# **Реализация Расширенного Раздельного Туннелирования для WireGuard VPN в Windows: Архитектура, Код и Развертывание на Уровне Драйвера**

## **I. Введение: Преодоление Ограничений WireGuard**

### **1.1. Проблема Раздельного Туннелирования на Уровне Приложений**

WireGuard зарекомендовал себя как высокопроизводительный, современный и безопасный VPN-протокол, отличающийся простотой и минималистичным подходом.1 Однако его встроенная функциональность раздельного туннелирования (split tunneling) ограничена параметром

AllowedIPs в конфигурационном файле. Этот параметр определяет, какой трафик (на основе IP-адресов или подсетей) должен проходить через VPN-туннель. Например, конфигурация AllowedIPs = 0.0.0.0/0 направляет весь IPv4-трафик через VPN, в то время как AllowedIPs = 192.168.1.0/24 ограничивает туннель только трафиком для указанной локальной сети.1

Такой подход, основанный на IP-адресации, представляет собой фундаментальное ограничение для современных VPN-приложений, ориентированных на пользователя. Основные недостатки заключаются в следующем:

* **Отсутствие привязки к приложениям:** Невозможно напрямую указать, что трафик от chrome.exe должен идти мимо туннеля, а трафик от outlook.exe — через него. Пользователи мыслят категориями приложений, а не IP-адресов.1
* **Динамические IP-адреса:** Многие облачные сервисы и веб-сайты (например, Netflix, Office 365) используют широкий и постоянно меняющийся пул IP-адресов, распределенных через сети доставки контента (CDN). Поддержание актуального списка IP-адресов для исключения таких сервисов становится практически невыполнимой задачей.1
* **Сложность управления:** Ручное управление списками IP-адресов и подсетей неудобно для конечного пользователя и подвержено ошибкам.1

Неспособность стандартного WireGuard гибко управлять трафиком на уровне отдельных приложений, а не только IP-адресов, является значительным препятствием для создания конкурентоспособного и удобного VPN-продукта. Пользователи ожидают, что VPN-клиент сможет легко адаптироваться к их рабочим процессам, позволяя определенным приложениям обходить туннель, в то время как другие используют его. Зависимость от статических или динамически изменяющихся IP-адресов для фильтрации не только усложняет управление для конечного пользователя, но и делает функцию раздельного туннелирования крайне ненадежной в условиях постоянно меняющейся сетевой инфраструктуры облачных сервисов. Таким образом, для создания современного VPN-продукта, соответствующего ожиданиям пользователей, необходимо реализовать механизм, который работает на уровне приложений, что требует выхода за рамки стандартных возможностей WireGuard и внедрения более глубокой интеграции с сетевым стеком операционной системы Windows.

### **1.2. Обзор Предлагаемой Архитектуры: Многокомпонентный Подход**

Прямое управление сетевым трафиком на уровне всей системы, включая его перехват и перенаправление, требует повышенных привилегий. Стандартное пользовательское приложение (GUI) не должно обладать такими привилегиями по соображениям безопасности. Запуск всего приложения с правами администратора является плохой практикой, так как это увеличивает поверхность атаки и создает риски для системы.1

Для решения этой задачи предлагается многокомпонентная архитектура, основанная на модели привилегированной вспомогательной службы (Privileged Helper Service). Этот подход разделяет логику приложения на компоненты с разными уровнями привилегий, что является ключевым принципом безопасного проектирования.1 Рекомендуемая архитектура состоит из трех основных компонентов:

* **Графический интерфейс на C# (GUI):** Компонент, с которым взаимодействует пользователь. Он отвечает за отображение списка запущенных процессов, прием выбора пользователя и отображение статуса VPN-соединения. Это приложение работает с обычными правами пользователя.1
* **Привилегированная служба Windows на C++:** "Мозг" всей системы, работающий как служба Windows с правами учетной записи LocalSystem. Эта служба выполняет все задачи, требующие повышенных привилегий: управление туннелем WireGuard, взаимодействие с сетевым стеком Windows и применение правил фильтрации.1
* **Драйвер-выноска (Callout Driver) WFP на C++:** Компонент режима ядра, необходимый для реализации наиболее надежного и гибкого метода перенаправления трафика.1

Взаимодействие между GUI с низкими привилегиями и службой с высокими привилегиями осуществляется через защищенный канал межпроцессного взаимодействия (IPC), такой как именованные каналы (Named Pipes).1

Выбор многокомпонентной архитектуры с четким разделением привилегий является критически важным для обеспечения безопасности, стабильности и устойчивости VPN-приложения в реальных условиях эксплуатации. Этот подход является прямым применением принципа наименьших привилегий, согласно которому каждый компонент системы должен иметь только те права, которые абсолютно необходимы для выполнения его функций. Если бы все приложение работало с административными правами, любая уязвимость в пользовательском интерфейсе могла бы быть использована для получения полного контроля над системой. Разделение функционала на изолированные, взаимодействующие компоненты значительно снижает потенциальный ущерб от возможных компрометаций, минимизируя поверхность атаки и повышая общую надежность системы.

## **II. Управление Туннелем WireGuard: Нативный API**

### **2.1. Интеграция с wireguard-nt (wireguard.dll)**

Для реализации динамического раздельного туннелирования приложению необходим надежный способ управления жизненным циклом и конфигурацией туннеля WireGuard. Хотя существует метод управления через утилиты командной строки (wireguard.exe и wg.exe), он имеет существенные недостатки для профессионального продукта. Любое изменение конфигурации (например, добавление IP-адреса в AllowedIPs) требует остановки службы, модификации текстового .conf файла на диске и перезапуска службы. Это приводит к кратковременному прерыванию VPN-соединения, что негативно сказывается на пользовательском опыте.1

Более современным и предпочтительным методом является использование нативного C API, предоставляемого проектом wireguard-nt в виде wireguard.dll.1 Этот подход позволяет управлять туннелем напрямую из кода, без вызова внешних процессов и перезапуска служб.1

**Загрузка API:** Ядро приложения на C++ должно динамически загружать wireguard.dll с помощью функции LoadLibraryEx() и получать адреса необходимых функций через GetProcAddress(). Заголовочный файл wireguard.h, поставляемый с wireguard-nt, содержит все необходимые определения типов данных и прототипы функций.1

**Ключевые функции API:**

* WireGuardCreateAdapter: Программно создает виртуальный сетевой адаптер WireGuard, указывая его имя (например, "MyVPN") и тип ("WireGuard").1
* WireGuardSetConfiguration: Применяет конфигурацию к созданному адаптеру. Важнейшее отличие от метода командной строки заключается в том, что конфигурация передается в виде C-структур (WIREGUARD\_INTERFACE, WIREGUARD\_PEER, WIREGUARD\_ALLOWED\_IP и др.), а не через .conf файл. Это позволяет изменять параметры, включая AllowedIPs, "на лету".1
* WireGuardSetAdapterState: Включает (WIREGUARD\_ADAPTER\_STATE\_UP) или выключает (WIREGUARD\_ADAPTER\_STATE\_DOWN) адаптер.1

Использование нативного API wireguard-nt является ключом к созданию гибкого и отзывчивого приложения, обеспечивающего бесшовный пользовательский опыт. Когда пользователь в графическом интерфейсе выбирает приложение для исключения из туннеля, и система определяет IP-адреса этого приложения (в рамках стратегии А) или устанавливает фильтр (в рамках стратегии Б), привилегированная служба может немедленно обновить конфигурацию WireGuard, вызвав WireGuardSetConfiguration с новым набором AllowedIPs или другими параметрами. Этот процесс происходит без разрыва соединения и каких-либо видимых для пользователя задержек, поскольку WireGuard интегрируется как библиотека, а не как внешний инструмент, требующий перезапуска. Такая возможность динамической и незаметной для пользователя адаптации поведения VPN является критически важной для профессионального продукта, который должен обеспечивать непрерывность сетевого подключения и высокий уровень удобства использования.

**Пример инициализации API и Динамического Обновления Конфигурации (Псевдокод/Структура):**

C++

// В C++ службе  
#**include** <windows.h>  
#**include** <vector>  
#**include** <string>  
#**include** "wireguard.h" // Из wireguard-nt/api/wireguard.h  
  
// Объявление указателей на функции WireGuardNT API  
static WIREGUARD\_CREATE\_ADAPTER\_FUNC \*WireGuardCreateAdapter;  
static WIREGUARD\_SET\_CONFIGURATION\_FUNC \*WireGuardSetConfiguration;  
static WIREGUARD\_SET\_ADAPTER\_STATE\_FUNC \*WireGuardSetAdapterState;  
//... другие функции, необходимые для взаимодействия с WireGuardNT  
  
// Функция для инициализации API WireGuardNT путем динамической загрузки wireguard.dll  
HMODULE InitializeWireGuardNTApi() {  
 HMODULE hWireGuardDll = LoadLibraryExW(L"wireguard.dll", NULL, LOAD\_LIBRARY\_SEARCH\_APPLICATION\_DIR | LOAD\_LIBRARY\_SEARCH\_SYSTEM32);  
 if (!hWireGuardDll) {  
 // Обработка ошибки: не удалось загрузить библиотеку wireguard.dll  
 // Например, логирование ошибки: GetLastError()  
 return NULL;  
 }  
  
 // Инициализация указателей на функции WireGuardNT API  
 // Макрос X упрощает инициализацию и проверку наличия функций  
#**define** X(Name) ((\*(FARPROC \*)&Name = GetProcAddress(hWireGuardDll, #Name)) == NULL)  
 if (X(WireGuardCreateAdapter) |  
| X(WireGuardSetConfiguration) |  
| X(WireGuardSetAdapterState))  
#**undef** X  
 {  
 // Обработка ошибки: не удалось загрузить одну или несколько необходимых функций  
 // Логирование ошибки и освобождение библиотеки  
 DWORD lastError = GetLastError();  
 FreeLibrary(hWireGuardDll);  
 SetLastError(lastError); // Сохраняем код последней ошибки  
 return NULL;  
 }  
 return hWireGuardDll;  
}  
  
// Пример структуры для хранения полной конфигурации WireGuard  
// Включает интерфейс, пиры и их AllowedIPs  
struct WireGuardFullConfig {  
 WIREGUARD\_INTERFACE Interface;  
 // Можно добавить массив пиров, если их несколько  
 // WIREGUARD\_PEER Peers;   
 // WIREGUARD\_ALLOWED\_IP AllowedIPs; // Для динамического управления  
};  
  
// Пример функции для обновления AllowedIPs для конкретного пира  
// В реальной системе это может быть сложнее, если пиров много или AllowedIPs управляются комплексно  
void UpdatePeerAllowedIPs(WIREGUARD\_ADAPTER\_HANDLE adapterHandle, const std::vector<WIREGUARD\_ALLOWED\_IP>& newAllowedIPs) {  
 // Получение текущей конфигурации адаптера  
 // Для этого потребуется WireGuardGetConfiguration, которая не показана в примере,  
 // но является частью WireGuardNT API.  
 // Предположим, что у нас есть способ получить текущие параметры интерфейса и пиров.  
  
 // Создаем временную структуру для новой конфигурации  
 WireGuardFullConfig tempConfig;  
 // Заполняем tempConfig текущими параметрами интерфейса и пиров,  
 // например, из сохраненного состояния или путем вызова WireGuardGetConfiguration.  
 // tempConfig.Interface =...;  
 // tempConfig.DemoServer =...; // Если у нас один пир  
  
 // Динамически выделяем память для AllowedIPs, которые будут переданы в WireGuardSetConfiguration  
 // Это критически важно, так как AllowedIPs должны быть частью единого блока памяти,  
 // передаваемого в WireGuardSetConfiguration.  
 size\_t totalConfigSize = sizeof(WIREGUARD\_INTERFACE) +  
 sizeof(WIREGUARD\_PEER) + // Для одного пира  
 newAllowedIPs.size() \* sizeof(WIREGUARD\_ALLOWED\_IP);  
  
 // Выделяем память для всего блока конфигурации  
 // Включая интерфейс, пиры и AllowedIPs  
 // В реальной реализации может потребоваться более сложная структура  
 // или повторное использование памяти, если конфигурация часто меняется.  
 void\* configBuffer = malloc(totalConfigSize);  
 if (!configBuffer) {  
 // Обработка ошибки выделения памяти  
 return;  
 }  
  
 WIREGUARD\_INTERFACE\* pInterface = (WIREGUARD\_INTERFACE\*)configBuffer;  
 WIREGUARD\_PEER\* pPeer = (WIREGUARD\_PEER\*)(pInterface + 1); // Пир следует за интерфейсом  
 WIREGUARD\_ALLOWED\_IP\* pAllowedIPs = (WIREGUARD\_ALLOWED\_IP\*)(pPeer + 1); // AllowedIPs следуют за пиром  
  
 // Заполняем интерфейс (пример)  
 pInterface->Flags = WIREGUARD\_INTERFACE\_HAS\_PRIVATE\_KEY;  
 // pInterface->PrivateKey =... (ваш приватный ключ)  
 pInterface->PeersCount = 1; // У нас один пир  
  
 // Заполняем пир (пример)  
 pPeer->Flags = WIREGUARD\_PEER\_HAS\_PUBLIC\_KEY | WIREGUARD\_PEER\_HAS\_ENDPOINT;  
 // pPeer->PublicKey =... (публичный ключ пира)  
 // pPeer->Endpoint =... (адрес конечной точки пира)  
 pPeer->AllowedIPs = pAllowedIPs; // Указываем на массив AllowedIPs  
 pPeer->AllowedIPsCount = newAllowedIPs.size();  
  
 // Копируем новые AllowedIPs  
 if (!newAllowedIPs.empty()) {  
 memcpy(pAllowedIPs, newAllowedIPs.data(), newAllowedIPs.size() \* sizeof(WIREGUARD\_ALLOWED\_IP));  
 }  
  
 // Применяем новую конфигурацию  
 if (!WireGuardSetConfiguration(adapterHandle, pInterface, (UINT32)totalConfigSize)) {  
 // Обработка ошибки установки конфигурации  
 // Например, логирование GetLastError()  
 }  
  
 free(configBuffer); // Освобождаем выделенную память  
}  
  
// В основной логике службы:  
// HMODULE wgDll = InitializeWireGuardNTApi();  
// if (wgDll) {  
// WIREGUARD\_ADAPTER\_HANDLE adapter = WireGuardCreateAdapter(L"MyVPNAdapter", L"WireGuard", NULL);  
// if (adapter) {  
// // Пример: Добавление 1.1.1.1/32 и 8.8.8.8/32 в AllowedIPs  
// std::vector<WIREGUARD\_ALLOWED\_IP> initialAllowedIPs;  
// WIREGUARD\_ALLOWED\_IP ip1 = {.AddressFamily = AF\_INET,.Cidr = 32 };  
// // Заполнить ip1.Address.v4 соответствующим IP-адресом  
// // Например, ip1.Address.v4.sin\_addr.S\_un.S\_addr = RtlUlongByteSwap(inet\_addr("1.1.1.1"));  
// initialAllowedIPs.push\_back(ip1);  
//  
// WIREGUARD\_ALLOWED\_IP ip2 = {.AddressFamily = AF\_INET,.Cidr = 32 };  
// // Заполнить ip2.Address.v4 соответствующим IP-адресом  
// // Например, ip2.Address.v4.sin\_addr.S\_un.S\_addr = RtlUlongByteSwap(inet\_addr("8.8.8.8"));  
// initialAllowedIPs.push\_back(ip2);  
//  
// UpdatePeerAllowedIPs(adapter, initialAllowedIPs);  
//  
// WireGuardSetAdapterState(adapter, WIREGUARD\_ADAPTER\_STATE\_UP);  
// //...  
// // Позже, когда пользователь исключает приложение, обновляем AllowedIPs  
// // std::vector<WIREGUARD\_ALLOWED\_IP> updatedAllowedIPs = { /\*... \*/ };  
// // UpdatePeerAllowedIPs(adapter, updatedAllowedIPs);  
//  
// WireGuardCloseAdapter(adapter);  
// }  
// FreeLibrary(wgDll);  
// }

## **III. Пользовательский Интерфейс (C#): Взаимодействие с Системой**

### **3.1. Получение Списка Запущенных Приложений**

Графический интерфейс пользователя (GUI), реализованный на C#, является точкой входа для пользователя. Он должен предоставлять интуитивно понятный способ выбора приложений для раздельного туннелирования. Основная задача GUI — предоставить пользователю список запущенных в системе приложений, из которого он сможет выбрать те, трафик которых должен идти мимо VPN.1

Для получения списка всех активных процессов используется метод System.Diagnostics.Process.GetProcesses().1 Этот метод возвращает массив объектов

Process, каждый из которых представляет один запущенный процесс. Для каждого объекта Process необходимо получить информацию, полезную для пользователя:

* Имя процесса: process.ProcessName.1
* Путь к исполняемому файлу: process.MainModule.FileName. Этот путь является ключевой информацией, которая будет передаваться в C++ службу для настройки фильтрации.1
* Иконка приложения: System.Drawing.Icon.ExtractAssociatedIcon(process.MainModule.FileName). Иконка значительно улучшает юзабилити, позволяя пользователю быстрее идентифицировать приложения.1

Критическим аспектом является совместимость 32- и 64-битных процессов. При попытке доступа к свойству process.MainModule из 32-битного приложения для 64-битного процесса возникает исключение System.ComponentModel.Win32Exception с сообщением "A 32 bit process cannot access modules of a 64 bit process".1 Это распространенная и серьезная проблема, которая может полностью заблокировать работу функции. Для ее решения необходимо обеспечить, чтобы C#-приложение работало как 64-битный процесс на 64-битной ОС. Это достигается правильной настройкой проекта в Visual Studio: платформа решения (Platform target) должна быть установлена в "Any CPU", а флажок "Предпочитать 32-разрядную версию" (Prefer 32-bit) должен быть снят.1 Эта конфигурация гарантирует, что на 64-битной системе приложение будет запущено как 64-битный процесс, что даст ему необходимые права для запроса информации у других 64-битных процессов. Кроме того, код должен корректно обрабатывать исключения при доступе к

MainModule, так как для некоторых системных процессов (например, "System" или "System Idle Process") это свойство недоступно в принципе.1

Надежная работа функции перечисления процессов является фундаментальной для удобства использования продукта. Если графический интерфейс не может корректно отобразить список запущенных приложений или постоянно выдает ошибки из-за проблем с доступом к информации о процессах, это напрямую подрывает функциональность раздельного туннелирования и негативно сказывается на пользовательском опыте. Правильная настройка проекта C# для обеспечения совместимости с 64-битными процессами и тщательная обработка исключений при доступе к MainModule являются критически важными шагами, которые гарантируют стабильную и функциональную работу пользовательского интерфейса, что в свою очередь способствует повышению удовлетворенности пользователя и общему восприятию качества продукта.

**Код: System.Diagnostics.Process.GetProcesses и MainModule.FileName (C#)**

C#

using System;  
using System.Diagnostics;  
using System.ComponentModel;  
using System.Drawing; // Для System.Drawing.Icon  
using System.Collections.Generic;  
using System.IO; // Для Path.GetFileName  
  
public class ProcessInfo  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public string Path { get; set; }  
 public Icon Icon { get; set; }  
 public string DisplayName => string.IsNullOrEmpty(Name)? Path.GetFileName(Path) : Name;  
}  
  
public class ApplicationLister  
{  
 public static List<ProcessInfo> GetRunningApplications()  
 {  
 List<ProcessInfo> appList = new List<ProcessInfo>();  
 Process processes = Process.GetProcesses(); // [1], 3.1; [5]  
  
 foreach (Process p in processes)  
 {  
 try  
 {  
 // Пропускаем процессы без главного модуля или с пустым путем  
 // Некоторые системные процессы не имеют MainModule  
 if (p.MainModule == null |  
| string.IsNullOrWhiteSpace(p.MainModule.FileName))  
 {  
 continue;  
 }  
  
 string appPath = p.MainModule.FileName; // [1], 3.1; [8, 9]  
 string appName = p.ProcessName;  
  
 Icon appIcon = null;  
 try  
 {  
 appIcon = Icon.ExtractAssociatedIcon(appPath); // [1], 3.1  
 }  
 catch (Exception iconEx)  
 {  
 // Игнорируем ошибки извлечения иконки, если иконка недоступна  
 // Это может произойти, если файл заблокирован или не является исполняемым  
 Console.WriteLine($"Warning: Could not extract icon for {appName} ({appPath}): {iconEx.Message}");  
 }  
  
 appList.Add(new ProcessInfo  
 {  
 Id = p.Id,  
 Name = appName,  
 Path = appPath,  
 Icon = appIcon  
 });  
 }  
 catch (Win32Exception ex)  
 {  
 // [1], 3.1: "A 32 bit process cannot access modules of a 64 bit process"  
 // Это исключение может возникнуть, если текущее C# приложение 32-битное, а процесс - 64-битный.  
 // Или для системных процессов, к которым нет доступа (например, "System", "System Idle Process").  
 Console.WriteLine($"Access Denied or other Win32 error for process {p.ProcessName} (ID: {p.Id}): {ex.Message}");  
 }  
 catch (InvalidOperationException ex)  
 {  
 // Процесс мог завершиться между GetProcesses() и доступом к MainModule  
 Console.WriteLine($"Process {p.ProcessName} (ID: {p.Id}) has exited: {ex.Message}");  
 }  
 catch (Exception ex)  
 {  
 // Общая обработка других неожиданных исключений  
 Console.WriteLine($"An unexpected error occurred for process {p.ProcessName} (ID: {p.Id}): {ex.Message}");  
 }  
 }  
 return appList;  
 }  
}

### **3.2. Межпроцессное Взаимодействие (IPC) с Привилегированной Службой**

Поскольку GUI работает с правами обычного пользователя, а служба — с правами LocalSystem, для их взаимодействия необходим надежный и безопасный механизм межпроцессного взаимодействия (IPC).1 Анализ различных вариантов IPC показывает, что WCF (Windows Communication Foundation) является мощным, но избыточно сложным для локального взаимодействия между двумя процессами на одной машине. Файлы, отображаемые в память (Memory-Mapped Files), очень быстры, но требуют ручной реализации примитивов синхронизации (мьютексы, семафоры) для избежания состояния гонки.1

**Именованные каналы (Named Pipes)** являются рекомендуемым выбором. Это стандартный, хорошо документированный и безопасный механизм для клиент-серверного IPC в Windows. Он идеально подходит для модели "GUI-клиент" и "служба-сервер".1

**Схема реализации на Named Pipes:**

* **C++ служба (сервер):** В отдельном потоке создает сервер именованного канала с помощью функции CreateNamedPipe. Затем ожидает подключения клиента (GUI) с помощью ConnectNamedPipe. Для каждого успешного подключения создается экземпляр для чтения (ReadFile) и записи (WriteFile).1
* **C# GUI (клиент):** Использует класс System.IO.Pipes.NamedPipeClientStream для подключения к каналу, созданному службой. После подключения клиент может отправлять сериализованные команды (например, строки "ADD\_APP:C:\path\to\app.exe" или "REMOVE\_APP:C:\path\to\app.exe").1
* **Двунаправленная связь:** Канал используется не только для команд от GUI к службе, но и для обратной связи. Служба может отправлять в GUI обновления статуса туннеля, статистику трафика и подтверждения выполнения команд.1

Выбор именованных каналов для межпроцессного взаимодействия между графическим интерфейсом и привилегированной службой является оптимальным с точки зрения безопасности и стабильности. Поскольку служба работает с высокими привилегиями, любой ненадежный канал связи может стать вектором для атак, направленных на повышение привилегий или нарушение работы системы. Именованные каналы, будучи встроенным и хорошо изученным механизмом Windows, обеспечивают контролируемый и изолированный канал связи, который минимизирует риски несанкционированного доступа или манипуляций. Это не просто техническое удобство, а критически важный компонент архитектуры безопасности, который способствует общей надежности и защищенности всего VPN-приложения.

**Код: Именованные Каналы (Named Pipes) - Клиент и Сервер (Псевдокод/Структура)**

C#

// В C# GUI (клиент)  
using System;  
using System.IO;  
using System.IO.Pipes;  
using System.Text;  
using System.Threading.Tasks;  
  
public class VpnServiceClient  
{  
 private NamedPipeClientStream pipeClient;  
 private StreamReader reader;  
 private StreamWriter writer;  
 private string pipeName;  
  
 public VpnServiceClient(string pipeName = "MyVpnControlPipe")  
 {  
 this.pipeName = pipeName;  
 // "." указывает на локальный компьютер, PipeDirection.InOut для двунаправленной связи  
 pipeClient = new NamedPipeClientStream(".", this.pipeName, PipeDirection.InOut); // [1], 3.2; [11]  
 }  
  
 public async Task<bool> ConnectAsync()  
 {  
 try  
 {  
 // Попытка подключения к серверу с таймаутом  
 await pipeClient.ConnectAsync(5000); // Таймаут 5 секунд  
 reader = new StreamReader(pipeClient);  
 writer = new StreamWriter(pipeClient) { AutoFlush = true }; // Автоматическая очистка буфера  
 Console.WriteLine("Connected to VPN service.");  
 return true;  
 }  
 catch (TimeoutException)  
 {  
 Console.WriteLine("Failed to connect to VPN service: Timeout.");  
 return false;  
 }  
 catch (Exception ex)  
 {  
 Console.WriteLine($"Failed to connect to VPN service: {ex.Message}");  
 return false;  
 }  
 }  
  
 public void Disconnect()  
 {  
 if (pipeClient!= null && pipeClient.IsConnected)  
 {  
 pipeClient.Close();  
 pipeClient.Dispose();  
 Console.WriteLine("Disconnected from VPN service.");  
 }  
 }  
  
 public async Task<string> SendCommandAsync(string command)  
 {  
 if (pipeClient == null ||!pipeClient.IsConnected)  
 {  
 return "Error: Not connected to service.";  
 }  
 try  
 {  
 await writer.WriteLineAsync(command); // Отправка команды  
 return await reader.ReadLineAsync(); // Ожидание и чтение ответа от службы  
 }  
 catch (IOException ex)  
 {  
 // Канал мог быть разорван  
 Console.WriteLine($"IPC communication error: {ex.Message}");  
 Disconnect(); // Попытка отключиться и очистить ресурсы  
 return "Error: IPC communication failed.";  
 }  
 }  
}  
  
// В C++ службе (сервер)  
#include <windows.h>  
#include <string>  
#include <thread>  
#include <vector>  
#include <iostream> // Для вывода в консоль службы (для отладки)  
  
// Имя именованного канала. Должно совпадать с именем в клиенте.  
#**define** PIPE\_NAME L"\\\\.\\pipe\\MyVpnControlPipe" // [1], 3.2  
  
// Функция для обработки одного клиентского подключения  
void HandleClientConnection(HANDLE hPipe) {  
 char buffer;  
 DWORD bytesRead;  
 BOOL success;  
  
 // Цикл чтения команд от клиента  
 while (true) {  
 // Чтение данных из канала  
 success = ReadFile(hPipe, buffer, sizeof(buffer) - 1, &bytesRead, NULL); // [1], 3.2  
 if (!success |  
| bytesRead == 0) {  
 // Клиент отключился или произошла ошибка чтения  
 if (GetLastError() == ERROR\_BROKEN\_PIPE) {  
 std::cout << "Client disconnected." << std::endl;  
 } else {  
 std::cerr << "Error reading from pipe: " << GetLastError() << std::endl;  
 }  
 break;  
 }  
 buffer = '\0'; // Null-terminate the string  
  
 std::string command(buffer);  
 std::cout << "Received command: " << command << std::endl;  
  
 // Здесь должна быть логика обработки команды и формирования ответа  
 // Например: "ADD\_APP:C:\path\to\app.exe"  
 // Служба вызывает WFP API для добавления/удаления фильтров  
 std::string response = "ACK: " + command; // Пример ответа  
  
 DWORD bytesWritten;  
 // Запись ответа в канал  
 success = WriteFile(hPipe, response.c\_str(), (DWORD)response.length(), &bytesWritten, NULL); // [1], 3.2  
 if (!success) {  
 std::cerr << "Error writing to pipe: " << GetLastError() << std::endl;  
 break;  
 }  
 }  
 // Отключение и закрытие канала после завершения взаимодействия  
 DisconnectNamedPipe(hPipe);  
 CloseHandle(hPipe);  
}  
  
// Функция для запуска сервера именованных каналов  
void StartPipeServer() {  
 HANDLE hPipe;  
 while (true) {  
 // Создание именованного канала  
 hPipe = CreateNamedPipe(  
 PIPE\_NAME,  
 PIPE\_ACCESS\_DUPLEX, // Двунаправленный доступ (чтение/запись)  
 PIPE\_TYPE\_MESSAGE | // Сообщения, а не байтовый поток  
 PIPE\_READMODE\_MESSAGE | // Чтение в режиме сообщений  
 PIPE\_WAIT, // Блокирующий режим  
 PIPE\_UNLIMITED\_INSTANCES, // Максимальное количество экземпляров (неограниченно)  
 1024, // Размер выходного буфера  
 1024, // Размер входного буфера  
 0, // Таймаут по умолчанию  
 NULL); // Атрибуты безопасности по умолчанию // [1], 3.2  
  
 if (hPipe == INVALID\_HANDLE\_VALUE) {  
 std::cerr << "CreateNamedPipe failed: " << GetLastError() << std::endl;  
 // Можно добавить задержку перед повторной попыткой  
 Sleep(5000);  
 continue;  
 }  
  
 // Ожидание подключения клиента  
 // ConnectNamedPipe блокируется до подключения клиента или ошибки  
 if (ConnectNamedPipe(hPipe, NULL) |  
| GetLastError() == ERROR\_PIPE\_CONNECTED) { // [1], 3.2  
 std::cout << "Client connected. Spawning handler thread." << std::endl;  
 // Создание нового потока для обработки подключения клиента  
 std::thread clientThread(HandleClientConnection, hPipe);  
 clientThread.detach(); // Отсоединяем поток, чтобы он работал независимо  
 } else {  
 // Ошибка подключения, закрываем хендл и повторяем  
 CloseHandle(hPipe);  
 }  
 }  
}  
  
// Пример вызова в main или ServiceMain:  
// std::thread pipeServerThread(StartPipeServer);  
// pipeServerThread.detach();

### **3.3. Маршалинг Данных (P/Invoke): Передача Списков Строк**

Для некоторых задач, например, для получения от C++ ядра списка уже отфильтрованных приложений, может потребоваться прямой вызов функций из C++ DLL из управляемого кода C#. Это делается с помощью механизма Platform Invoke (P/Invoke).1

**Определение P/Invoke:** В C# коде объявляются статические extern методы, помеченные атрибутом ``, где core.dll — имя вашей C++ библиотеки.1

**Сложность маршалинга (преобразования типов):** Передача простых типов (например, int, bool) тривиальна. Однако передача сложных структур данных, таких как список строк (std::vector<std::string> из C++ в C# List<string>), требует особого подхода.1 Нельзя напрямую передать

std::vector<std::string> из C++ в C# List<string>.

**Решение заключается в использовании C-совместимого формата данных:**

1. **На стороне C++:** Экспортируемая функция должна выделять память под массив указателей на C-строки (char\*\* или wchar\_t\*\*) и возвращать этот массив вместе с его размером (например, через out-параметр). Также должна быть отдельная функция для освобождения этой памяти, выделенной в C++, чтобы избежать утечек памяти.1  
   C++  
   // Пример экспортируемой С++ функции (в core.dll)  
   // Предполагаем, что строки в UTF-8 или ANSI  
   extern "C" \_\_declspec(dllexport) void GetFilteredAppPaths(char\*\*\* appPaths, int\* count);  
   extern "C" \_\_declspec(dllexport) void FreeFilteredAppPathsArray(char\*\* appPaths, int count);  
     
   // Если используются широкие символы (UTF-16), что предпочтительнее для Windows API  
   // extern "C" \_\_declspec(dllexport) void GetFilteredAppPathsW(wchar\_t\*\*\* appPaths, int\* count);  
   // extern "C" \_\_declspec(dllexport) void FreeFilteredAppPathsArrayW(wchar\_t\*\* appPaths, int count);
2. **На стороне C#:** В объявлении DllImport используется out IntPtr или ref IntPtr и атрибут [MarshalAs]. C# код вызывает первую функцию, получает указатель на массив указателей, затем использует Marshal.Copy для копирования указателей, а Marshal.PtrToStringAnsi (или Marshal.PtrToStringUni для wchar\_t\*) для преобразования каждого указателя в C# строку. После использования данных обязательно вызывается вторая функция (FreeFilteredAppPathsArray) для освобождения памяти, выделенной в C++, чтобы избежать утечек памяти.1

Корректный маршалинг данных между C# и C++, особенно управление памятью, выделенной в нативном коде, является фундаментальным для предотвращения утечек памяти и обеспечения надежности всей системы. Без явного освобождения памяти, выделенной в C++ и переданной в C#, управляемая среда.NET не сможет автоматически управлять этой памятью, что приведет к постепенному накоплению неиспользуемых ресурсов. В долгосрочно работающем приложении, таком как привилегированная служба, это может привести к деградации производительности и, в конечном итоге, к сбоям системы. Поэтому тщательное следование протоколу выделения и освобождения памяти является обязательным для стабильности и производительности.

**Код: Маршалинг C++ char\*\* в C# string (Псевдокод/Структура)**

C#

// В C# GUI  
using System;  
using System.Runtime.InteropServices;  
using System.Collections.Generic;  
  
public class CoreInterop  
{  
 // Объявление P/Invoke функций для взаимодействия с C++ DLL  
 // [1], 3.3; [12]  
 public static extern void GetFilteredAppPaths(out IntPtr appPaths, out int count);  
  
 // [1], 3.3  
 public static extern void FreeFilteredAppPathsArray(IntPtr appPaths, int count);  
  
 // Если C++ использует wchar\_t\* (предпочтительно для Windows)  
 //  
 // public static extern void GetFilteredAppPathsW(out IntPtr appPaths, out int count);  
 //  
 // public static extern void FreeFilteredAppPathsArrayW(IntPtr appPaths, int count);  
  
 public static List<string> GetFilteredAppsList()  
 {  
 IntPtr pathsPtr = IntPtr.Zero;  
 int appCount = 0;  
 List<string> apps = new List<string>();  
  
 try  
 {  
 // Вызов C++ функции для получения списка путей  
 GetFilteredAppPaths(out pathsPtr, out appCount); // [1], 3.3  
  
 if (appCount > 0 && pathsPtr!= IntPtr.Zero)  
 {  
 // Создание массива IntPtr для хранения указателей на строки  
 IntPtr ptrs = new IntPtr[appCount];  
 // Копирование указателей из нативной памяти в управляемый массив  
 Marshal.Copy(pathsPtr, ptrs, 0, appCount); // [1], 3.3  
  
 // Преобразование каждого указателя на строку в C# string  
 for (int i = 0; i < appCount; i++)  
 {  
 // Использование PtrToStringAnsi для char\* или PtrToStringUni для wchar\_t\*  
 apps.Add(Marshal.PtrToStringAnsi(ptrs[i])); // [1], 3.3  
 }  
 }  
 }  
 catch (Exception ex)  
 {  
 Console.WriteLine($"Error getting filtered apps: {ex.Message}");  
 // Дополнительная обработка ошибок, например, логирование  
 }  
 finally  
 {  
 // Обязательно: освобождение памяти, выделенной в C++  
 if (pathsPtr!= IntPtr.Zero)  
 {  
 FreeFilteredAppPathsArray(pathsPtr, appCount); // [1], 3.3  
 }  
 }  
 return apps;  
 }  
}

## **IV. Привилегированная Служба (C++): Управление WFP Фильтрами**

### **4.1. Основы Windows Filtering Platform (WFP)**

Ядро на C++, работающее в контексте привилегированной службы, является центром всей логики раздельного туннелирования. Оно использует Windows Filtering Platform (WFP) для перехвата и управления сетевым трафиком. WFP — это современный и мощный набор API и системных служб, предоставляющий платформу для создания приложений сетевой фильтрации. WFP пришла на смену устаревшим технологиям, таким как TDI-фильтры, NDIS-фильтры и Winsock LSP. Встроенный в Windows "Брандмауэр Защитника Windows в режиме повышенной безопасности" (WFAS) сам реализован поверх WFP.1

**Ключевые компоненты WFP:**

* **Base Filtering Engine (BFE):** Служба пользовательского режима (bfe.dll в процессе svchost.exe), которая управляет всей системой фильтрации. C++ ядро будет взаимодействовать с BFE для добавления, удаления и изменения правил.1
* **Уровни фильтрации (Filtering Layers):** Предопределенные точки в сетевом стеке, где могут быть применены фильтры. Для нашей задачи ключевым является уровень FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4 (и \_V6 для IPv6), который перехватывает попытки установки исходящих TCP/UDP соединений.1
* **Фильтры (Filters):** Правила, состоящие из условий и действия. Условие может быть, например, "путь к приложению", "удаленный порт" или "протокол". Действие определяет, что делать с трафиком, который соответствует условиям: разрешить (Permit), заблокировать (Block) или передать для дальнейшей обработки драйверу-выноске (Callout).1
* **Драйверы-выноски (Callout Drivers):** Драйверы режима ядра, которые могут выполнять глубокий анализ или модификацию сетевого трафика. Они необходимы для реализации наиболее продвинутой логики перенаправления.1

### **4.2. Создание Пользовательского Подуровня WFP для Высокого Приоритета**

При работе с WFP нельзя просто добавить разрешающий фильтр и ожидать, что он сработает. Другие приложения, включая сам WireGuard, также используют WFP. Например, функция "kill-switch" в официальном клиенте WireGuard реализована с помощью WFP-фильтров, которые имеют действие Block для всего трафика, который не исходит от его собственного виртуального интерфейса.1 Если наш разрешающий фильтр для

chrome.exe будет иметь более низкий приоритет, чем блокирующий фильтр WireGuard, трафик все равно будет заблокирован.

Порядок применения фильтров определяется механизмом арбитража WFP. Сначала трафик проходит через подуровни (sub-layers) в порядке убывания их веса (приоритета). Внутри каждого подуровня фильтры также применяются в порядке убывания их веса. Решение Block обычно переопределяет решение Permit из подуровня с более низким приоритетом.1

Следовательно, чтобы гарантированно управлять трафиком нужных приложений, необходимо создать свой собственный подуровень с высоким приоритетом. Это гарантирует, что правила для раздельного туннелирования будут рассмотрены до правил брандмауэра Windows или блокирующих правил WireGuard.1 Значение

0x8000 является хорошей отправной точкой для веса подуровня (subLayer.weight), но для переопределения других фильтров может потребоваться более высокое значение, например, MAXUINT16.1

Все изменения WFP должны выполняться внутри транзакции (FwpmTransactionBegin0, FwpmTransactionCommit0), что гарантирует их атомарное применение. Если какой-либо шаг не удастся, вся транзакция откатывается, оставляя систему в согласованном состоянии.1

Управление приоритетом WFP-фильтров через создание собственного высокоприоритетного подуровня является фундаментальным для обеспечения предсказуемого и корректного поведения раздельного туннелирования. Без этого механизма фильтры, установленные VPN-клиентом, могут быть легко переопределены или проигнорированы другими системными компонентами или сторонним программным обеспечением (например, антивирусами или другими VPN-клиентами), что приведет к нежелательному поведению или полному отказу функции. Создание собственного подуровня гарантирует, что правила, определяющие, какой трафик должен обходить VPN, будут обработаны с наивысшим приоритетом, предотвращая конфликты и обеспечивая надежную работу раздельного туннелирования.

### **4.3. Реализация Фильтрации на Основе Идентификатора Приложения (App ID)**

После создания инфраструктуры (поставщика и подуровня) можно добавлять фильтры для конкретных приложений, выбранных пользователем. Для каждого исключаемого приложения вызывается функция FwpmFilterAdd0.1

**Ключевые свойства структуры FWPM\_FILTER0:**

* layerKey: Указывает на уровень фильтрации, например, FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4 и FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V6.1
* subLayerKey: Указывает GUID нашего кастомного подуровня.1
* action.type: Для стратегии перенаправления (Стратегия Б) это будет FWP\_ACTION\_CALLOUT\_TERMINATING, что указывает WFP передать управление нашему драйверу-выноске.1
* filterCondition: Массив структур FWPM\_FILTER\_CONDITION0, описывающих условия срабатывания фильтра. Основным условием является идентификатор приложения (FWPM\_CONDITION\_APP\_ID). Этот тип условия позволяет фильтровать трафик на основе полного пути к исполняемому файлу. Однако значением этого условия должен быть не просто текстовый путь, а специальный бинарный блоб (blob).1

**Получение App ID:** Для преобразования пути к файлу (например, C:\Program Files\Google\Chrome\Application\chrome.exe) в необходимый формат блоба традиционно используется API-функция FwpmGetAppIdFromFileName0.1

Важный Аспект: Обход FwpmGetAppIdFromFileName0 для Совместимости с EDR/AV

Функция FwpmGetAppIdFromFileName0 для своей работы внутренне вызывает CreateFileW для доступа к исполняемому файлу. Системы защиты конечных точек (EDR) и антивирусы (AV) часто блокируют доступ к своим собственным процессам, чтобы предотвратить их отключение или изменение. Если пользователь захочет исключить из VPN трафик своего антивируса, вызов FwpmGetAppIdFromFileName0 для его процесса может завершиться ошибкой "Доступ запрещен".1

Решение этой проблемы продемонстрировано в проекте EDRSilencer. Оно заключается в ручном формировании App ID блоба на основе пути к файлу, без фактического вызова CreateFileW.1 Этот подход позволяет обойти защитные механизмы EDR/AV и является обязательным для реализации в надежном продукте, который должен работать в разнообразных системных окружениях.

Реализация ручного формирования App ID блоба является обязательным требованием для создания надежного VPN-продукта, который должен корректно функционировать в разнообразных системных окружениях, сосуществуя с EDR/AV решениями. Зависимость от FwpmGetAppIdFromFileName0 может привести к непредсказуемым сбоям или невозможности применения правил для критически важных приложений, таких как антивирусы, что напрямую влияет на стабильность и надежность VPN-клиента. Разработка собственной логики для создания App ID блоба, которая не вызывает файловые операции, подверженные блокировке EDR/AV, позволяет приложению беспрепятственно интегрироваться в защищенные среды, обеспечивая полную функциональность раздельного туннелирования без конфликтов с существующими защитными механизмами.

Весь процесс добавления или удаления фильтра для приложения должен быть обернут в WFP-транзакцию для обеспечения целостности конфигурации.1

**Код: Добавление WFP Фильтра с FWPM\_CONDITION\_APP\_ID (Псевдокод/Структура)**

C++

// В C++ службе  
#**include** <fwpmu.h> // Для WFP API  
#**include** <guiddef.h> // Для GUID  
#**include** <vector>  
#**include** <string>  
#**include** <iostream> // Для логирования  
  
// Определения GUID (должны быть уникальными для вашего приложения)  
// Эти GUID должны быть сгенерированы один раз и использоваться постоянно.  
// Пример:  
// {12345678-1234-1234-1234-1234567890AB} - YOUR\_PROVIDER\_GUID  
// {AABBCCDD-AABB-AABB-AABB-AABBCCDDEEFF} - YOUR\_SUBLAYER\_GUID  
// {11223344-1122-1122-1122-112233445566} - YOUR\_CALLOUT\_GUID\_V4  
// {99887766-9988-9988-9988-998877665544} - YOUR\_CALLOUT\_GUID\_V6  
  
// Заглушки GUID для примера. В реальном коде используйте уникальные GUID.  
const GUID YOUR\_PROVIDER\_GUID = { /\*... \*/ };  
const GUID YOUR\_SUBLAYER\_GUID = { /\*... \*/ };  
const GUID YOUR\_CALLOUT\_GUID\_V4 = { /\*... \*/ };  
const GUID YOUR\_CALLOUT\_GUID\_V6 = { /\*... \*/ };  
  
  
// Функция для ручного формирования App ID блоба (на основе принципов EDRSilencer)  
// Это упрощенный псевдокод для демонстрации концепции.  
// Реальная реализация требует тщательной обработки пути, конвертации в NT-путь,  
// добавления заголовков и т.д. См. EDRSilencer для примера.  
// [1], 4.3; [19, 21, 22, 23]  
FWP\_BYTE\_BLOB\* CreateAppIdBlobFromPath(const WCHAR\* filePath) {  
 // В реальной реализации здесь будет сложная логика, которая формирует  
 // бинарный блоб App ID из пути к файлу без вызова CreateFileW.  
 // Это включает:  
 // 1. Преобразование пути из пользовательского формата (например, "C:\Program Files\App\app.exe")  
 // в формат NT-пути (например, "\Device\HarddiskVolume1\Program Files\App\app.exe").  
 // 2. Добавление префикса "app\_id://".  
 // 3. Кодирование полученной строки в UTF-16 и представление в виде FWP\_BYTE\_BLOB.  
 // Пример (очень упрощенный, только для демонстрации структуры):  
 size\_t pathLen = wcslen(filePath);  
 size\_t prefixLen = wcslen(L"app\_id://");  
 size\_t totalLen = prefixLen + pathLen; // Длина строки в WCHARs  
   
 FWP\_BYTE\_BLOB\* appIdBlob = (FWP\_BYTE\_BLOB\*)FwpmAlloc(sizeof(FWP\_BYTE\_BLOB));  
 if (!appIdBlob) return nullptr;  
  
 appIdBlob->size = (ULONG)(totalLen \* sizeof(WCHAR)); // Размер в байтах  
 appIdBlob->data = (UINT8\*)FwpmAlloc(appIdBlob->size);  
 if (!appIdBlob->data) {  
 FwpmFreeMemory0((void\*\*)&appIdBlob);  
 return nullptr;  
 }  
  
 // Копирование префикса и пути (в реальном коде использовать безопасные функции, например StringCchCopyW)  
 wcscpy\_s((WCHAR\*)appIdBlob->data, totalLen + 1, L"app\_id://");  
 wcscat\_s((WCHAR\*)appIdBlob->data, totalLen + 1, filePath);  
  
 return appIdBlob;  
}  
  
// Функция для добавления WFP фильтра для исключения приложения  
DWORD AddAppExclusionFilter(HANDLE engineHandle, const WCHAR\* appPath) {  
 FWPM\_FILTER0 filter;  
 memset(&filter, 0, sizeof(filter)); // Обнуление структуры  
  
 // Заполнение отображаемых данных фильтра  
 filter.displayData.name = (WCHAR\*)L"MyVpnAppExclusionFilter";  
 filter.displayData.description = (WCHAR\*)L"Excludes specified application traffic from VPN tunnel.";  
   
 // Установка слоя и подуровня  
 filter.layerKey = FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4; // Фильтруем исходящие TCP/UDP соединения IPv4 // [1], 4.3  
 filter.subLayerKey = YOUR\_SUBLAYER\_GUID; // Ваш кастомный подуровень с высоким приоритетом // [1], 4.2  
   
 // Установка веса фильтра (приоритета)  
 filter.weight.type = FWP\_UINT8;  
 filter.weight.uint8 = FWP\_FILTER\_WEIGHT\_MAX; // Максимальный приоритет для этого фильтра  
  
 // Установка действия фильтра: передать управление драйверу-выноске  
 filter.action.type = FWP\_ACTION\_CALLOUT\_TERMINATING; // [1], 4.3  
 filter.action.calloutKey = YOUR\_CALLOUT\_GUID\_V4; // GUID вашего драйвера-выноски для IPv4  
  
 // Создание условия фильтра по App ID  
 FWPM\_FILTER\_CONDITION0 condition;  
 memset(&condition, 0, sizeof(condition));  
 condition.fieldKey = FWPM\_CONDITION\_APP\_ID; // Фильтрация по идентификатору приложения // [1], 4.3  
 condition.matchType = FWP\_MATCH\_EQUAL; // Точное совпадение  
  
 // Получение App ID блоба с помощью кастомной функции  
 FWP\_BYTE\_BLOB\* appIdBlob = CreateAppIdBlobFromPath(appPath); // Использование кастомной функции  
 if (!appIdBlob) {  
 std::cerr << "Failed to create App ID blob for: " << appPath << std::endl;  
 return ERROR\_OUTOFMEMORY; // Или другая подходящая ошибка  
 }  
 condition.conditionValue.type = FWP\_BYTE\_BLOB\_TYPE;  
 condition.conditionValue.byteBlob = appIdBlob;  
  
 filter.numFilterConditions = 1;  
 filter.filterCondition = &condition;  
  
 DWORD result = FwpmFilterAdd0(engineHandle, &filter, NULL, NULL); // [1], 4.3  
 if (result!= ERROR\_SUCCESS) {  
 std::cerr << "FwpmFilterAdd0 failed for IPv4: " << result << std::endl;  
 }  
  
 // Освобождение памяти, выделенной для App ID блоба  
 FwpmFreeMemory0((void\*\*)&appIdBlob->data);  
 FwpmFreeMemory0((void\*\*)&appIdBlob);  
  
 // Повторить для FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V6 и YOUR\_CALLOUT\_GUID\_V6  
 // (аналогичная логика, только меняются layerKey и calloutKey)  
 filter.layerKey = FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V6;  
 filter.action.calloutKey = YOUR\_CALLOUT\_GUID\_V6;  
 appIdBlob = CreateAppIdBlobFromPath(appPath); // Создаем новый блоб для IPv6 фильтра  
 if (!appIdBlob) {  
 std::cerr << "Failed to create App ID blob for IPv6: " << appPath << std::endl;  
 return ERROR\_OUTOFMEMORY;  
 }  
 condition.conditionValue.byteBlob = appIdBlob;  
   
 DWORD resultV6 = FwpmFilterAdd0(engineHandle, &filter, NULL, NULL);  
 if (resultV6!= ERROR\_SUCCESS) {  
 std::cerr << "FwpmFilterAdd0 failed for IPv6: " << resultV6 << std::endl;  
 }  
 FwpmFreeMemory0((void\*\*)&appIdBlob->data);  
 FwpmFreeMemory0((void\*\*)&appIdBlob);  
  
 return (result == ERROR\_SUCCESS && resultV6 == ERROR\_SUCCESS)? ERROR\_SUCCESS : ERROR\_GEN\_FAILURE;  
}  
  
// Пример использования в транзакции:  
// HANDLE engineHandle = NULL;  
// DWORD status = FwpmEngineOpen0(NULL, RPC\_C\_AUTHN\_WINNT, NULL, NULL, &engineHandle);  
// if (status == ERROR\_SUCCESS) {  
// status = FwpmTransactionBegin0(engineHandle, 0); // Начать транзакцию  
// if (status == ERROR\_SUCCESS) {  
// // Добавление поставщика и подуровня (выполняется один раз при инициализации службы)  
// // FwpmProviderAdd0(...);  
// // FwpmSubLayerAdd0(...);  
//  
// // Добавление фильтра для конкретного приложения  
// status = AddAppExclusionFilter(engineHandle, L"C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe");  
// if (status == ERROR\_SUCCESS) {  
// status = FwpmTransactionCommit0(engineHandle); // Зафиксировать транзакцию  
// if (status == ERROR\_SUCCESS) {  
// std::cout << "Filter added successfully and committed." << std::endl;  
// } else {  
// std::cerr << "FwpmTransactionCommit0 failed: " << status << std::endl;  
// FwpmTransactionAbort0(engineHandle); // Откатить в случае ошибки  
// }  
// } else {  
// std::cerr << "AddAppExclusionFilter failed: " << status << std::endl;  
// FwpmTransactionAbort0(engineHandle); // Откатить  
// }  
// } else {  
// std::cerr << "FwpmTransactionBegin0 failed: " << status << std::endl;  
// }  
// FwpmEngineClose0(engineHandle);  
// } else {  
// std::cerr << "FwpmEngineOpen0 failed: " << status << std::endl;  
// }

## **V. Драйвер-Выноска (C++): Перенаправление Трафика в Режиме Ядра**

### **5.1. Необходимость Драйвера для Перенаправления Соединений**

Хотя фильтры WFP, описанные в предыдущей части, могут разрешать или блокировать трафик приложения, они не управляют его маршрутизацией. Если в конфигурации WireGuard указано AllowedIPs = 0.0.0.0/0, то системная таблица маршрутизации все равно попытается направить разрешенный трафик в туннель.1 Простое разрешение (

Permit) трафика на уровне ALE\_AUTH\_CONNECT недостаточно, так как решение о маршрутизации принимается на более низких уровнях стека.

Чтобы по-настоящему разделить трафик и принудительно направить его на физический сетевой интерфейс в обход VPN-туннеля, необходимо вмешаться в процесс установки соединения на низком уровне. WFP предоставляет для этого специальные уровни перенаправления, такие как FWPM\_LAYER\_ALE\_CONNECT\_REDIRECT\_V4 и FWPM\_LAYER\_ALE\_CONNECT\_REDIRECT\_V6. Доступ к этим уровням возможен только из режима ядра через драйвер-выноску (callout driver).1 Перенаправление соединения на уровне

ALE\_CONNECT\_REDIRECT — это единственный надежный способ гарантировать, что трафик пойдет по нужному пути.

**Архитектура решения с драйвером-выноской:**

1. **C++ служба** добавляет WFP-фильтр для целевого приложения (например, chrome.exe).1
2. В качестве действия для этого фильтра указывается FWP\_ACTION\_CALLOUT\_TERMINATING и GUID нашего драйвера-выноски.1
3. **Драйвер-выноска** регистрирует в системе свою функцию-классификатор (classifyFn) на уровнях FWPM\_LAYER\_ALE\_CONNECT\_REDIRECT\_V4 и FWPM\_LAYER\_ALE\_CONNECT\_REDIRECT\_V6.1
4. Когда chrome.exe пытается установить новое соединение, WFP находит соответствующий фильтр и вызывает classifyFn нашего драйвера.1
5. Внутри classifyFn драйвер изменяет параметры соединения, чтобы оно было установлено через физический сетевой интерфейс, а не через WireGuard.1

Драйвер-выноска, работающий на уровне перенаправления соединений WFP, является единственным надежным способом гарантировать, что трафик выбранных приложений полностью обходит VPN-туннель. Это позволяет достичь беспрецедентного уровня контроля над сетевым стеком, поскольку решения о маршрутизации принимаются на самом низком уровне, до того, как соединение будет установлено через WireGuard. Такая глубокая интеграция и манипуляция сетевым потоком обеспечивают полную реализацию раздельного туннелирования, что невозможно достичь только с помощью фильтрации на уровне приложений или IP-адресов.

### **5.2. Реализация Функции classifyFn: Модификация FWPS\_CONNECT\_REQUEST0**

Центральным элементом драйвера является его callback-функция classifyFn. Она получает на вход данные о классифицируемом сетевом событии и должна принять решение о его судьбе.1

Алгоритм перенаправления в classifyFn 1:

1. **Получение хендла классификации:** Вызывается FwpsAcquireClassifyHandle0(), чтобы получить хендл, который будет использоваться в последующих вызовах.1
2. **Получение изменяемых данных соединения:** Вызывается FwpsAcquireWritableLayerDataPointer0(). Эта функция возвращает указатель на структуру FWPS\_CONNECT\_REQUEST0, содержащую все параметры исходящего соединения (локальный/удаленный IP и порт, PID процесса и т.д.).1
3. **Модификация параметров соединения:** Это ключевой шаг. Драйвер изменяет поля в структуре FWPS\_CONNECT\_REQUEST0.1
   * **Перенаправление на локальный прокси (стандартный паттерн):** Можно изменить remoteAddressAndPort на адрес локального прокси-сервиса (например, 127.0.0.1:8888). В этом случае прокси-сервис, работающий в пользовательском режиме, принимает соединение и уже от своего имени устанавливает новое соединение с исходным адресатом через физический интерфейс.1
   * **Прямое перенаправление на интерфейс (более сложный подход):** Можно попытаться манипулировать параметрами соединения, чтобы заставить сетевой стек направить его через конкретный физический интерфейс, LUID которого можно определить заранее с помощью GetAdaptersAddresses. Этот метод сложнее, но позволяет избежать необходимости в промежуточном прокси.1
4. **Применение изменений:** Вызывается FwpsApplyModifiedLayerData0(), чтобы зафиксировать изменения в сетевом стеке.1
5. **Разрешение модифицированного соединения:** Функция classifyFn должна вернуть FWP\_ACTION\_PERMIT, чтобы разрешить установку уже измененного соединения. В структуре classifyOut устанавливается classifyOut->actionType = FWP\_ACTION\_PERMIT.1
6. **Освобождение хендла:** Вызывается FwpsReleaseClassifyHandle0().1

Разработка драйвера режима ядра для перенаправления трафика является чрезвычайно сложной и рискованной задачей. Ошибки в управлении памятью, некорректная работа с указателями или неверное использование API могут привести к критическим сбоям системы (например, "синему экрану смерти" - BSOD) или созданию серьезных уязвимостей безопасности, позволяющих повысить привилегии. Это требует от разработчика высочайшего уровня внимания к деталям, глубокого понимания архитектуры Windows и сетевого стека, а также применения строгих практик безопасного программирования. Любое упущение на этом уровне может иметь катастрофические последствия для стабильности и безопасности операционной системы.

**Код: FwpsAcquireWritableLayerDataPointer0 и Изменение Параметров Соединения (Псевдокод/Структура)**

C++

// В C++ драйвере-выноске (classifyFn)  
#**include** <fwpsk.h>  
#**include** <fwpmk.h>  
#**include** <netioapi.h> // Для GetAdaptersAddresses (если используется прямое перенаправление на интерфейс)  
#**include** <ws2ipdef.h> // Для INETADDR\_SETLOOPBACK и других макросов IP-адресов  
#**include** <in6addr.h> // Для IPv6 адресов  
#**include** <iostream> // Для DbgPrint (в режиме ядра используется DbgPrint)  
  
// Заглушки GUID для примера. В реальном коде используйте уникальные GUID.  
// const GUID YOUR\_CALLOUT\_GUID\_V4 = { /\*... \*/ }; // Определен в службе, должен быть известен здесь  
// const GUID YOUR\_CALLOUT\_GUID\_V6 = { /\*... \*/ };  
  
// Глобальная переменная или контекст для хранения LUID физического адаптера  
// LUID physicalAdapterLuid = 0; // Должен быть инициализирован при загрузке драйвера  
 // или передан из пользовательского режима через IOCTL  
  
// Функция classifyFn для WFP драйвера-выноски  
NTSTATUS NTAPI CalloutClassifyFn(  
 \_In\_ const FWPS\_INCOMING\_VALUES0\* inFixedValues,  
 \_In\_ const FWPS\_INCOMING\_METADATA\_VALUES0\* inMetaValues,  
 \_Inout\_opt\_ void\* layerData,  
 \_In\_opt\_ const void\* classifyContext,  
 \_In\_ const FWPS\_FILTER0\* filter,  
 \_In\_ UINT64 flowContext,  
 \_Inout\_ FWPS\_CLASSIFY\_OUT0\* classifyOut  
) {  
 UNREFERENCED\_PARAMETER(inFixedValues);  
 UNREFERENCED\_PARAMETER(classifyContext);  
 UNREFERENCED\_PARAMETER(flowContext);  
  
 // Инициализация classifyOut  
 classifyOut->actionType = FWP\_ACTION\_CONTINUE; // По умолчанию продолжаем обработку  
 classifyOut->rights &= ~FWPS\_RIGHT\_ACTION\_WRITE; // Запрещаем запись по умолчанию  
  
 // Проверяем, что это наш фильтр, который передал управление  
 // (filter->action.type уже FWP\_ACTION\_CALLOUT\_TERMINATING, т.к. мы здесь)  
 if (!IsEqualGUID(&filter->action.calloutKey, &YOUR\_CALLOUT\_GUID\_V4) &&  
 !IsEqualGUID(&filter->action.calloutKey, &YOUR\_CALLOUT\_GUID\_V6)) {  
 DbgPrint("MyWfpDriver: classifyFn called for unknown callout GUID.\n");  
 return STATUS\_SUCCESS;  
 }  
  
 FWPS\_CONNECT\_REQUEST0\* connectRequest = NULL;  
 UINT64 classifyHandle = 0;  
 NTSTATUS status = STATUS\_SUCCESS;  
  
 // Получаем PID процесса, инициирующего соединение  
 // Это метаданные, которые служба может использовать для принятия решения о фильтрации  
 // (например, если список исключенных PID передается в драйвер через IOCTL)  
 UINT32 processId = 0;  
 if (FWPS\_IS\_METADATA\_FIELD\_PRESENT(inMetaValues, FWPS\_METADATA\_FIELD\_PROCESS\_ID)) {  
 processId = inMetaValues->processId;  
 }  
  
 // Здесь должна быть логика проверки, нужно ли исключить этот процесс из VPN  
 // Например, IsProcessExcluded(processId, inMetaValues->processPath)  
 // Если процесс должен быть исключен:  
 // if (IsProcessExcludedFromVpn(processId)) {  
  
 status = FwpsAcquireClassifyHandle0(classifyContext, 0, &classifyHandle); // [1], 5.2; [1]; [24]  
 if (!NT\_SUCCESS(status)) {  
 DbgPrint("MyWfpDriver: FwpsAcquireClassifyHandle0 failed: %X\n", status);  
 goto Exit;  
 }  
  
 // Получение изменяемых данных соединения  
 status = FwpsAcquireWritableLayerDataPointer0(classifyHandle, filter->layerKey, &layerData); // [1], 5.2; [1]; [25]; [25]  
 if (!NT\_SUCCESS(status)) {  
 DbgPrint("MyWfpDriver: FwpsAcquireWritableLayerDataPointer0 failed: %X\n", status);  
 goto Exit;  
 }  
  
 connectRequest = (FWPS\_CONNECT\_REQUEST0\*)layerData;  
  
 // --- Модификация параметров соединения для перенаправления ---  
 // [1], 5.2; [1]; [25]; [25]  
  
 // Вариант 1: Перенаправление на локальный прокси (127.0.0.1:8888)  
 // Это более простой и распространенный паттерн.  
 // Прокси-сервис в пользовательском режиме должен слушать на 127.0.0.1:8888  
 if (filter->layerKey == FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4) {  
 SOCKADDR\_IN\* remoteAddr = (SOCKADDR\_IN\*)&connectRequest->remoteAddressAndPort;  
 remoteAddr->sin\_addr.S\_un.S\_addr = RtlUlongByteSwap(INET\_ADDR(127, 0, 0, 1)); // 127.0.0.1  
 remoteAddr->sin\_port = RtlUshortByteSwap(8888); // Порт прокси (например)  
 DbgPrint("MyWfpDriver: Redirecting IPv4 connection to 127.0.0.1:8888\n");  
 } else if (filter->layerKey == FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V6) {  
 SOCKADDR\_IN6\* remoteAddr6 = (SOCKADDR\_IN6\*)&connectRequest->remoteAddressAndPort;  
 // Устанавливаем адрес IPv6 loopback ::1  
 IN6ADDR\_SETLOOPBACK(remoteAddr6);  
 remoteAddr6->sin6\_port = RtlUshortByteSwap(8888); // Порт прокси  
 DbgPrint("MyWfpDriver: Redirecting IPv6 connection to [::1]:8888\n");  
 }  
  
 // Вариант 2: Прямое перенаправление на конкретный физический интерфейс  
 // Этот подход сложнее, так как требует знания LUID физического адаптера  
 // и может потребовать дополнительных манипуляций с параметрами  
 // connectRequest->networkInterfaceLuid = physicalAdapterLuid; // [1], 5.2  
 // DbgPrint("MyWfpDriver: Redirecting connection to specific LUID: %llX\n", physicalAdapterLuid);  
  
 // Если используется FwpsRedirectHandleCreate0 (Windows 8+) и локальный прокси  
 // connectRequest->localRedirectHandle = cachedRedirectHandle; // [25]; [25]  
 // connectRequest->localRedirectTargetPID = GetProxyProcessId(); // [25]  
  
 // Применяем изменения к сетевому стеку  
 status = FwpsApplyModifiedLayerData0(classifyHandle, layerData); // [1], 5.2; [1]; [25]  
 if (!NT\_SUCCESS(status)) {  
 DbgPrint("MyWfpDriver: FwpsApplyModifiedLayerData0 failed: %X\n", status);  
 goto Exit;  
 }  
  
 // Разрешаем модифицированное соединение  
 classifyOut->actionType = FWP\_ACTION\_PERMIT; // [1], 5.2; [1]  
 classifyOut->rights |= FWPS\_RIGHT\_ACTION\_WRITE; // Разрешаем запись, так как мы изменили данные  
  
Exit:  
 if (classifyHandle) {  
 FwpsReleaseClassifyHandle0(classifyHandle); // Освобождаем хендл классификации // [1], 5.2; [1]  
 }  
 if (!NT\_SUCCESS(status)) {  
 // Если произошла ошибка, блокируем соединение  
 classifyOut->actionType = FWP\_ACTION\_BLOCK;  
 DbgPrint("MyWfpDriver: classifyFn failed, blocking connection. Status: %X\n", status);  
 }  
 return status;  
}

### **5.3. Коммуникация Между Службой и Драйвером (IOCTL)**

Привилегированная C++ служба должна сообщать драйверу, трафик каких приложений необходимо перенаправлять, или передавать другие динамические параметры. Это классическая задача коммуникации между пользовательским режимом (user-mode) и режимом ядра (kernel-mode).1

**Схема реализации на IOCTL (Input/Output Control):**

1. **Драйвер:** В своей функции DriverEntry создает объект устройства (IoCreateDevice) и символическую ссылку на него (IoCreateSymbolicLink). Также он регистрирует функцию-обработчик для запросов IRP\_MJ\_DEVICE\_CONTROL. Этот обработчик будет получать кастомные команды (IOCTL) и данные от пользовательского режима.1
2. **C++ служба:** Открывает хендл драйвера с помощью функции CreateFile, используя имя символической ссылки (например, \\.\MyWfpDriver). Это позволяет службе получить доступ к устройству драйвера.1
3. **C++ служба:** Отправляет данные в драйвер (например, список PID процессов для перенаправления или пути к исполняемым файлам) с помощью функции DeviceIoControl, указывая кастомный код операции (IOCTL) и буферы с данными.1

Важный Аспект: Усиление Безопасности (Hardening) в Драйвере

Любые данные, поступающие из пользовательского режима, являются недоверенными. Драйвер обязан тщательно проверять все указатели и буферы, полученные через DeviceIoControl. Если этого не сделать, вредоносное приложение может передать в драйвер некорректный указатель на область памяти ядра, что приведет к отказу системы (BSOD) или к уязвимости безопасности, позволяющей повысить привилегии.1

Для безопасной обработки в обработчике IRP\_MJ\_DEVICE\_CONTROL необходимо:

* Оборачивать все операции доступа к пользовательским буферам в блоки try/except.1
* Использовать функции ProbeForRead и ProbeForWrite для проверки того, что переданные указатели и размеры буферов указывают на валидные, доступные для чтения/записи области памяти пользовательского режима.1 Это абсолютное и не подлежащее обсуждению требование безопасности при разработке драйверов.

Строгая валидация входных данных из пользовательского режима и использование механизмов защиты ядра (таких как try/except и ProbeForRead/Write) являются абсолютным и не подлежащим обсуждению требованием безопасности при разработке драйверов. Несоблюдение этих мер безопасности создает прямую угрозу стабильности и целостности операционной системы. Злонамеренное программное обеспечение может использовать некорректно обработанные входные данные для вызова системных сбоев (BSOD) или для получения несанкционированного доступа к привилегированным областям памяти, что может привести к полному компрометации системы. Таким образом, эти меры являются не просто рекомендациями по кодированию, а критически важными защитными барьерами, которые должны быть реализованы для создания надежного и безопасного системного компонента.

**Код: Пример Использования DeviceIoControl (Псевдокод/Структура)**

C++

// В C++ драйвере (DriverEntry и MyDeviceControl)  
#**include** <ntddk.h>  
#**include** <wdf.h> // Если используется WDF (рекомендуется для новых драйверов)  
// Включаем файл с определениями наших IOCTL кодов  
#**include** "MyDriverIoctl.h"   
  
// Имя символической ссылки, через которую пользовательский режим будет взаимодействовать с драйвером  
#**define** DOS\_DEVICE\_NAME L"\\DosDevices\\MyWfpDriver"  
  
// Глобальный указатель на список исключенных путей приложений или PID'ов  
// В реальной реализации это будет более сложная структура данных с синхронизацией  
// std::vector<std::wstring> g\_excludedAppPaths;   
// KSPIN\_LOCK g\_lock; // Для синхронизации доступа к g\_excludedAppPaths  
  
// Функция DriverEntry: точка входа драйвера  
NTSTATUS DriverEntry(  
 \_In\_ PDRIVER\_OBJECT DriverObject,  
 \_In\_ PUNICODE\_STRING RegistryPath  
) {  
 UNREFERENCED\_PARAMETER(RegistryPath);  
 NTSTATUS ntStatus;  
 UNICODE\_STRING ntDeviceName;  
 UNICODE\_STRING ntWin32NameString;  
 PDEVICE\_OBJECT deviceObject = NULL;  
  
 RtlInitUnicodeString(&ntDeviceName, L"\\Device\\MyWfpDriver"); // Имя устройства ядра  
  
 // Создание объекта устройства  
 ntStatus = IoCreateDevice(  
 DriverObject,  
 0,  
 &ntDeviceName,  
 FILE\_DEVICE\_UNKNOWN, // Тип устройства  
 FILE\_DEVICE\_SECURE\_OPEN, // Безопасный доступ  
 FALSE,  
 &deviceObject  
 );  
 if (!NT\_SUCCESS(ntStatus)) {  
 DbgPrint("MyWfpDriver: IoCreateDevice failed: %X\n", ntStatus);  
 return ntStatus;  
 }  
  
 // Инициализация точек входа драйвера  
 DriverObject->MajorFunction = MyCreateClose;  
 DriverObject->MajorFunction = MyCreateClose;  
 DriverObject->MajorFunction = MyDeviceControl; // Наш обработчик IOCTL  
 DriverObject->DriverUnload = MyUnloadDriver;  
  
 // Инициализация символической ссылки Win32  
 RtlInitUnicodeString(&ntWin32NameString, DOS\_DEVICE\_NAME);  
  
 // Создание символической ссылки между именем устройства и именем Win32  
 ntStatus = IoCreateSymbolicLink(&ntWin32NameString, &ntDeviceName); // [1], 5.3; [26]  
 if (!NT\_SUCCESS(ntStatus)) {  
 DbgPrint("MyWfpDriver: IoCreateSymbolicLink failed: %X\n", ntStatus);  
 IoDeleteDevice(deviceObject);  
 }  
  
 // Инициализация спин-лока (если используется)  
 // KeInitializeSpinLock(&g\_lock);  
  
 return ntStatus;  
}  
  
// Обработчик IRP\_MJ\_CREATE и IRP\_MJ\_CLOSE  
NTSTATUS MyCreateClose(  
 \_In\_ PDEVICE\_OBJECT DeviceObject,  
 \_In\_ PIRP Irp  
) {  
 UNREFERENCED\_PARAMETER(DeviceObject);  
 Irp->IoStatus.Status = STATUS\_SUCCESS;  
 Irp->IoStatus.Information = 0;  
 IoCompleteRequest(Irp, IO\_NO\_INCREMENT);  
 return STATUS\_SUCCESS;  
}  
  
// Обработчик IRP\_MJ\_DEVICE\_CONTROL  
NTSTATUS MyDeviceControl(  
 \_In\_ PDEVICE\_OBJECT DeviceObject,  
 \_In\_ PIRP Irp  
) {  
 UNREFERENCED\_PARAMETER(DeviceObject);  
 PIO\_STACK\_LOCATION irpSp = IoGetCurrentIrpStackLocation(Irp);  
 NTSTATUS ntStatus = STATUS\_SUCCESS;  
 ULONG inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;  
 PVOID inBuf = NULL; // Входной буфер  
  
 // Для METHOD\_BUFFERED входной буфер находится в Irp->AssociatedIrp.SystemBuffer  
 // Для других методов (DIRECT, NEITHER) нужно использовать Irp->UserBuffer и ProbeForRead/Write  
 if (irpSp->Parameters.DeviceIoControl.IoControlCode == IOCTL\_MY\_ADD\_APP\_PATH ||  
 irpSp->Parameters.DeviceIoControl.IoControlCode == IOCTL\_MY\_REMOVE\_APP\_PATH) {  
 inBuf = Irp->AssociatedIrp.SystemBuffer;  
 } else {  
 // Для других IOCTL, где буфер не METHOD\_BUFFERED, требуется ProbeForRead/Write  
 inBuf = Irp->UserBuffer;  
 }  
   
 // [1], 5.3; [26, 29, 30]: Защита ядра от недоверенных данных  
 // Оборачиваем доступ к пользовательским буферам в try/except блок  
 \_\_try {  
 switch (irpSp->Parameters.DeviceIoControl.IoControlCode) {  
 case IOCTL\_MY\_ADD\_APP\_PATH: { // Пример IOCTL для добавления пути приложения  
 if (inBufLength < sizeof(WCHAR) |  
| inBuf == NULL) {  
 ntStatus = STATUS\_INVALID\_PARAMETER;  
 DbgPrint("MyWfpDriver: IOCTL\_MY\_ADD\_APP\_PATH: Invalid input buffer.\n");  
 break;  
 }  
 // Проверка валидности буфера для чтения (если не METHOD\_BUFFERED)  
 // ProbeForRead(inBuf, inBufLength, sizeof(WCHAR));   
  
 // Предполагаем, что inBuf содержит WCHAR\* путь к приложению  
 WCHAR\* appPath = (WCHAR\*)inBuf;  
 // Длина строки должна быть корректно передана, чтобы избежать чтения за пределами буфера  
 // Например, передавать длину как отдельный параметр IOCTL или использовать null-терминатор  
 // AddExcludedAppPath(appPath); // Добавить appPath в список исключений драйвера  
 DbgPrint("MyWfpDriver: Received ADD\_APP\_PATH: %ws\n", appPath);  
 // Пример: добавить путь в список для classifyFn  
 // KIRQL oldIrql;  
 // KeAcquireSpinLock(&g\_lock, &oldIrql);  
 // g\_excludedAppPaths.push\_back(appPath); // Осторожно: wstring в ядре сложнее  
 // KeReleaseSpinLock(&g\_lock, oldIrql);  
 break;  
 }  
 case IOCTL\_MY\_REMOVE\_APP\_PATH: { // Пример IOCTL для удаления пути приложения  
 if (inBufLength < sizeof(WCHAR) |  
| inBuf == NULL) {  
 ntStatus = STATUS\_INVALID\_PARAMETER;  
 DbgPrint("MyWfpDriver: IOCTL\_MY\_REMOVE\_APP\_PATH: Invalid input buffer.\n");  
 break;  
 }  
 WCHAR\* appPath = (WCHAR\*)inBuf;  
 DbgPrint("MyWfpDriver: Received REMOVE\_APP\_PATH: %ws\n", appPath);  
 // Логика удаления из списка  
 break;  
 }  
 //... другие IOCTLs  
 default:  
 ntStatus = STATUS\_INVALID\_DEVICE\_REQUEST;  
 DbgPrint("MyWfpDriver: Unknown IOCTL received: %X\n", irpSp->Parameters.DeviceIoControl.IoControlCode);  
 break;  
 }  
 } \_\_except (EXCEPTION\_EXECUTE\_HANDLER) {  
 // Обработка исключений, если доступ к пользовательской памяти вызвал ошибку  
 ntStatus = GetExceptionCode();  
 DbgPrint("MyWfpDriver: Exception in IOCTL handler: %X\n", ntStatus);  
 }  
  
 Irp->IoStatus.Status = ntStatus;  
 Irp->IoStatus.Information = 0; // Количество переданных байт (если применимо)  
 IoCompleteRequest(Irp, IO\_NO\_INCREMENT);  
 return ntStatus;  
}  
  
// Функция MyUnloadDriver: выгрузка драйвера  
VOID MyUnloadDriver(  
 \_In\_ PDRIVER\_OBJECT DriverObject  
) {  
 UNICODE\_STRING uniWin32NameString;  
 RtlInitUnicodeString(&uniWin32NameString, DOS\_DEVICE\_NAME);  
  
 // Удаление символической ссылки  
 IoDeleteSymbolicLink(&uniWin32NameString);  
  
 // Удаление объекта устройства  
 if (DriverObject->DeviceObject!= NULL) {  
 IoDeleteDevice(DriverObject->DeviceObject);  
 }  
 DbgPrint("MyWfpDriver: Driver unloaded.\n");  
}  
  
// --- MyDriverIoctl.h (файл заголовка для IOCTL кодов) ---  
#**ifndef** \_MY\_DRIVER\_IOCTL\_H\_  
#**define** \_MY\_DRIVER\_IOCTL\_H\_  
  
// Определяем тип устройства  
#**define** FILE\_DEVICE\_MY\_WFP\_DRIVER 0x8000 // Выберите уникальное значение  
  
// Определяем IOCTL коды  
// CTL\_CODE(DeviceType, Function, Method, Access)  
#**define** IOCTL\_MY\_ADD\_APP\_PATH \  
 CTL\_CODE(FILE\_DEVICE\_MY\_WFP\_DRIVER, 0x800, METHOD\_BUFFERED, FILE\_ANY\_ACCESS) // [1], 5.3; [28]  
  
#**define** IOCTL\_MY\_REMOVE\_APP\_PATH \  
 CTL\_CODE(FILE\_DEVICE\_MY\_WFP\_DRIVER, 0x801, METHOD\_BUFFERED, FILE\_ANY\_ACCESS)  
  
#**endif** // \_MY\_DRIVER\_IOCTL\_H\_  
  
// В C++ службе (user-mode)  
#**include** <windows.h>  
#**include** <string>  
#**include** <iostream>  
#**include** <vector>  
  
// Включаем файл с определениями IOCTL кодов  
#**include** "MyDriverIoctl.h"   
  
// Функция для отправки пути приложения в драйвер  
bool SendAppPathToDriver(const std::wstring& appPath, DWORD ioctlCode) {  
 HANDLE hDevice = CreateFile[1, 28];  
  
 if (hDevice == INVALID\_HANDLE\_VALUE) {  
 std::wcerr << L"Failed to open driver handle: " << GetLastError() << std::endl;  
 return false;  
 }  
  
 DWORD bytesReturned;  
 // Отправка команды и данных в драйвер  
 BOOL success = DeviceIoControl(  
 hDevice,  
 ioctlCode, // Ваш IOCTL код (например, IOCTL\_MY\_ADD\_APP\_PATH) // [1], 5.3; [28]  
 (LPVOID)appPath.c\_str(), // Входной буфер: путь к приложению  
 (DWORD)(appPath.length() + 1) \* sizeof(WCHAR), // Размер входного буфера (с учетом null-терминатора)  
 NULL, // Выходной буфер (не используется в этом примере)  
 0, // Размер выходного буфера  
 &bytesReturned,  
 NULL // OVERLAPPED структура (для асинхронных операций, не используется здесь)  
 );  
  
 CloseHandle(hDevice); // Закрываем хендл драйвера  
 if (!success) {  
 std::wcerr << L"DeviceIoControl failed with code " << ioctlCode << L": " << GetLastError() << std::endl;  
 return false;  
 }  
 return true;  
}  
  
// Пример использования в службе:  
// SendAppPathToDriver(L"C:\\Program Files\\MyApplication\\app.exe", IOCTL\_MY\_ADD\_APP\_PATH);  
// SendAppPathToDriver(L"C:\\Program Files\\AnotherApp\\app.exe", IOCTL\_MY\_REMOVE\_APP\_PATH);

### **5.4. Обзор Открытых WFP Драйверов с Примерами**

Хотя прямые "золотые жилы" с полной реализацией split tunneling на уровне драйвера, готовые к использованию, могут быть ограничены, существует несколько полезных открытых проектов, демонстрирующих работу с WFP и драйверами-выносками, которые могут служить отправной точкой для разработки. Эти ресурсы предоставляют ценные строительные блоки и примеры реализации низкоуровневых операций, значительно ускоряя процесс разработки, но требуют глубокого понимания и адаптации для создания полноценного продукта, отвечающего специфическим требованиям раздельного туннелирования.

**Таблица 3: Обзор Релевантных Открытых WFP Драйверов и Ресурсов**

| Название проекта/ресурса | URL/GitHub | Краткое описание | Ценность для задачи Split Tunneling |
| --- | --- | --- | --- |
| **Windows Filtering Platform Sample (Microsoft WDK Samples)** | [learn.microsoft.com](https://learn.microsoft.com/en-us/samples/microsoft/windows-driver-samples/windows-filtering-platform-sample/) | Официальные образцы WFP-драйверов от Microsoft, демонстрирующие различные аспекты WFP, включая classifyFn и работу с потоками. Включает WFPSamplerCalloutDriver.Inf и ClassifyFunctions\_ProxyCallouts.cpp. | Предоставляет фундаментальные примеры работы с WFP, регистрации драйверов-выносок, базовой логики classifyFn и перенаправления соединений. Отличная отправная точка для понимания основ. 15 |
| **BOT-Man-JL/WFP-Traffic-Redirection-Driver** | (<https://github.com/BOT-Man-JL/WFP-Traffic-Redirection-Driver>) | Форк "Windows Filtering Platform Traffic Inspection Sample", демонстрирующий перенаправление трафика на сетевом и фреймовом уровнях. Содержит inspect.h/c для логики классификации/реинжекции. | Полезен для изучения продвинутых техник перенаправления трафика и логики обработки пакетов в режиме ядра. 34 |
| **oreliyahu1/NetFilterDriver** | (<https://github.com/oreliyahu1/NetFilterDriver>) | Пример драйвера ядра Windows на базе WFP для фильтрации/взаимодействия с обработкой пакетов. Содержит NetFilterDriver\_WfpCallouts.cpp и NetFilterDriver\_Wfp.cpp. | Демонстрирует базовую структуру WFP-драйвера, регистрацию выносок и взаимодействие с пакетами. Может служить примером для общей архитектуры драйвера. 37 |
| **mullvad/libwfp** | [github.com/mullvad/libwfp](https://github.com/mullvad/libwfp) | C++ библиотека для взаимодействия с WFP. Предоставляет строители для определения провайдеров, фильтров и условий. | Хотя это не драйвер, библиотека упрощает взаимодействие с WFP API из пользовательского режима (C++ службы), помогая в создании фильтров и управлении WFP-объектами. 38 |
| **netero1010/EDRSilencer** | (<https://github.com/netero1010/EDRSilencer>) | Инструмент для обхода EDR-решений путем блокировки их исходящего трафика через WFP. | Исходный код (EDRSilencer.c, utils.c) содержит реализацию ручного формирования App ID блоба, что крайне полезно для обхода FwpmGetAppIdFromFileName0 и обеспечения совместимости с EDR/AV. 18 |

## **VI. Развертывание и Подписание Драйвера: Путь к Продукту**

### **6.1. Цифровая Подпись Драйвера: Тестовый Режим vs. Аттестационная Подпись Microsoft**

Создание функционального драйвера — это только половина дела. Для его работы на компьютерах пользователей необходимо решить сложные вопросы цифровой подписи.1

**Этап разработки (тестовая подпись):** На тестовой машине, где будет проводиться отладка, администратор должен включить тестовый режим подписи командой bcdedit /set testsigning on (требуется перезагрузка).1 В свойствах проекта драйвера в Visual Studio можно установить режим подписи "Test Sign". При сборке проекта будет сгенерирован тестовый сертификат, и

.sys файл будет подписан им. Этого достаточно для загрузки драйвера только на машинах с включенным тестовым режимом.1

Продуктовый релиз (аттестационная подпись):

Неоспоримая реальность: Начиная с Windows 10 версии 1607, при включенной функции Secure Boot, операционная система не загрузит ни один новый драйвер режима ядра, если он не подписан Microsoft через портал разработчиков оборудования. Самостоятельная подпись драйвера, даже с использованием коммерческого сертификата для подписи кода, недостаточна.1 Попытка установить некорректно подписанный драйвер на обычной пользовательской машине с включенным Secure Boot приведет либо к блокировке установки, либо к ошибке "Secure Boot Violation" при загрузке системы, что является абсолютно неприемлемым для конечного пользователя.1

Процесс получения подписи Microsoft (Attestation Signing) 1:

1. **Приобретение EV-сертификата (Extended Validation):** Необходимо купить EV Code Signing сертификат. Он является обязательным условием для регистрации аккаунта на портале Microsoft.
2. **Регистрация в Windows Hardware Developer Program:** Создать учетную запись на Partner Center (ранее - SysDev).
3. **Упаковка драйвера в CAB-файл:** Все файлы драйвера (.sys, .inf, .cat, .pdb) упаковываются в один CAB-архив с помощью утилиты makecab.exe.
4. **Подпись CAB-файла:** Созданный CAB-архив подписывается вашим EV-сертификатом.
5. **Отправка на портал:** Подписанный CAB-файл загружается на Partner Center для "аттестационной подписи". Microsoft автоматически проверяет пакет и, в случае успеха, переподписывает его своей подписью. В результате вы получаете .cat файл, подписанный Microsoft.
6. **Результат:** Драйвер с таким каталогом будет доверенным на всех версиях Windows и будет корректно загружаться при включенном Secure Boot.

Процесс аттестационной подписи Microsoft является не просто технической процедурой, а значительным регуляторным и доверительным барьером, установленным Microsoft для обеспечения целостности ядра Windows. Этот процесс требует существенных организационных и финансовых затрат, включая приобретение EV-сертификата и прохождение многоступенчатой процедуры на портале Partner Center. Понимание и планирование этого сложного процесса на ранних стадиях разработки продукта является критически важным для его успешного публичного распространения и установления доверия пользователей. Без аттестационной подписи драйвер не сможет быть установлен на большинстве современных пользовательских систем, что сделает продукт нежизнеспособным.

### **6.2. Создание Надежного Установщика (WiX/NSIS)**

Установщик должен корректно развернуть все компоненты приложения: GUI на C#, привилегированную службу на C++ и драйвер режима ядра.1

**WiX Toolset:** Мощный инструмент на основе XML для создания MSI-пакетов. Для установки и управления службой Windows используются элементы <ServiceInstall> и <ServiceControl>.1 Сам по себе установщик Windows (MSI) не поддерживает установку драйверов режима ядра (атрибут

Type="kernelDriver" в ServiceInstall не работает). Стандартной практикой является использование Custom Action для запуска утилиты DPInst.exe (Driver Package Installer) из состава WDK или PnPutil.exe (встроен в Windows). DPInst.exe и PnPutil.exe корректно обрабатывают INF-файл и устанавливают драйвер в систему.1

**NSIS (Nullsoft Scriptable Install System):** Альтернатива на основе скриптов. Для установки службы можно использовать плагин SimpleSC::Install или вызывать системную утилиту sc.exe create.1 Для установки драйвера используется команда

nsExec::ExecToLog для вызова PnPutil.exe или DPInst.exe с путем к INF-файлу.1

### **6.3. Чистое Удаление: Очистка WFP Объектов**

Установщик должен обеспечивать полное и чистое удаление всех компонентов. При удалении приложения необходимо не только остановить и удалить службу и драйвер, но и корректно удалить все созданные WFP-объекты (фильтры, подуровни, поставщика). Это делается вызовами FwpmFilterDeleteByKey0, FwpmSubLayerDeleteByKey0 и FwpmProviderDeleteByKey0 из кода службы перед ее остановкой.1 Образцы WFP от Microsoft содержат примеры кода для корректной деинсталляции.1

Тщательная проработка процесса установки и удаления, включая корректную очистку всех системных ресурсов, особенно WFP-объектов, является критически важной для обеспечения профессионального качества продукта и положительного пользовательского опыта. Неполное удаление может оставить "мусор" в системе, такой как недействительные фильтры или записи о поставщиках WFP, что может привести к конфликтам с другими приложениями, системной нестабильности или непредсказуемому сетевому поведению. Такие "остатки" негативно влияют на репутацию продукта и увеличивают нагрузку на службу поддержки. Обеспечение чистого удаления демонстрирует внимание к деталям и уважение к пользовательской системе, что способствует долгосрочной удовлетворенности и доверию к продукту.

## **VII. Производительность и Будущие Перспективы**

### **7.1. Анализ Производительности WFP Решения**

При выборе технологии для сетевой фильтрации важно учитывать ее производительность. Хотя NDIS-фильтры работают на более низком уровне сетевого стека, для задач IP-фильтрации на современных версиях Windows (начиная с Vista) WFP является более производительным и эффективным решением.1 Это связано с тем, что WFP-выноски обрабатывают только релевантный трафик (согласно условиям фильтра) и работают с нативной для современного стека структурой

NET\_BUFFER\_LIST, избегая затратных преобразований в устаревший формат NDIS\_PACKET, необходимых для старых NDIS IM-драйверов.1 Некоторые тесты показывают, что для чистого захвата всех пакетов WFP может быть медленнее, но для нашей задачи перенаправления, где важен контекст высокого уровня (например, App ID), WFP является единственным подходящим и современным выбором.1

Накладные расходы, связанные с переходом контекста из пользовательского режима (C++ служба) в режим ядра (драйвер-выноска) и обратно, не являются бесплатной операцией с точки зрения производительности процессора.1 Однако для нашей задачи, где решения принимаются на уровне установки соединения (а не для каждого пакета), эти накладные расходы пренебрежимо малы и не окажут заметного влияния на пропускную способность сети.1

Выбор WFP для реализации раздельного туннелирования является технологически обоснованным и производительным решением. WFP, будучи современной платформой, разработанной Microsoft для сетевой фильтрации в Windows, обеспечивает эффективную работу с минимальными накладными расходами на уровне соединения. Это позволяет избежать проблем с производительностью, которые могли бы возникнуть при использовании устаревших или менее оптимизированных подходов, гарантируя, что функция раздельного туннелирования не будет негативно влиять на общую скорость и отзывчивость сетевого подключения пользователя.

### **7.2. eBPF для Windows: Актуальность WFP в Будущем**

Этот раздел представляет взгляд в будущее, чтобы оценить долгосрочную актуальность выбранной архитектуры. Microsoft активно работает над реализацией eBPF (extended Berkeley Packet Filter) для Windows.1

Важно понимать, что eBPF для Windows не заменяет WFP, а строится поверх него. Многие сетевые хуки eBPF, такие как BPF\_CGROUP\_INET4\_CONNECT (используемый для перенаправления соединений), реализованы через существующие уровни WFP, в частности, через уровень перенаправления соединений.1 Это означает, что WFP остается фундаментальным слоем сетевой фильтрации в Windows.

Инвестиции в разработку решения на базе WFP сегодня являются стратегически верными. Накопленные знания и созданная архитектура будут напрямую применимы и легко адаптируемы для eBPF, когда эта технология станет общедоступной в Windows. Это открывает путь к созданию более кросс-платформенного кода (совместимого между Linux и Windows) в будущем.1

Выбор WFP для реализации раздельного туннелирования является стратегически обоснованным, поскольку он обеспечивает совместимость с текущими системными требованиями и открывает путь для плавной миграции к будущим технологиям, таким как eBPF. Тот факт, что eBPF в Windows строится поверх WFP, означает, что инвестиции в изучение и разработку на WFP не являются тупиковыми, а, наоборот, закладывают прочный фундамент для долгосрочной жизнеспособности продукта. Это позволяет разработчикам использовать передовые возможности Windows сегодня, будучи уверенными в том, что их знания и архитектурные решения будут актуальны и в будущем, обеспечивая адаптивность и конкурентоспособность продукта.

## **VIII. Заключение и Рекомендации**

Данный отчет детально проанализировал различные подходы к реализации раздельного туннелирования на уровне приложений для WireGuard VPN-клиента в Windows. Были рассмотрены архитектурные решения, особенности взаимодействия между компонентами пользовательского режима и режима ядра, а также критически важные аспекты развертывания и безопасности.

Настоятельно рекомендуется **Стратегия Б (Перенаправление с помощью WFP и драйвера-выноски)** как наиболее надежное, безопасное и функциональное решение, полностью отвечающее как явным, так и неявным требованиям, предъявляемым к современному VPN-приложению.1 Хотя этот подход требует значительно больших усилий в разработке и сопряжен с необходимостью прохождения сложной процедуры аттестационной подписи драйвера Microsoft, он единственный обеспечивает необходимый уровень контроля над сетевым трафиком и бесшовный пользовательский опыт.

**Таблица 1: Сравнение Методов Реализации Раздельного Туннелирования**

| Метод | Гранулярность | Сложность | Накладные расходы на производительность | Уровень безопасности | Препятствия при развертывании |
| --- | --- | --- | --- | --- | --- |
| WireGuard AllowedIPs | IP/Подсеть | Низкая | Нативная (отсутствуют) | Высокий | Простое (редактирование.conf) |
| WFP-фильтрация по App-ID (Block/Permit) | Приложение | Средняя | Низкие | Высокий | Среднее (требуется привилегированная служба) |
| WFP-перенаправление (драйвер-выноска) | Приложение | Высокая | Очень низкие (на уровне соединения) | Наивысший (при корректной реализации) | Высокое (требуется подпись драйвера Microsoft) 1 |

Эта таблица наглядно демонстрирует компромиссы между различными подходами. Она показывает, что значительное увеличение сложности и трудозатрат на развертывание метода с WFP-перенаправлением полностью оправдывается его непревзойденной гибкостью, производительностью и функциональностью, что делает его единственным выбором для создания профессионального и конкурентоспособного продукта.

**Таблица 2: Ключевые WFP Слои и Условия для Split Tunneling**

| Слой WFP | Описание | Ключевое условие | Действие | Роль в Split Tunneling |
| --- | --- | --- | --- | --- |
| FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4 | Перехватывает попытки установки исходящих TCP/UDP соединений (IPv4). | FWPM\_CONDITION\_APP\_ID (бинарный блоб пути к файлу) | FWP\_ACTION\_CALLOUT\_TERMINATING | Идентификация трафика приложения, инициирующего соединение. |
| FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V6 | Перехватывает попытки установки исходящих TCP/UDP соединений (IPv6). | FWPM\_CONDITION\_APP\_ID (бинарный блоб пути к файлу) | FWP\_ACTION\_CALLOUT\_TERMINATING | Идентификация трафика приложения, инициирующего соединение. |
| FWPM\_LAYER\_ALE\_CONNECT\_REDIRECT\_V4 | Уровень перенаправления соединений (IPv4). Доступен только из режима ядра. | Не применимо (фильтр уже сработал) | FWP\_ACTION\_PERMIT (после модификации) | Перенаправление соединения на физический интерфейс в обход VPN-туннеля. |
| FWPM\_LAYER\_ALE\_CONNECT\_REDIRECT\_V6 | Уровень перенаправления соединений (IPv6). Доступен только из режима ядра. | Не применимо (фильтр уже сработал) | FWP\_ACTION\_PERMIT (после модификации) | Перенаправление соединения на физический интерфейс в обход VPN-туннеля. |

Эта таблица систематизирует ключевые WFP-элементы, необходимые для реализации, делая сложные концепции более понятными и структурированными для разработчика. Она подчеркивает, как различные слои WFP взаимодействуют для достижения цели раздельного туннелирования на уровне приложений, предоставляя четкое представление о точках вмешательства в сетевой стек.

#### Источники

1. Раздельное туннелирование WireGuard в Windows\_.pdf
2. README.md - wireguard-nt - WireGuard implementation for NT kernel, дата последнего обращения: июня 18, 2025, <https://git.zx2c4.com/wireguard-nt/tree/README.md?id=30a2817d913460ed8a23388d3da485cf9347afa3>
3. WireGuard/wireguard-nt: This repo is a mirror only. Official repository is at https://git.zx2c4.com/wireguard-nt - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/WireGuard/wireguard-nt>
4. example.c « example - wireguard-nt - WireGuard implementation for ..., дата последнего обращения: июня 18, 2025, <https://git.zx2c4.com/wireguard-nt/tree/example/example.c>
5. Process.GetProcesses Method (System.Diagnostics) | Microsoft Learn, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process.getprocesses?view=net-9.0>
6. Process.GetCurrentProcess Method (System.Diagnostics) | Microsoft Learn, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process.getcurrentprocess?view=net-9.0>
7. Process.GetProcessesByName Method (System.Diagnostics) | Microsoft Learn, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process.getprocessesbyname?view=net-9.0>
8. ProcessModule.FileName Property (System.Diagnostics) | Microsoft ..., дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processmodule.filename?view=net-9.0>
9. Get All Modules Used By Process In C# - C# Corner, дата последнего обращения: июня 18, 2025, <https://www.c-sharpcorner.com/UploadFile/87b416/get-all-modules-used-by-process3/>
10. Using named pipes for interprocess communication in C# | Michael John Peña, дата последнего обращения: июня 18, 2025, <https://michaeljohnpena.com/blog/namedpipes/>
11. NamedPipeClientStream Class (System.IO.Pipes) | Microsoft Learn, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.pipes.namedpipeclientstream?view=net-9.0>
12. c++ - Marshal an array of strings from C# to C code using p/invoke ..., дата последнего обращения: июня 18, 2025, <https://stackoverflow.com/questions/13317931/marshal-an-array-of-strings-from-c-sharp-to-c-code-using-p-invoke>
13. P/Invoke Tutorial: Passing strings (Part 2) - CodeProject, дата последнего обращения: июня 18, 2025, <https://www.codeproject.com/Articles/401922/P-Invoke-Tutorial-Passing-strings-Part>
14. How To P/Invoke char\* [] in C# - Stack Overflow, дата последнего обращения: июня 18, 2025, <https://stackoverflow.com/questions/25137788/how-to-p-invoke-char-in-c-sharp>
15. win32/desktop-src/FWP/windows-filtering-platform-start-page.md at docs - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/MicrosoftDocs/win32/blob/docs/desktop-src/FWP/windows-filtering-platform-start-page.md>
16. win32/desktop-src/FWP/about-windows-filtering-platform.md at docs - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/MicrosoftDocs/win32/blob/docs/desktop-src/FWP/about-windows-filtering-platform.md>
17. Introduction to the Windows Filtering Platform - Pavel Yosifovich, дата последнего обращения: июня 18, 2025, <https://scorpiosoftware.net/2022/12/25/introduction-to-the-windows-filtering-platform/>
18. test1213145/ms-EDRSilencer: A tool uses Windows ... - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/test1213145/ms-EDRSilencer>
19. How EDRSilencer Helps Attackers Bypass EDR Security Solutions - Packetlabs, дата последнего обращения: июня 18, 2025, <https://www.packetlabs.net/posts/how-edrsilencer-helps-attackers-bypass-edr-security-solutions/>
20. EDRSilencer Disrupting Endpoint Security - ProCircular, дата последнего обращения: июня 18, 2025, <https://blog.procircular.com/edrsilencer-disrupting-endpoint-security>
21. netero1010/EDRSilencer: A tool uses Windows Filtering Platform (WFP) to block Endpoint Detection and Response (EDR) agents from reporting security events to the server. - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/netero1010/EDRSilencer>
22. Releases · netero1010/EDRSilencer - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/netero1010/EDRSilencer/releases>
23. EDRSilencer/EDRSilencer.c at main · netero1010/EDRSilencer - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/netero1010/EDRSilencer/blob/main/EDRSilencer.c>
24. FwpsAcquireClassifyHandle0 function (fwpsk.h) - Windows drivers | Microsoft Learn, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fwpsk/nf-fwpsk-fwpsacquireclassifyhandle0>
25. Using Bind or Connect Redirection - Windows drivers - Learn Microsoft, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/using-bind-or-connect-redirection>
26. Windows-driver-samples/general/ioctl/wdm/sys/sioctl.c at main - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/Microsoft/Windows-driver-samples/blob/master/general/ioctl/wdm/sys/sioctl.c>
27. Ioctl.cpp - microsoft/Windows-driver-samples - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/microsoft/Windows-driver-samples/blob/main/pos/drivers/barcodescanner/Ioctl.cpp>
28. Windows-driver-samples/general/ioctl/wdm/exe/testapp.c at main - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/Microsoft/Windows-driver-samples/blob/master/general/ioctl/wdm/exe/testapp.c>
29. Windows Filtering Platform Stream Edit Sample - Learn Microsoft, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/samples/microsoft/windows-driver-samples/windows-filtering-platform-stream-edit-sample/>
30. Windows Filtering Platform Sample - Learn Microsoft, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/samples/microsoft/windows-driver-samples/windows-filtering-platform-sample/>
31. Windows Driver Development | windows driver samples | Windows ..., дата последнего обращения: июня 18, 2025, <https://joyasystems.com/sample-code%2Fwindows%20driver%20samples%2FWindows%20Filtering%20Platform%20Sample%2FC%2B%2B%2Fsys%2FWFPSamplerCalloutDriver.InX>
32. Download the Windows Driver Kit (WDK) - Learn Microsoft, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk>
33. Stream Inspection - Windows drivers | Microsoft Learn, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/stream-inspection>
34. WFP-Traffic-Redirection-Driver/sys/inspect.c at master · BOT-Man-JL ..., дата последнего обращения: июня 18, 2025, <https://github.com/BOT-Man-JL/WFP-Traffic-Redirection-Driver/blob/master/sys/inspect.c>
35. WFP-Traffic-Redirection-Driver/README.md at master - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/BOT-Man-JL/WFP-Traffic-Redirection-Driver/blob/master/README.md>
36. WFP-Traffic-Redirection-Driver/sys/inspect.h at master - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/BOT-Man-JL/WFP-Traffic-Redirection-Driver/blob/master/sys/inspect.h>
37. NetFilterDriver is a sample of Windows kernel driver that based WFP - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/oreliyahu1/NetFilterDriver>
38. mullvad/libwfp: C++ library for interacting with the Windows Filtering Platform (WFP) - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/mullvad/libwfp>
39. Hulkstance/usermode-kernel-communication - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/Hulkstance/usermode-kernel-communication>
40. mbn-code/The-Kernel-Driver-Guide-External: Kernal Driver Tutorial For Getting Started With Kernel Drivers - GitHub, дата последнего обращения: июня 18, 2025, <https://github.com/mbn-code/The-Kernel-Driver-Guide-External>
41. EchoInst: An Example Wix Install of the WDK Echo Driver - CodeProject, дата последнего обращения: июня 18, 2025, <https://www.codeproject.com/Articles/573183/EchoInst-An-Example-Wix-Install-of-the-WDK-Echo-Dr>
42. Automatic Attestation Signing Windows driver - Oracle Blogs, дата последнего обращения: июня 18, 2025, <https://blogs.oracle.com/linux/post/automatic-attestation-signing>
43. Attestation sign Windows drivers - Learn Microsoft, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/code-signing-attestation>
44. Driver Element (Difxapp Extension) | Docs, дата последнего обращения: июня 18, 2025, <https://docs.firegiant.com/wix3/xsd/difxapp/driver/>
45. PnPUtil Examples - Windows drivers - Learn Microsoft, дата последнего обращения: июня 18, 2025, <https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/pnputil-examples>
46. Driver installation and update - NSIS Wiki - SourceForge, дата последнего обращения: июня 18, 2025, <https://nsis.sourceforge.io/Driver_installation_and_update>