Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
—
Институт компьютерных наук и технологий
Кафедра «Информационная безопасность компьютерных систем»

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

СЕРВИСЫ MS WINDOWS
по дисциплине «Безопасность современных информационных технологий»

Выполнил
студент гр. 33508/3 Проценко Е.Г.
_____

Руководитель Иванов Д.В.
_____

Санкт-Петербург
2016

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Написать программу-сервер и программу-клиент, работающие под Windows XP и 7. Сервер должен работать в качестве сервиса Windows и предоставлять доступ локальным и удаленным клиентам к файловой системе (локальной и удаленной).

Требования:

- необходимо использовать программу-клиент из лабораторной работы №1 (без изменений ее кода);
- в качестве основы программы-сервер необходимо использовать исходный код программы-сервер из лабораторной работы №1 (statefull сервер);
- работа в Windows XP/7 (все SP);
- инсталляция, деинсталляция, запуск, останов из командной строки (реализация SCP);
- корректный останов и перезапуск службы по запросу от SCM (через оснастку "Службы") (в том числе корректное завершение текущих активных соединений с удаленными клиентами).

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1 Об SCM

SCM (Service Control Manager) запускается вместе с системой. Он является RPC сервером, так SCM может управлять сервисами и контролирующими программами на удаленных компьютерах.

SCM предоставляет интерфейс для следующих задач:

- содержание базы данных установленных сервисов;
- запуск сервисов при загрузке системы или по требованию;
- содержание состояния работающих сервисов;
- передача управляющих запросов сервисам;
- блокировка и разблокировка базы данных сервисов.

SCM поддерживает типа handle, который предоставляет доступ к следующим объектам:

- база данных установленных сервисов;
- сами сервисы;
- замок базы данных.

SCManager объект является объектом базы данных. Та, в свою очередь, является контейнером для объектов сервисов. Функция **OpenSCManager** возвращает handle на объект SCManager на определенном компьютере. Этот handle используется для установки, удаления, открытия и перечисления сервисов, а также для работы с замком.

Объект сервиса определяет установленный сервис. Функции **CreateService** и **OpenService** возвращают такой handle.

OpenSCManager, CreateService, OpenService могут запросить разный уровень доступа к SCManager и сервисному объектам. Запрошенный доступ разрешается или запрещается, в зависимости от маркера доступа вызывающего процесса, и дескриптора безопасности соответствующего SCManager или сервисного объекта.

CloseServiceHandle закрывает handles от SCManager и сервисного объекта.

SCM содержит базу данных установленных сервисов и driver services и предоставляет единый и безопасный метод управления ими. База данных содержит информацию о том, как каждый сервис или сервис драйвера должен быть запущен.

2.2 Сервисные программы

Сервисные программы содержат исполняемый код для одного или более сервисов. Сервисная программа, созданная с типом SERVICE_WIN32_OWN_PROCESS содержит код только одного сервиса.Сервисная программа, созданная с типом SERVICE_WIN32_SHARE_PROCESS содержит код для более чем одного сервиса, позволяющий им делить код. Пример сервисной программы, которая занимается этим – это generic service host process, Svchost.exe, which hosts internal Windows services. Заметьте, что Svchost.exe зарезервирован для использования операционной системой и не может использоваться не Windows сервисами. Вместо этого, разработчики должны создавать их собственные сервисные хостовые программы.

Сервисные программы могут быть сконфигурированы так, чтобы работать в контексте пользовательского аккаунта локального, primary или trusted домена. Так же оно может быть сконфигурировано так, чтобы запускаться в специальном сервисном пользовательском режиме.

Следующие топики описывают требуемый интерфейс SCM, который сервисная программа должна иметь:

- Service Entry Point
- Service ServiceMain Function
- Service Control Handler Function

2.3 Service Entry Point

Сервисы в основном написаны как консольное приложения. Entry point консольного приложения это функция **main**. **Main** получает аргументы from **ImagePath** value from the registry key for the service. Больше информации здесь: CreateService function.

Когда SCM запускает сервисную программу, она ожидает когда та вызовет функцию **StartServiceCtrlDispatcher**. Рекомендации:

- Сервис типа SERVICE_WIN32_OWN_PROCESS должен вызывать StartServiceCtrlDispatcher сразу, из главного потока. Вы можете выполнять любые инициализации после того, как сервер начнет работу, как описано в Service ServiceMain Function.
- Если тип сервиса SERVICE_WIN32_SHARE_PROCESS и нужна общая инициализации для всех сервисов в программе, вы можете выполнить инициализацию в main thread перед вызовом **StartServiceCtrlDispatcher**, если это занимает менее 30 секунд. Однако, вы должны создать другой поток, чтобы сделать общую инициализацию, пока главный поток вызывает StartServiceCtrlDispatcher**.** Но по-прежнему все специфичные настройки должны быть выполнены после начала работы сервиса.

Функция **StartServiceCtrlDispatcher** принимает на вход структуру SERVICE_TABLE_ENTRY для каждого сервиса, содержащегося в программе. Каждая структура определяет имя сервиса и entry point для него. For an example, see Writing a Service Program's main Function.

Если функция **StartServiceCtrlDispatcher** успешно выполнилась, то вызывающий поток не возобновит работу, пока для всех работающих сервисов в процессе не будет получено состояние **SERVICE_STOPPED.** SCM отправляет контролирующие запросы к этому потоку через

именованый pipe. Поток выступает в качестве диспетчера управления, выполняя следующие задачи:

- Создание нового потока для вызова соответствующего entry point, когда новый сервис стартует.
- Вызов соответствующей handler function, чтобы обрабатывать сервисные контролирующие запросы.

2.4 Service ServiceMain Function

Когда сервисная контролирующая программа посылает запрос о том, что новый сервис запустился, SCM запускает его и отправляет запрос на запуск на control dispatcher. Control dispatcher создает новый поток, чтобы запустить там ServiceMain function для сервиса. For an example, see Writing a ServiceMain Function.

The ServiceMain function должна выполнять следующие задачи:

1. Инициализация глобальных переменных
2. Вызов функции RegisterServiceCtrlHandler немедленно, чтобы зарегистрировать Handler function, чтобы обрабатывать контролирующие запросы для сервиса. Возвращаемое значение **RegisterServiceCtrlHandler** - service status handle, который будет использоваться в тех вызовах, которые оповещают SCM о статусе сервиса.
3. Выполнять инициализацию. Если время исполнения инициализации ожидается очень коротким (менее одной секунды), то инициализация может быть выполнена прямо в **ServiceMain.** Если время инициализации ожидается дольше, чем одна секнда, то сервис должен использовать одну из следующих техник инициализации:

- Вызываем функцию SetServiceStatus, чтобы сообщить, что SERVICE_RUNNING, но не принимает никакие controls, пока не закончена инициализация. The service does this by calling

SetServiceStatus with **dwCurrentState** set to SERVICE_RUNNING and **dwControlsAccepted** set to 0 in the **SERVICE_STATUS** structure. Это гарантирует, что SCM не отправит никакие контролирующие запросы сервису, пока тот не готов и освобождает SCM для того, чтобы он занимался другими сервисами. Такой подход рекомендован для повышения производительности, особенно для autostart смервисов.

4. Когда инициализация завершена, вызовите **SetServiceStatus,** чтобы задать SERVICE_RUNNING статус сервиса и определить controls, которые сервис готов принимать. For a list of controls, see the **SERVICE_STATUS** structure.
5. Выполните задачи обслуживания или, если нет тасков в ожидании, верните контроль вызывающему. Любые изменения в состоянии сервиса являются основаниями для вызова **SetServiceStatus,** чтобы сообщить информацию о новом состоянии.
6. Если произошла ошибка вовремя инициализации или работы сервиса, сервис должен вызвать **SetServiceStatus**, чтобы обозначить состояние сервиса как SERVICE_STOP_PENDING, если очистка будет долгой. После того, как чистка окончена, вызовите **SetServiceStatus,** чтобы обозначит состояние сервиса как SERVICE_STOPPED from the last thread to terminate. Be sure to set the **dwServiceSpecificExitCode** and **dwWin32ExitCode** members of the **SERVICE_STATUS** structure to identify the error.

2.5 Service Control Handler

Каждый сервис имеет control handler, the **Handler** function, которая вызывается by control dispatcher, когда сервисный процесс получает control request от service control program. Therefore, this function executes in the context of the control dispatcher. For an example, see **Writing a Control Handler Function**.

A service calls the RegisterServiceCtrlHandler or RegisterServiceCtrlHandlerEx function to register its service control handler function.

Когда control handler вызван, сервис должен вызвать SetServiceStatus, чтобы сообщить его стату для SCM только если обработка контролирующего кода приводит к сменен статуса сервиса. Вызывать **SetServiceStatus** не обязательно.

2.6 SCP

Для установки, удаления, запуска и останова сервисов нужно работать с SCM, чтобы начать работу с ним нужно выполнить функцию **OpenSCManager,** по результатам которой мы получим handle на SCM.

Оттуда мы можем достать интересующий нас сервис функцией **OpenService.** Так мы получим handle на сервис.

Теперь, чтобы удалить, запустить, остановить сервис нужно вызвать **DeleteService, StartService, ControlService,** передав аргументов handle на сервис.

Чтобы создать сервис понадобится функция **CreateService.**

3 ВЫВОД

Я познакомился с реализацией сервисов и сервисных приложений.

Так же в интернете были найдены исходные коды для dns, dhcp, ftp, tftp и других сервисов.

В основном примеры кода пришлось искать на github’e, а также смотреть спецификации на MSDN.

4 ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1 Какие операции выполняются при установке сервиса в систему?

Функция **CreateService** создает сервисный объект и устанавливает его в SCM, занося в базу данных, создавая ключ в реестре с соответствующим именем, как и будущий сервис, по следующему пути:
`HKEY_LOCAL_MACHINE\System\CurrentControlSer\Services`

Также в базе данных сохраняется конфигурационная и дополнительная информация, такая как: зависимости, описание, группы, путь к образу, тип загрузки, отображаемое имя, тип сервиса.

4.2 Что такое SCP и зачем необходимо его реализовывать?

SCP (Service Control Programs) – запускают и контролируют сервисы. SCP предоставляют следующие возможности: запуск сервиса, отправление контролирующих запросов на сервис, запрос текущего состояния сервиса.

4.3 Какие возможности по управлению сервисами НЕ предоставляет оснастка “Службы”?

Оснастка “Службы” не предоставляет возможности установки и удаления сервиса.

О программе, управляющей сервисом предоставляют только путь к ее образу, больше никакой дополнительной информации о самом процессе. Ее можно посмотреть в специальных программах типа Process Explorer или Process Hacker.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. http://www.codeproject.com/Articles/499465/Simple-Windows-Service-in-Cplusplus
2. http://stackoverflow.com/
3. https://github.com/
4. https://github.com/sunnyden/reactos/blob/81dce7ad39f0fa5584d0cf1727443d77835e7d7e/base/services/tftpd/tftpd.cpp - исходные коды сервиса tftp-сервера
5. https://msdn.microsoft.com/

ПРИЛОЖЕНИЕ А

```
#include <iostream>
#include "Example1.h"
#include <Windows.h>

#include <stdio.h>
#include <windows.h>
#include <tchar.h>
#include "accctrl.h"
#include "aclapi.h"

char * get_owner(const char * file_name);

bool login_status = false;
char login_actual[256] = "";
char * opened_file[256];
FILE *filik = NULL;

int download_size;

int LogIn(const char * login, const char * password)
{
    std::cout << login << " logged in" <<login_status << std::endl;
    if (login_status == true) return -1;

    BOOL status;

    HANDLE token;

    status = LogonUser(login, //
        ".",
        password, //
        LOGON32_LOGON_NETWORK,
        LOGON32_PROVIDER_DEFAULT,
        &token);

    if (!status)
    {
        std::cout << "LogonUser error: " << GetLastError() << std::endl;
        return GetLastError();
    }

    status = ImpersonateLoggedOnUser(token);
    if (!status)
    {
        std::cout << "Impersonation error: " << GetLastError() << std::endl;
        return GetLastError();
    }

    login_status = true;
    strcpy(login_actual,login);

    return 0;
}

int LogOut()
{
    BOOL status;
    status = RevertToSelf();
    if (!status)
    {
        std::cout << "RevertingToSelf error: " << GetLastError() << std::endl;
        return GetLastError();
    }
    std::cout << login_actual << " logged out!" << std::endl;
    login_status = false;
    strcpy(login_actual,"");
    return 0;
}

int CreateFileJP(const char * file_name)
{
    std::cout << "Creating file JP" << std::endl;
    if (login_status == false) return -1;

    FILE *f;
    f = fopen( file_name, "rb" );
```

```
    if ( f != 0 )
    {
        std::cout << "file exists" << std::endl;
        fclose( f );
        return -1;
    }else
    {
        std::cout << "file_not_exist" << std::endl;
        //fclose( f );
    }

    filik = fopen(file_name, "wb");
    if (!filik) // если есть доступ к файлу,
    {
      return -1;
    }
    //fclose(opened_file);
    return 0;
}

int OpenFileJP(const char * file_name)
{
    if (login_status == false) return -1;

    filik = fopen( file_name, "rb" );

    if ( filik != 0 )
    {
        std::cout << "file exists" << std::endl;
        fseek(filik, 0, SEEK_END);//перемещает указатель, соответствующий потоку hFile, на
новое место расположения отстоящее от SEEK_END на 0 байтов.
        download_size = ftell(filik);
        //if (file_size == -1) return -4;
        fseek(filik, 0, SEEK_SET);
    }
    else
    {
        std::cout << "file_not_exist" << std::endl;
        fclose( filik );
        return -1;
        //fclose( f );
    }
    //fclose(opened_file);
    return 0;
}

int Upload(int buffer,int type)
{
  /*FILE *out;
  out = fopen((char*)file_name,"ab");
  fwrite(buffer,sizeof(char),size,out);
  fclose(out);*/
  //FILE *hFile;
  //fopen_s(&filik, (char *)buffer, "ab");
  //printf("%x", buffer);

  /*for (int i = 0; i < size; i++)
    fprintf(hFile, "%c", buffer[i]);
  fclose(hFile); */

    if (type == 2)
        fclose(filik);
    else
        fputc(buffer, filik);
    return 0;

}

char * get_owner(const char * file_name)
{
    DWORD dwRtnCode = 0;
    PSID pSidOwner = NULL;
    BOOL bRtnBool = TRUE;
    LPTSTR AcctName = NULL;
    LPTSTR DomainName = NULL;
    DWORD dwAcctName = 1, dwDomainName = 1;
    SID_NAME_USE eUse = SidTypeUnknown;
    HANDLE hFile;
    PSECURITY_DESCRIPTOR pSD = NULL;
```

```
// Get the handle of the file object.
hFile = CreateFile(
                  TEXT(file_name),
                  GENERIC_READ,
                  FILE_SHARE_READ,
                  NULL,
                  OPEN_EXISTING,
                  SECURITY_ANONYMOUS,
                  NULL);

// Check GetLastError for CreateFile error code.
if (hFile == INVALID_HANDLE_VALUE) {
          DWORD dwErrorCode = 0;

          dwErrorCode = GetLastError();
          _tprintf(TEXT("CreateFile error = %d\n"), dwErrorCode);
          return NULL;
}



// Get the owner SID of the file.
dwRtnCode = GetSecurityInfo(
                  hFile,
                  SE_FILE_OBJECT,
                  OWNER_SECURITY_INFORMATION,
                  &pSidOwner,
                  NULL,
                  NULL,
                  NULL,
                  &pSD);

CloseHandle(hFile);
// Check GetLastError for GetSecurityInfo error condition.
if (dwRtnCode != ERROR_SUCCESS) {
          DWORD dwErrorCode = 0;

          dwErrorCode = GetLastError();
          _tprintf(TEXT("GetSecurityInfo error = %d\n"), dwErrorCode);
          return NULL;
}

// First call to LookupAccountSid to get the buffer sizes.
bRtnBool = LookupAccountSid(
                  NULL,           // local computer
                  pSidOwner,
                  AcctName,
                  (LPDWORD)&dwAcctName,
                  DomainName,
                  (LPDWORD)&dwDomainName,
                  &eUse);

// Reallocate memory for the buffers.
AcctName = (LPTSTR)GlobalAlloc(
          GMEM_FIXED,
          dwAcctName);

// Check GetLastError for GlobalAlloc error condition.
if (AcctName == NULL) {
          DWORD dwErrorCode = 0;

          dwErrorCode = GetLastError();
          _tprintf(TEXT("GlobalAlloc error = %d\n"), dwErrorCode);
          return NULL;
}

DomainName = (LPTSTR)GlobalAlloc(
       GMEM_FIXED,
       dwDomainName);

// Check GetLastError for GlobalAlloc error condition.
if (DomainName == NULL) {
      DWORD dwErrorCode = 0;

      dwErrorCode = GetLastError();
      _tprintf(TEXT("GlobalAlloc error = %d\n"), dwErrorCode);
      return NULL;

}
```

```
    // Second call to LookupAccountSid to get the account name.
    bRtnBool = LookupAccountSid(
          NULL,                   // name of local or remote computer
          pSidOwner,              // security identifier
          AcctName,               // account name buffer
          (LPDWORD)&dwAcctName,   // size of account name buffer
          DomainName,             // domain name
          (LPDWORD)&dwDomainName, // size of domain name buffer
          &eUse);                 // SID type

    // Check GetLastError for LookupAccountSid error condition.
    if (bRtnBool == FALSE) {
          DWORD dwErrorCode = 0;

          dwErrorCode = GetLastError();

          if (dwErrorCode == ERROR_NONE_MAPPED)
              _tprintf(TEXT
                   ("Account owner not found for specified SID.\n"));
          else
              _tprintf(TEXT("Error in LookupAccountSid.\n"));
          return NULL;

    } else if (bRtnBool == TRUE)

        // Print the account name.
        _tprintf(TEXT("Account owner = %s\n"), AcctName);

    return AcctName;
}

int Delete(const char * file_name)
{
    if (login_status == false) return -1;

    char * owner = get_owner(file_name);
    std::cout << "owner: " << owner << std::endl;
    std::cout << "login_actual: " << login_actual << std::endl;
    if (strcmp(owner, login_actual) == 0)
    {
        if(remove(file_name)) {
            return -2;
        }
        else return 0;
    }
    else return -3;

}

int Download(int * buffer)
{
    if (download_size == 0){
        fclose(filik);
        filik = NULL;
        return 1;
    }
    download_size--;
    *buffer = fgetc(filik);
    //std::cout << download_size << " ";
    if (*buffer != EOF) return 0;
    else
    {
        fclose(filik);
        filik = NULL;
        return 1;
    }
}

// Naive security callback.
RPC_STATUS CALLBACK SecurityCallback(RPC_IF_HANDLE /*hInterface*/, void* /*pBindingHandle*/)
{
    return RPC_S_OK; // Always allow anyone.
}

int rpc()
{
   RPC_STATUS status;

   // Uses the protocol combined with the endpoint for receiving
```

```
    // remote procedure calls.
    status = RpcServerUseProtseqEp(
        reinterpret_cast<unsigned char*>("ncacn_ip_tcp"), // Use TCP/IP protocol.
        RPC_C_PROTSEQ_MAX_REQS_DEFAULT, // Backlog queue length for TCP/IP.
        reinterpret_cast<unsigned char*>("4747"), // TCP/IP port to use.
        NULL); // No security.

    if (status)
        exit(status);

    // Registers the Example1 interface.
    status = RpcServerRegisterIf2(
        Example1_v1_0_s_ifspec, // Interface to register.
        NULL, // Use the MIDL generated entry-point vector.
        NULL, // Use the MIDL generated entry-point vector.
        RPC_IF_ALLOW_CALLBACKS_WITH_NO_AUTH, // Forces use of security callback.
        RPC_C_LISTEN_MAX_CALLS_DEFAULT, // Use default number of concurrent calls.
        (unsigned)-1, // Infinite max size of incoming data blocks.
        SecurityCallback); // Naive security callback.

    if (status)
        exit(status);

    // Start to listen for remote procedure
    // calls for all registered interfaces.
    // This call will not return until
    // RpcMgmtStopServerListening is called.
    status = RpcServerListen(
      1, // Recommended minimum number of threads.
      RPC_C_LISTEN_MAX_CALLS_DEFAULT, // Recommended maximum number of threads.
      FALSE); // Start listening now.

    if (status)
        exit(status);
}

// Memory allocation function for RPC.
// The runtime uses these two functions for allocating/deallocating
// enough memory to pass the string to the server.
void* __RPC_USER midl_user_allocate(size_t size)
{
    return malloc(size);
}

// Memory deallocation function for RPC.
void __RPC_USER midl_user_free(void* p)
{
    free(p);
}


SERVICE_STATUS        g_ServiceStatus = {0};
SERVICE_STATUS_HANDLE g_StatusHandle = NULL;
HANDLE                g_ServiceStopEvent = INVALID_HANDLE_VALUE;

VOID WINAPI ServiceMain (DWORD argc, LPTSTR *argv);
VOID WINAPI ServiceCtrlHandler (DWORD);
DWORD WINAPI ServiceWorkerThread (LPVOID lpParam);

#define SERVICE_NAME  _T("BSIT2")

void InstallService()
{
    SC_HANDLE serviceControlManager = OpenSCManager(0, 0, SC_MANAGER_CREATE_SERVICE);

    if (serviceControlManager)
    {
        TCHAR path[_MAX_PATH + 1];
        if (GetModuleFileName(0, path, sizeof(path) / sizeof(path[0])))
        {
            SC_HANDLE service = CreateService(serviceControlManager,
                SERVICE_NAME, SERVICE_NAME,
                SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
                SERVICE_AUTO_START, SERVICE_ERROR_IGNORE, path,
                0, 0, 0, 0, 0);

            if (service)
            {
                printf("OK: Install service\n");
                CloseServiceHandle(service);
```

```
            }
            else
            {
                printf("ERROR: Install service. Error code: %d\n", GetLastError());
            }
        }

        CloseServiceHandle(serviceControlManager);
    }
}

void UninstallService()
{
    SC_HANDLE serviceControlManager = OpenSCManager(0, 0, SC_MANAGER_CONNECT);

    if (serviceControlManager)
    {
        SC_HANDLE service = OpenService(serviceControlManager,
            SERVICE_NAME, SERVICE_QUERY_STATUS | DELETE);
        if (service)
        {
            SERVICE_STATUS serviceStatus;
            if (QueryServiceStatus(service, &serviceStatus))
            {
                if (serviceStatus.dwCurrentState == SERVICE_STOPPED)
                {
                    if (!DeleteService(service))
                    {
                        printf("ERROR: Delete service\n");
                        return;
                    }
                    else
                    {
                        printf("OK: Delete service\n");
                    }
                }
            }

            CloseServiceHandle(service);
        }

        CloseServiceHandle(serviceControlManager);
    }
}

void RunService()
{
    SERVICE_STATUS_PROCESS ssStatus;
    SC_HANDLE serviceControlManager = OpenSCManager(0, 0, SC_MANAGER_CONNECT);
    DWORD dwBytesNeeded;
    if (serviceControlManager)
    {
        SC_HANDLE hService = OpenService(serviceControlManager, SERVICE_NAME,
SERVICE_ALL_ACCESS);
        if (hService)
        {
            if (!QueryServiceStatusEx(
                hService,                     // handle to service
                SC_STATUS_PROCESS_INFO,         // information level
                (LPBYTE)&ssStatus,              // address of structure
                sizeof(SERVICE_STATUS_PROCESS), // size of structure
                &dwBytesNeeded))              // size needed if buffer is too small
            {
                printf("QueryServiceStatusEx failed (%d)\n", GetLastError());
                CloseServiceHandle(hService);
                CloseServiceHandle(serviceControlManager);
                return;
            }
            if (ssStatus.dwCurrentState != SERVICE_STOPPED && ssStatus.dwCurrentState !=
SERVICE_STOP_PENDING)
            {
                printf("Cannot start the service because it is already running\n");
                CloseServiceHandle(hService);
                CloseServiceHandle(serviceControlManager);
                return;
            }
            if (!StartService(hService, NULL, NULL))
            {
                printf("ERROR: Start service. Error code: %d\n", GetLastError());
                return;
```

```
            }
            else
            {
                printf("OK: Start service\n");
            }
        }
    }

    CloseServiceHandle(serviceControlManager);
}

void StopService()
{
    SC_HANDLE serviceControlManager = OpenSCManager(0, 0, SC_MANAGER_CONNECT);
    if (serviceControlManager)
    {
        SC_HANDLE hService = OpenService(serviceControlManager, SERVICE_NAME, SERVICE_STOP);
        if (hService)
        {
            SERVICE_STATUS ss;
            if (!ControlService(hService, SERVICE_CONTROL_STOP, &ss))
            {
                printf("ERROR: Stop service : %d\n", GetLastError());
                return;
            }
            else
            {
                printf("OK: Stop service\n");
            }
        }
    }
    CloseServiceHandle(serviceControlManager);
}

int _tmain (int argc, char *argv[])
{
    if (argc > 1)
    {
        if (strcmp(argv[1], "-install") == 0)
            InstallService();
        else if (strcmp(argv[1], "-uninstall") == 0)
            UninstallService();
        else if (strcmp(argv[1], "-run") == 0)
            RunService();
        else if (strcmp(argv[1], "-stop") == 0)
            StopService();
    }
    //else
    //{
    SERVICE_TABLE_ENTRY ServiceTable[] =
    {
        {SERVICE_NAME, (LPSERVICE_MAIN_FUNCTION) ServiceMain},
        {NULL, NULL}
    };

    if (StartServiceCtrlDispatcher (ServiceTable) == FALSE)
    {
        return GetLastError ();
    }

    return 0;
    //}



}

VOID WINAPI ServiceMain (DWORD argc, LPTSTR *argv)
{
    DWORD Status = E_FAIL;

    // Register our service control handler with the SCM
    g_StatusHandle = RegisterServiceCtrlHandler (SERVICE_NAME, ServiceCtrlHandler);

    if (g_StatusHandle == NULL)
    {
        goto EXIT;
    }

    // Tell the service controller we are starting
```

```
ZeroMemory (&g_ServiceStatus, sizeof (g_ServiceStatus));
g_ServiceStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
g_ServiceStatus.dwControlsAccepted = 0;
g_ServiceStatus.dwCurrentState = SERVICE_START_PENDING;
g_ServiceStatus.dwWin32ExitCode = 0;
g_ServiceStatus.dwServiceSpecificExitCode = 0;
g_ServiceStatus.dwCheckPoint = 0;

if (SetServiceStatus (g_StatusHandle , &g_ServiceStatus) == FALSE)
{
    OutputDebugString(_T(
      "My Sample Service: ServiceMain: SetServiceStatus returned error"));
}

/*
 * Perform tasks necessary to start the service here
 */

// Create a service stop event to wait on later
g_ServiceStopEvent = CreateEvent (NULL, FALSE, FALSE, NULL);
if (g_ServiceStopEvent == NULL)
{
    // Error creating event
    // Tell service controller we are stopped and exit
    g_ServiceStatus.dwControlsAccepted = 0;
    g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
    g_ServiceStatus.dwWin32ExitCode = GetLastError();
    g_ServiceStatus.dwCheckPoint = 1;

    if (SetServiceStatus (g_StatusHandle, &g_ServiceStatus) == FALSE)
                {
                    OutputDebugString(_T(
                      "My Sample Service: ServiceMain: SetServiceStatus returned error"));
                }
    goto EXIT;
}


// Tell the service controller we are started
g_ServiceStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_PAUSE_CONTINUE;
g_ServiceStatus.dwCurrentState = SERVICE_RUNNING;
g_ServiceStatus.dwWin32ExitCode = 0;
g_ServiceStatus.dwCheckPoint = 0;

if (SetServiceStatus (g_StatusHandle, &g_ServiceStatus) == FALSE)
{
    OutputDebugString(_T(
      "My Sample Service: ServiceMain: SetServiceStatus returned error"));
}

/*// Start a thread that will perform the main task of the service
HANDLE hThread = CreateThread (NULL, 0, ServiceWorkerThread, NULL, 0, NULL);

// Wait until our worker thread exits signaling that the service needs to stop
WaitForSingleObject (hThread, INFINITE);*/

rpc();


/*
 * Perform any cleanup tasks
 */
/*RevertToSelf();
RpcMgmtStopServerListening(NULL);
if (filik)
{
    fclose(filik);
    filik = NULL;
}
login_status = false;*/


CloseHandle (g_ServiceStopEvent);

// Tell the service controller we are stopped
g_ServiceStatus.dwControlsAccepted = 0;
g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
g_ServiceStatus.dwWin32ExitCode = 0;
g_ServiceStatus.dwCheckPoint = 3;
```

```
    if (SetServiceStatus (g_StatusHandle, &g_ServiceStatus) == FALSE)
    {
        OutputDebugString(_T(
          "My Sample Service: ServiceMain: SetServiceStatus returned error"));
    }

EXIT:
    return;
}

//
https://github.com/sunnyden/reactos/blob/81dce7ad39f0fa5584d0cf1727443d77835e7d7e/base/service
s/tftpd/tftpd.cpp
VOID WINAPI ServiceCtrlHandler (DWORD CtrlCode)
{
    switch (CtrlCode)
         {
    case SERVICE_CONTROL_INTERROGATE:
        break;

    case SERVICE_CONTROL_SHUTDOWN:
    case SERVICE_CONTROL_STOP :

        if (g_ServiceStatus.dwCurrentState != SERVICE_RUNNING)
           break;

        /*
         * Perform tasks necessary to stop the service here
         */
         RevertToSelf();
    if (filik)
    {
        fclose(filik);
        filik = NULL;
    }
    login_status = false;
    //RpcMgmtStopServerListening(NULL);

        g_ServiceStatus.dwControlsAccepted = 0;
        g_ServiceStatus.dwCurrentState = SERVICE_STOP_PENDING;
        g_ServiceStatus.dwWin32ExitCode = 0;
        g_ServiceStatus.dwCheckPoint = 4;

        if (SetServiceStatus (g_StatusHandle, &g_ServiceStatus) == FALSE)
        {
            OutputDebugString(_T(
              "My Sample Service: ServiceCtrlHandler: SetServiceStatus returned error"));
        }

        // This will signal the worker thread to start shutting down
        SetEvent (g_ServiceStopEvent);

        return;

    case SERVICE_CONTROL_PAUSE:
        break;

    case SERVICE_CONTROL_CONTINUE:
        break;

    default:
        if (CtrlCode >= 128 && CtrlCode <= 255)
            // user defined control code
            break;
        else
            // unrecognised control code-
            break;
    }

    SetServiceStatus(g_StatusHandle, &g_ServiceStatus);
}

DWORD WINAPI ServiceWorkerThread (LPVOID lpParam)
{
    //  Periodically check if the service has been requested to stop
    while (WaitForSingleObject(g_ServiceStopEvent, 0) != WAIT_OBJECT_0)
    {
        /*
         * Perform main service function here
         */
```

```
        //  Simulate some work by sleeping
        Sleep(3000);
    }

    return ERROR_SUCCESS;
}
```