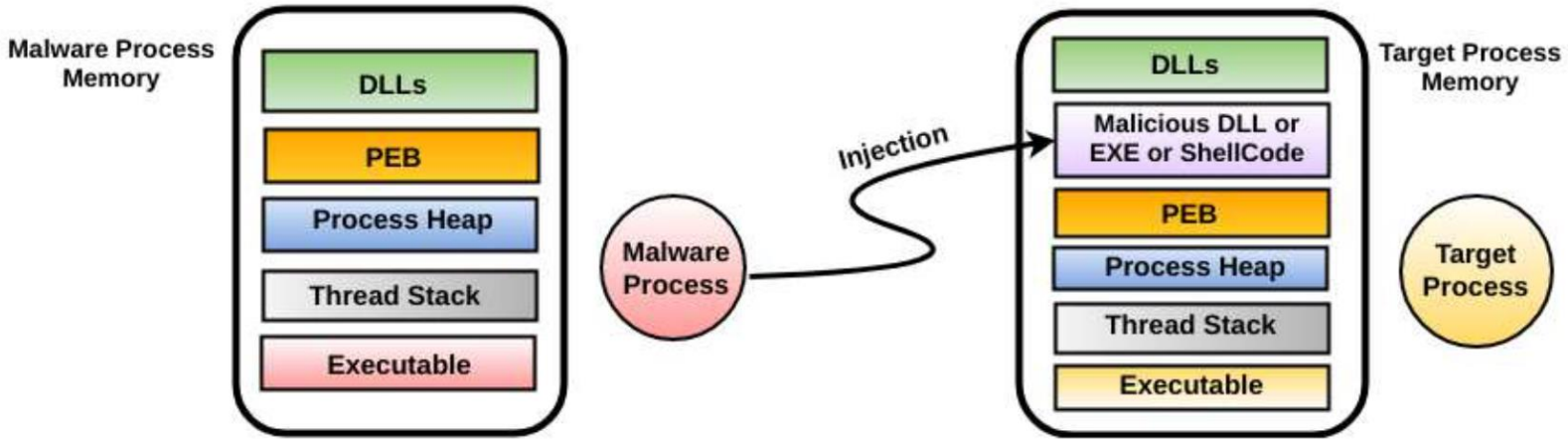
- A DLL file cannot be directly executed
- It needs to be loaded by an existing program that can call the functions in the library

- Rundll32.exe is a program on Windows that can help with dll testing
- It can load a given dll and call a given function

```
rundll32.exe HelloDll.dll, func1
```

Overview of Code Injection in a process

1. Locate the target process
2. Allocate memory in the target process
3. Write into the allocated memory
4. Execute the code



Locate the target process

- Malware would usually know the name of the process they are targeting but in order to programmatically access the process they need its PID.
- On Windows `CreateTool32HelpSnapshot()` function in the Win32 API is often used to get an object containing a list of all the current processes with their PIDs

```
std::vector<DWORD> find_pids_by_name(const std::wstring& target_name) {
    std::vector<DWORD> pids;

    HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (snap == INVALID_HANDLE_VALUE) return pids;

    PROCESSENTRY32W pe;
    pe.dwSize = sizeof(pe);

    std::wstring target_lc = to_lower(target_name);

    if (Process32FirstW(snap, &pe)) {
        do {
            std::wstring exe = pe.szExeFile;
            if (to_lower(exe) == target_lc) {
                pids.push_back(pe.th32ProcessID);
            }
        } while (Process32NextW(snap, &pe));
    }

    CloseHandle(snap);
    return pids;
}
```

Locate the target process

- Once the malware knows the PID, it will use the `OpenProcess()` function to get a handle for that process using the PID.

```
//Get a handle on the process
HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
if (!hProcess) {
    std::cerr << "OpenProcess failed. Error: " << GetLastError() << "\n";
    return 1;
}
```

Locate the target process: Detection

- Usage of `CreateTool32HelpSnapshot()` and `OpenProcess()` functions is an indication that a process is trying to get a handle on another process.

- You might also see `Process32First()` and `Process32Next()` functions that are used to iterate over the object returned by `CreateTool32HelpSnapshot()`

Allocating memory and writing to it, in the target process

- Malware would use memory allocation functions to allocate memory of a specific size, usually with read, write execute permissions.
- In the Windows API `VirtualAllocEx()` function is used for memory allocation within a remote process. The function requires the handle obtained from `OpenProcess()`

```
SIZE_T allocSize = (dll_to_inject.size() + 1);

LPVOID remoteMem = VirtualAllocEx(
    hProcess,
    NULL,
    allocSize,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);


std::cout << "Allocated " << allocSize
    << " bytes in process " << pid
    << " at address " << remoteMem << "\n";
```

Allocating memory and writing to it, in the target process

- In this next step the malicious code is actually written in the benign process's memory. The Win32 API function used for this is `WriteProcessMemory()`

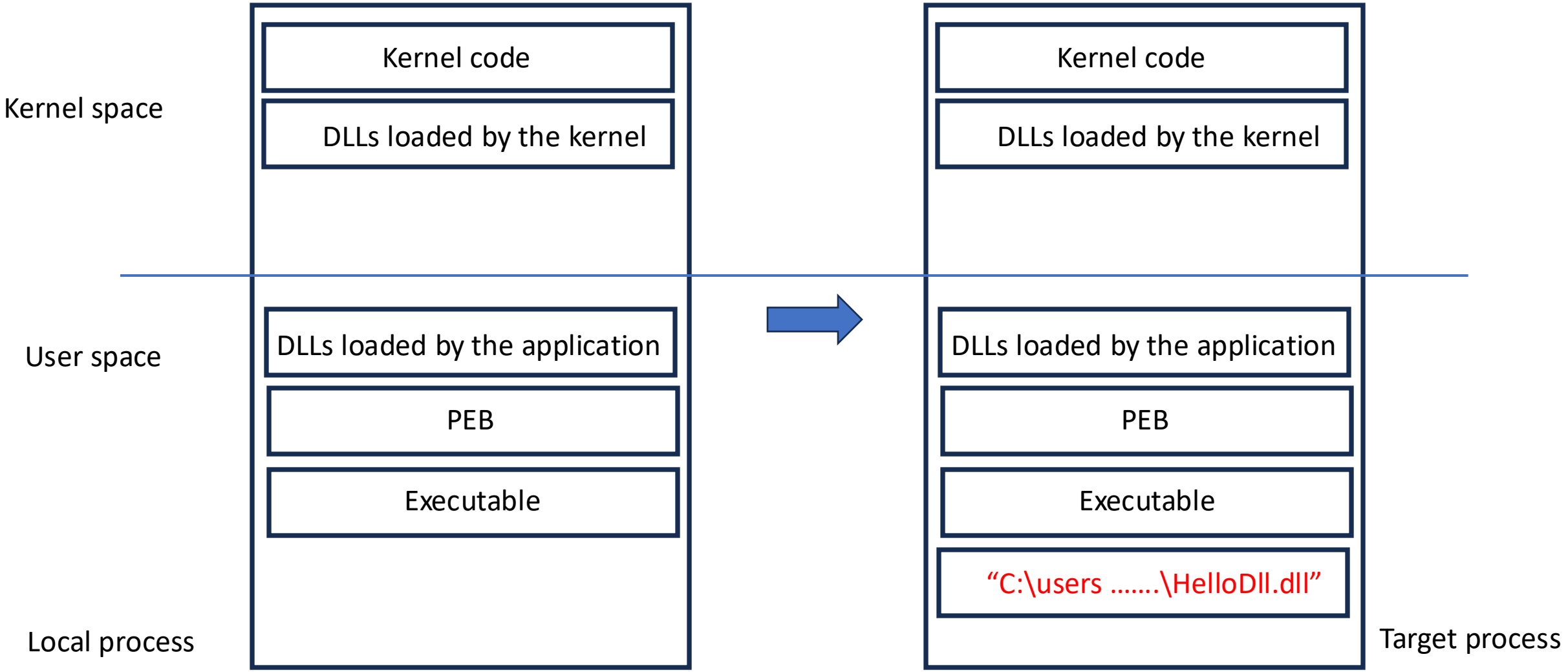
```
SIZE_T bytesWritten = 0;
BOOL ok = WriteProcessMemory(
    hProcess,                   // writing into current process for demo
    remoteMem,                  // base address in target to start writing
    dll_to_inject1,             // source buffer (in this process)
    allocSize,                  // how many bytes to copy
    &bytesWritten               // receives number of bytes actually written
);
```

Allocating memory and writing to it, in the target process: Detection

- Usage of `VirtualAllocEx()` and `WriteProcessMemory()` functions is an indication that a process is allocating memory in the virtual address space of another process.

- Additionally functions such as `LookupPrivilegeValue()` `AdjustTokenPrivileges()` and `OpenProcessToken()` are used to set the right privilege level for the malware process.

Process Memory



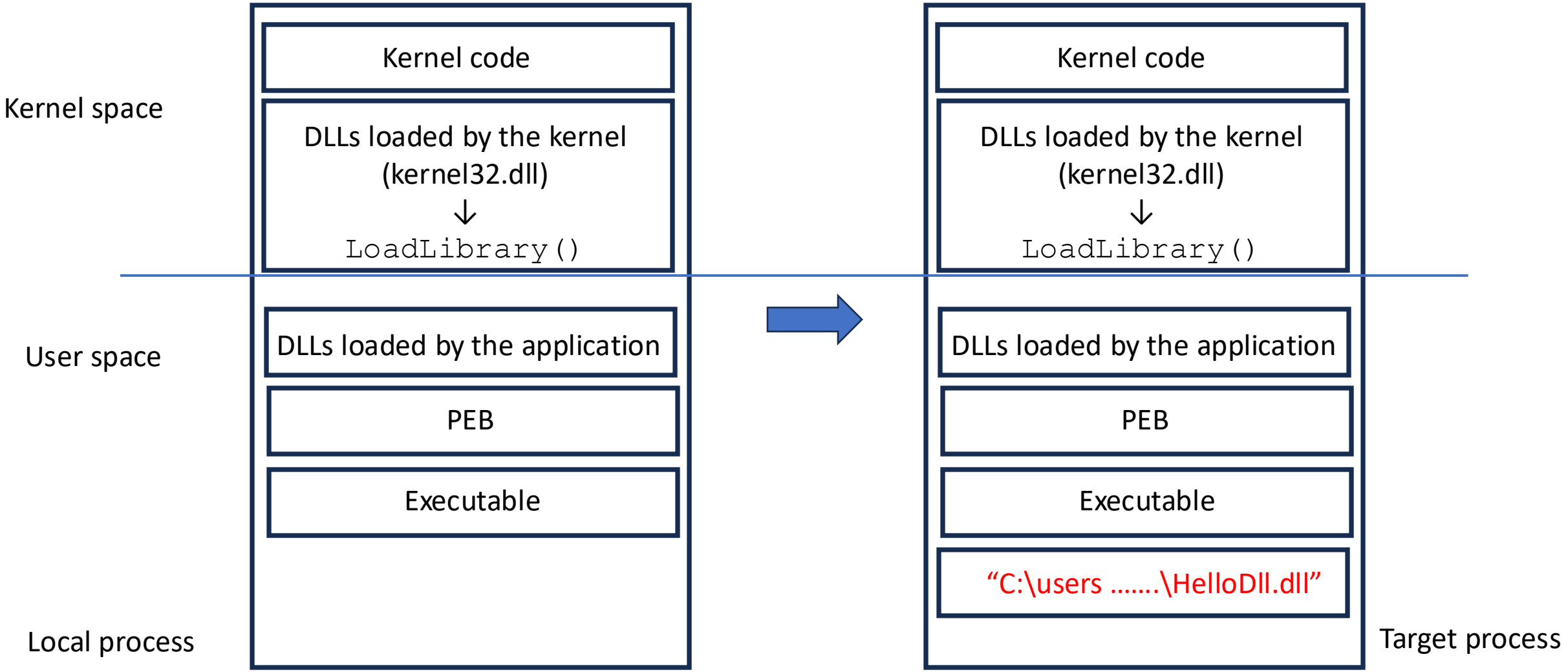
DLL Injection

- Once the dll path has been added in the target process' memory, the next task is to use this path and load the dll in the memory.

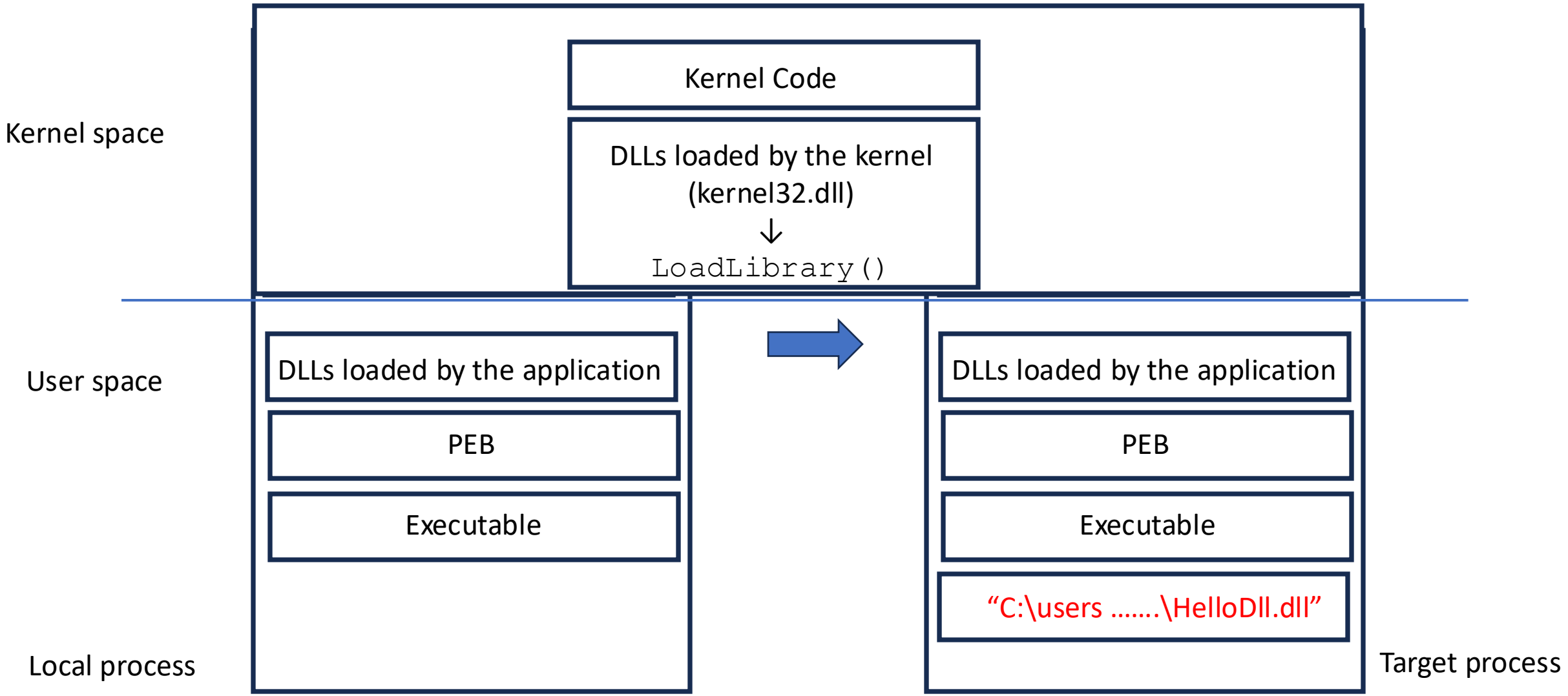
- Explicit Linking
  - `LoadLibrary` → loads the dll in the memory
  - `GetProcAddress` → gets the pointer to the desired function

- But if we use these functions directly then we will load the library and the function in the memory of the local process
  - We need to do this in the target process

Process Memory



Process Memory



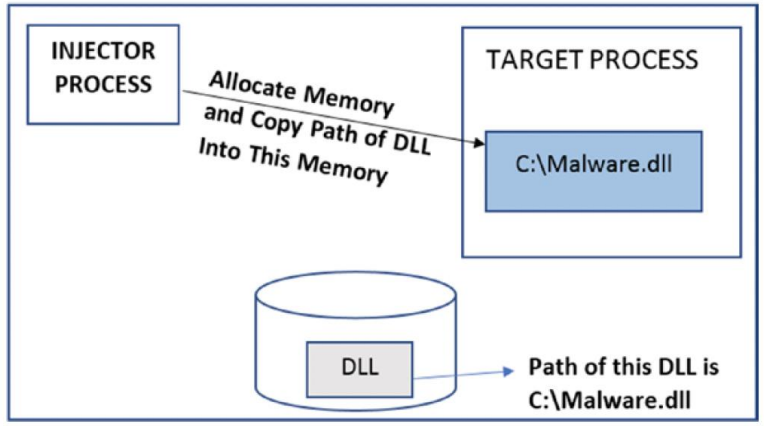
CreateRemoteThread

- Creates a thread that runs in the virtual address space of another process
- Call the `CreateRemoteThread` function to create a thread that
  - will run the `LoadLibrary` function
  - Pass the address to dllpath as a parameter to the `LoadLibrary` function

```
HANDLE hThread = CreateRemoteThread(
    hProcess,
    NULL,                   // Default security
    0,                      // Default stack size
    (LPTHREAD_START_ROUTINE)loadLibAddr, // Thread start: LoadLibraryA
    remoteString,           // Argument: pointer to DLL path in target
    0,                      // Run immediately
    NULL                    // Don't need the thread ID
);
```

DLL injection process

- The local process gets the handle of the target process.
- The local process first allocates memory in the target process, with the size of memory allocation, which is equal to the length of the path of the malicious DLL file.
  - For example, if the path of the DLL file on the disk is C:\Malware.dll, the size it allocates is 15 characters (including the trailing NULL character in the string).
- It then writes the path of the DLL file (i.e., C:\Malware.dll) to the memory allocated in the target process



DLL injection process

- The local process needs to somehow force the target process to invoke `LoadLibrary()` while passing to it the path of the DLL so that the DLL is loaded into its address space.
- Because kernel32.dll is always loaded at the same address in every Virtual Address Space, the address of the `LoadLibrary()` function can be found by looking for it in one's own Virtual Address Space.
- Once the local has the address of `LoadLibrary()` it can use `CreateRemoteThread()` to force the target process to run `LoadLibrary()` and use the path in the memory to load the malicious dll.

Methods to execute code in a target process

- `CreateRemoteThread`
- Using APC Injection
- `SetWindowsHookEx`
- Using Application Compatibility Shim
- Shellcode injection

Other Code Injection techniques

- Process Hollowing
- Hooking