**Жесткое внедрение DLL в Windows-программы**

На суд читателя представляется статья о внедрении собственного исполняемого кода в Windows программы.

В статье использованы различные технические термины, относящиеся к программированию под ОС Windows, в частности с применением Windows API. Тексты программ, приведенные в качестве примеров, тестировались в среде Borland C++ Builder 6.0. Работа приложения проверялась в среде Windows 2000, среда Windows 9x в силу морального устаревания не рассматривалась.

**Введение в специальность**

Внедрение своего кода в чужую программу может понадобится для множества задач, например, для отладки и анализа работы приложений, для установки различного рода программных защит, для "шпионских" целей. На сегодняшний день существует достаточно много методов внедрить свой код в чужую программу. Достаточно подробно они описаны в статье автора Tanaka ["Программы-невидимки"](http://bugtraq.ru/library/programming/invisible.html), Однако все описанные методы являются внешними по отношению к целевой программе, т.е. для внедрения кода каждый раз необходим запуск постороннего приложения, так называемого "загрузчика", осуществляющего внедрение. Предлагаемый способ отличается тем, что осуществляет разовую, достаточно простую модификацию внутренних структур .exe файла, не модифицирует исполняемый код программы, не увеличивает размер файла, не требует внешнего загрузчика, практически незаметен для пользователя. Метод позволяет модифицировать большинство .exe файлов.

**Как это работает**

Принцип работы основан на следующем: практически любая Windows-программа использует динамические библиотеки (DLL). В них могут храниться различные функции (в том числе системные - в USER32.DLL, GDI.DLL и т.д.), ресурсы типа диалогов, иконок, картинок, указателей мыши. Достаточно сложно найти программу, которая не использовала бы DLL вообще. Соответственно, программа, использующая DLL-библиотеки, содержит в своем .exe файле информацию о том, какие именно библиотеки ей нужны, и какие функции из этих библиотек она использует (импортирует). При запуске любой программы системный загрузчик Windows считывает список используемых этой программой DLL-библиотек и загружает (отображает) их в адресное пространство программы. После этого для каждой импортируемой программой функции загрузчик определяет адрес вызова.

 Каждая DLL-библиотека содержит функцию с именем DllMain следующего вида:

BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad);

 Назначение её - сугубо информационное. Вызывая эту функцию, загрузчик сообщает библиотеке о том, что она будет подключена к какому-то процессу, либо в контексте процесса происходит создание потока. Загрузка любой программы включает в себя последовательный вызов функций DllMain всех используемых DLL. Аналогично данный вызов осуществляется при отключении и выгрузке библиотеки. Думаю, что суть метода вам уже ясна: достаточно добавить к списку используемых программой DLL-библиотек свою, у которой функция DllMain содержит необходимый вам код. Весь код в рамках этой ф-ции будет выполнятся с приоритетом "заряженной" программы.

 В меру скромного воображения приведу несколько примеров использования данной методики для решения практических задач. Первое, что приходит в голову - это система "навесной" защиты, которую можно установить на любое готовое приложение. Метод проверки допуска может быть любым - от простейшего пароля, до обращения к внешнему устройству, содержащему private-key для декодирования части исполняемого кода программы (например, USB-ключ). Далее - как было сказано в предисловии - любые виды троянов/вирусов.

**Рецепт**

Что же, теперь нам необходимо изменить .exe файл таким образом, что бы в списке используемых программой DLL библиотек появилась наша библиотека. На первый взгляд данная задача представляется как "темный лес" - где этот список взять, чего в нем искать, как чего менять? Но всё не так печально! К нашему с вами счастью, формат .exe файла Windows достаточно строго стандартизирован и подробно описан в документации. Желающим подробно поковыряться во внутренностях могу посоветовать ознакомиться с вот этим документом:

 Peering Inside the PE: A Tour of the Win32 Portable Executable File Format,

для остальных я постараюсь привести здесь минимум информации, необходимый для реализации программы, "заряжающей" нашим кодом почти любой .exe файл. (Относительно ограничений метода - см. гл. Выводы) Приступам к пациенту. Формат файла программы, так же называемый "переносимым исполнительным" (PE - portable executable), определяет поведение операционной системы на всех этапах работы - начиная от отображения файла на адресное пространство процесса, загрузки необходимых библиотек, инициализации ресурсов, до собственно выгрузки программы. Самое важное из того, что следует знать о РЕ-файлах, это то, что исполняемый файл на диске и модуль, получаемый после загрузки, очень похожи. Причиной этого является то, что загрузчик Windows должен создать из дискового файла исполняемый процесс без больших усилий. Точнее говоря, загрузчик попросту использует механизмы отображения файлов в память, чтобы загрузить соответствующие части РЕ-файла в адресное пространство программы.\* Так же просто загружается и DLL. После того как ЕХЕ или .DLL модули загружены, Windows обращается с ними так же, как и с другими отображенными в память файлами.

 \* См. справку по функция CreateFile, MapViewOfFile, CreateFileMapping.

 Так как структура исполнительного файла является довольно громоздкой, вникать подробно в описание каждого её элемента мы не будет, лишь кратко остановимся на необходимых нам элементах. Все файлы программ для Windows имеют следующий формат (см. рис. 1):

рис. 1 Структура исполнительного файла.

 1. Заголовок MSDOS Начиная с нулевого смещения, в файле располагается заголовок MSDOS, имеющий формат IMAGE\_DOS\_HEADER (см. файл winnt.h). В этом заголовке нас интересует только одно поле:

IMAGE\_DOS\_HEADER->e\_lfanew,

содержащее смещение сигнатуры файла.

 2. Сигнатура PE-файла Сигнатура, иначе - подпись, означающая, что этот файл является исполнительным файлом для WIndows. Представляет собой строку

PE\0\0

и располагается в файле по смещению, указанному в IMAGE\_DOS\_HEADER->e\_lfanew.

 3. Заголовок IMAGE\_NT\_HEADERS Находится сразу же за сигнатурой PE, и представляет собой структуру из двух заголовков:

IMAGE\_FILE\_HEADER IMAGE\_OPTIONAL\_HEADER

 Данные именно из этих структур определяют, как будет выглядеть изображение файла в памяти. В самом конце структуры IMAGE\_OPTIONAL\_HEADER располагается массив записей, имеющий тип IMAGE\_DATA\_DIRECTORY и называемый DataDirectory. Начальные элементы массива содержат стартовый RVA\* и размеры важных частей исполняемого файла. В настоящее время некоторые элементы в конце массива в стандарте не используются. Первый элемент массива - это всегда адрес и размер экспортированной таблицы функций (если она присутствует). Второй элемент массива - адрес и размер импортированной таблицы функций (она то нас и интересует).

 \* RVA - Relative virtuall address - относительный виртуальный адрес. RVA - это просто смещение данного элемента по отношению к адресу, с которого начинается отображение файла в памяти. Пусть, к примеру, загрузчик Windows отобразил РЕ-файл в память, начиная с адреса 0х400000 в виртуальном адресном пространстве. Если некая таблица в отображении начинается с адреса 0х401464, то RVA данной таблицы 0х1464.

 Вот практически все, что нам нужно знать о формате файла! Резюмируя , выпишем логическую цепочку доступа к таблице импорте:

IMAGE\_DOS\_HEADER->e\_lfanew -> IMAGE\_NT\_HEADERS-> IMAGE\_OPTIONAL\_HEADER-> DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT]

 Теперь рассмотрим структуру самой таблицы импорта. Данная таблица находится в секции импорта (что такое секции в .exe-фале - см описание формата) и содержит всю информацию, необходимую загрузчику для подключения всех DLL и определения адресов вызываемых программой функций. Однозначно определить расположение таблицы импорта по имени секции не представляется возможным, т.к. в зависимости от компилятора данные имена значительно отличаются. Например, компилятор Borland C++ 5.5 (bcc32.exe) именует секцию импорта как ".idata"; компилятор от Microsoft хранит таблицу импорта в секции с именем ".text".\*

 \* Строго говоря, компилятор совсем не обязан выделять данные об импортируемых функциях в отдельную секцию. Например, компилятор от Microsoft хранит данные импорта в одной секции с исполняемым кодом.

 Рассмотрим подробнее, что представляет собой таблица импорта. Как и следовало ожидать, это массив записей определенного типа (IMAGE\_IMPORT\_DESCRIPTOR). Количество записей в массиве нигде в заголовках файла не хранится, а признаком конца массива является запись со всеми полями, установленными в NULL. Каждой используемой программой DLL библиотеке соответствует одна запись в таблице импорта следующего вида:

рис. 2 Структура таблицы импорта.

 Поля структуры IMAGE\_IMPORT\_DESCRIPTOR: 1. DWORD Characteristics Указатель на таблицу указателей (HintName Array) типа PIMAGE\_IMPORT\_BY\_NAME. В данной таблице содержатся адреса структур, cодержащих информацию об импортируемых из данной библиотеки функциях. Формат структуры описан в winnt.h как IMAGE\_IMPORT\_BY\_NAME, и включает в себя 2 поля: Hint - число, помогающее загрузчику найти адрес функции в самой библиотеке. Используется опционально, и не обязательно должно содержать верное значение.

Второй параметр - Name - строка, содержащая имя функции. 2. DWORD TimeDateStamp Дата и время создания файла. 3. DWORD Forwarder Chain Данное поле служит для реализации механизма "ссылочности" между DLL библиотеками. Например, одна библиотека часть своих экспортируемых функций представляет как экспортируемые функции другой библиотеки. К примеру, NTDLL.DLL перенаправляет вызовы частьи своих функций в KERNEL32.DLL. К сожалению, механизм передачи вызовов такого рода не описан, а программы, реализующие эту возможность, достаточно сложно найти. 3. DWORD Name Это RVA указатель на нуль-терминированную ASCII строку, содержащую имя файла DLL библиотеки. 4. PIMAGE\_THUNK\_DATA FirstThunk Указатель на вторую таблицу указателей, идентичную таблице HintName Array. До начала загрузки DLL библиотек в адресное пространство процесса эти указатели содержат адреса структур IMAGE\_IMPORT\_BY\_NAME. В процессе загрузки программы этот массив заполняется адресами соответствующих функций.

 Это практически всё, что нам нужно знать о формате .exe файла для решения поставленной задачи.

 После таблицы импорта (последняя запись, содержащая во всех полях NULL) в файле как правило идут данные импорта (массивы HintName, строки, с именами библиотек и т.д.). Следовательно, добавление еще одной записи о нашей DLL в существующую таблицу представляется достаточно громоздким и труднореализуемым - потому как в случае "сдвига" данных, идущих после таблицы импорта необходимо будет пересчитать все указатели на эти данные. Выход из данной ситуации - создание новой таблицы импорта, содержащей всю существующую информацию + информация о нашей DLL. Чтобы где-то расположить новую таблицу импорта, необходимо наличие свободного места в файле. И тут как нельзя кстати обнаруживается такая замечательная особенность PE-файлов, как "страничность", или, иначе говоря, выравнивание данных в файле на определенные адреса. Размеры этих страниц определяются параметром

IMAGE\_OPTIONAL\_HEADER->FileAlignment.

 Рассмотрим, как используется этот параметр. Допустим, что его величина равна 0x200h (стандартное значение), а исходный размер данных для какой-либо секции равен 0x500h (до создания компилятором файла). Так вот, после выравнивания на величину FileAlignment размер секции будет равен 0x600h, т.е. будет кратен величине FileAlignment. И хотя последние 0х100h байт нулей добавленные компилятором в секцию нигде не используются, они исправно отображаются на адресное пространство процесса. Соответственно все, что мы туда запишем, будет присутствовать в образе файла в памяти. Размер свободного места в секции зависит от конкретного файла, но, как правило, его достаточно для размещения новой таблицы импорта. В том случае, если ни в одной из секций файла нет свободного пространства, возможен вариант создания дополнительной секции либо увеличения длины самой последней из секций (в приводимой для примера программе не реализовано). Резюмируя, разбиваем процесс внедрения на следующие стадии:

 Открытие файла

Анализ файла на возможность модификации (проверка наличия свободного места в файле для новой таблицы импорта)

Формирование новой таблицы импорта, соответствующих массивов HintName (ссылки FirstThunk и Characteristics), имени библиотеки (ссылка Name).

Расчёт RVA всех структур, заносимых нами в файл (обоих массивов HintName, строки с именем DLL, нового адреса таблицы импорта).

Запись в файл новой таблицы импорта и всех новых структур.

Установка нового указателя на таблицу импорта в заголовке (второй элемент массива в IMAGE\_OPTIONAL\_HEADER->DataDirectory[]).

 В том случае, если планируется добавление новой секции либо увеличение размера существующей, необходима дополнительная модификация таблицы секций.

**Реализация**

Думаю, теории достаточно, приступаем к практике. Целиком проект можете загрузить [тут](http://bugtraq.ru/library/programming/.keep/harddll.zip). Разберем пошагово каждую операцию: 1. Открываем .exe файл, отображаем его для удобства работы на своё адресное пространство.

 IMAGE\_DOS\_HEADER \*mz\_head;

 IMAGE\_FILE\_HEADER \*pe\_head;

 IMAGE\_OPTIONAL\_HEADER \*pe\_opt\_head;

 IMAGE\_SECTION\_HEADER \*sect;

 char pe[] = "PE\0\0";

 HANDLE f = NULL;

 //Открываем файл

 f = CreateFile(openF->FileName.c\_str(), GENERIC\_READ | GENERIC\_WRITE,

 FILE\_SHARE\_WRITE, NULL,

 OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

 if (f == INVALID\_HANDLE\_VALUE)

 {

 Log->Lines->Add("Ошибка при открытии файла: ");

 parse\_error();

 return;

 }

 //Создаем отображение файла

 HANDLE fMap = CreateFileMapping( f, NULL,

 PAGE\_READWRITE,

 0, 0, NULL);

 CloseHandle(f);

 if (fMap == NULL)

 {

 Log->Lines->Add("Ошибка при вызове CreateFileMapping(): ");

 parse\_error();

 return;

 }

 int size = sizeof( IMAGE\_DOS\_HEADER );

 //Отображаем начало файла в память

 LPVOID fBeg = MapViewOfFile( fMap, FILE\_MAP\_WRITE, 0, 0, size);

 if (fBeg == NULL)

 {

 Log->Lines->Add("Ошибка при вызове MapViewOfFile(): ");

 parse\_error();

 return;

 }

 2. Проверяем, является ли файл PE-executable:

 //Определяем смещение РЕ-заголовка.

 mz\_head = (IMAGE\_DOS\_HEADER \*)fBeg;

 DWORD peOffset = mz\_head->e\_lfanew;

 UnmapViewOfFile(fBeg);

 //Отображаем в память с учетом смещения до РЕ-заголовка

 size = peOffset + sizeof( DWORD ) + sizeof( IMAGE\_FILE\_HEADER )

 + sizeof( IMAGE\_OPTIONAL\_HEADER );

 fBeg = MapViewOfFile( fMap, FILE\_MAP\_READ, 0, 0, size);

 if (fBeg == NULL)

 {

 Log->Lines->Add("Ошибка при вызове MapViewOfFile(): ");

 parse\_error();

 CloseHandle(fMap);return;

 }

 mz\_head = (IMAGE\_DOS\_HEADER \*)fBeg;

 (DWORD)pe\_head = (DWORD)fBeg + peOffset;

 //Проверяем, PE или не PE файл

 if ( strcmp(pe,(const char \*)pe\_head) != 0)

 {

 Log->Lines->Add("Этот файл не является Portable Executable - файлом.");

 UnmapViewOfFile(fBeg);CloseHandle(fMap);

 return;

 }

 UnmapViewOfFile(fBeg);

 //По новой отображаем файл в память полностью

 fBeg = MapViewOfFile( fMap, FILE\_MAP\_WRITE, 0, 0, 0);

 if (fBeg == NULL)

 {

 Log->Lines->Add("Ошибка при вызове MapViewOfFile(): ");

 parse\_error();

 CloseHandle(fMap); return;

 }

 3. Определяем расположение таблицы импорта, выводим информацию об используемых DLL.

 mz\_head = (IMAGE\_DOS\_HEADER \*)fBeg;

 (DWORD)pe\_head = (DWORD)fBeg + peOffset + sizeof(DWORD);

 (DWORD)pe\_opt\_head = (DWORD)pe\_head + sizeof(IMAGE\_FILE\_HEADER);

 //Определяем расположение таблицы импорта в секции импорта...

 DWORD ImportRVA = pe\_opt\_head->

 DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress;

 int sect\_num = -1;

 //Ищем секцию с таблицей импорта...

 (DWORD)sect = (DWORD)pe\_opt\_head + sizeof(IMAGE\_OPTIONAL\_HEADER);

 int i;

 for ( i=0; iNumberOfSections; i++)

 {

 if ( ImportRVA < sect->VirtualAddress )

 {

 sect--;

 sect\_num=i-1;

 break;

 }

 sect++;

 }

 if (sect\_num == -1)

 {

 Log->Lines->Add("Данная программа не использует динамические библиотеки!");

 UnmapViewOfFile(fBeg);CloseHandle(fMap);

 return;

 }

 sect++;

 DWORD AfterImportSecBeg = (DWORD)fBeg + sect->PointerToRawData;

 sect--;

 //Получаем файловый указатель на раздел c таблицей импорта.

 LPVOID ImportSecBeg;

 (DWORD)ImportSecBeg = (DWORD)fBeg + sect->PointerToRawData;

 //Вычисляем смещение таблицы импорта в секции

 //импорта относительно ее начала (секции).

 LPVOID ImportTable;

 (DWORD)ImportTable = ImportRVA - sect->VirtualAddress;

 (DWORD)ImportTable = (DWORD)ImportSecBeg

 + (DWORD)ImportTable;

 IMAGE\_IMPORT\_DESCRIPTOR \*DLLInfo = (IMAGE\_IMPORT\_DESCRIPTOR \*)ImportTable;

 LPVOID DLLName;

 DWORD DLLCounter = 0;

 //Выводим информацию об используемых DLL

 while (DLLInfo->Name != NULL)

 {

 DLLCounter++;

 (DWORD)DLLName = (DWORD)DLLInfo->Name - sect->VirtualAddress;

 (DWORD)DLLName = (DWORD)ImportSecBeg + (DWORD)DLLName;

 Log->Lines->Add(IntToStr(DLLCounter)+"->"+(LPSTR)DLLName);

 Application->ProcessMessages();

 DLLInfo++;

 }

 Log->Lines->Add("Всего используется "+IntToStr(DLLCounter) + " библиотек.");

 4. Определяем, имеется ли в файле достаточно свободного места для размещения новой таблицы импорта.

 //Вычисляем размер новой таблицы импорта:

 //Суммируем количество уже используемых DLL + наша DLL + zero запись.

 DWORD NewImportTableSize = sizeof(IMAGE\_IMPORT\_DESCRIPTOR)\*(DLLCounter+2);

 char dllName[] = "azx";

 NewImportTableSize += strlen(dllName)+1;

 //Получаем файловый указатель на конец секции импорта.

 LPVOID pos;

 (DWORD)pos = AfterImportSecBeg-1;

 DWORD maxFree = 0;

 DWORD prevPtr;

 LPVOID FreePtr = NULL;

 //Ищем максимальный кусок свободного места в секции...

 while ( pos >= ImportSecBeg )

 {

 if ( \*(BYTE \*)pos == 0x00 )

 {

 prevPtr = (DWORD)pos;

 while (\*(BYTE \*)pos == 0x00)

 (DWORD)pos -= 1;

 if ( ((DWORD)prevPtr - (DWORD)pos) > maxFree )

 {

 maxFree = ((DWORD)prevPtr - (DWORD)pos);

 (DWORD)FreePtr = (DWORD)pos + 1;

 }

 }

 (DWORD)pos -= 1;

 }

 //Модифицируем полученный указатель на свободный блок, т.к.

 //он может указывать на завершающий нулевой DWORD

 //какой-либо структуры

 (LPDWORD)FreePtr +=1;

 maxFree -=4;

 //Проверяем объем свободного места

 if ( maxFree < NewImportTableSize )

 {

 Log->Lines->Add("Недостаточно свободного места в таблице импорта \

 для занесения информации об дополнительной библиотеке.");

 UnmapViewOfFile(fBeg);

 CloseHandle(fMap);

 return;

 }

 else

 Log->Lines->Add("Достаточно свободного \

 места для занесения дополнительной информации.");

 Application->ProcessMessages();

 5. Финальная часть: Создаем в файле новую таблицу импорта и структуру IMAGE\_IMPORT\_BY\_NAME. Записываем в файл строки с именем нашей библиотеки и импортируемой функции. Вычисляем все необходимые адреса, заносим их в структуры IMAGE\_IMPORT\_DESCRIPTOR, IMAGE\_IMPORT\_BY\_NAME. Заносим в заголовок новый адрес таблицы импорта.

 //1. Копируем старую таблицу импорта в новое место

 memcpy(FreePtr, ImportTable, sizeof(IMAGE\_IMPORT\_DESCRIPTOR)\*DLLCounter);

 //2.1 Сохраняем строку с именем нашей DLL в старой таблице импорта

 //(для экономии места)

 memcpy(ImportTable, OUR\_DLL\_NAME, strlen(OUR\_DLL\_NAME));

 LPDWORD zeroPtr;

 (DWORD)zeroPtr = (DWORD)ImportTable + strlen(OUR\_DLL\_NAME);

 //2.2 Сохраняем структуру IMAGE\_IMPORT\_BY\_NAME в старой таблице импорта.

 //(так же для экономии места)

 IMAGE\_IMPORT\_BY\_NAME myName;

 myName.Hint = 0x00;

 myName.Name[0] = 0x00;

 WORD Hint = 0;

 char myFuncName[] = OUR\_FUNC\_NAME;

 hackRec patch;

 patch.ZeroDword = NULL;

 patch.IAT = ImportRVA + strlen(OUR\_DLL\_NAME) + sizeof(hackRec);

 patch.IATEnd = NULL;

 DWORD IIBN\_Table;

 memcpy(zeroPtr, &patch, sizeof(patch)); (DWORD)zeroPtr += sizeof(patch);

 memcpy(zeroPtr, &Hint, sizeof(WORD)); (DWORD)zeroPtr += sizeof(WORD);

 memcpy(zeroPtr, myFuncName, strlen(myFuncName)+1 );

 (DWORD)zeroPtr += strlen(myFuncName)+1;

 memcpy(zeroPtr, &myName, sizeof(IMAGE\_IMPORT\_BY\_NAME) );

 //2.3. Заполняем структуру IMAGE\_IMPORT\_DESCRIPTOR данными об нашей DLL

 IMAGE\_IMPORT\_DESCRIPTOR myDLL;

 //Вычисляем указатель на нашу структуру IMAGE\_IMPORT\_BY\_NAME:

 //это адрес начала старой таблицы импорта + длинна строки с именем

 //нашей DLL + нулевой DWORD

 IIBN\_Table = ImportRVA + strlen( OUR\_DLL\_NAME ) + sizeof(DWORD);

 //Указатель на таблицу Characteristics

 myDLL.Characteristics = IIBN\_Table;

 myDLL.TimeDateStamp = NULL;

 myDLL.ForwarderChain = NULL;

 //Записываем адрес строки с именем файла нашей DLL

 myDLL.Name = ImportRVA;

 //Указатель на таблицу FirstThunk

 myDLL.FirstThunk = IIBN\_Table;

 //Записываем в новую таблицу импорта запись о нашей DLL

 LPVOID OldFreePtr = FreePtr;

 (DWORD)FreePtr +=sizeof(IMAGE\_IMPORT\_DESCRIPTOR)\*DLLCounter;

 memcpy(FreePtr, &myDLL, sizeof(IMAGE\_IMPORT\_DESCRIPTOR));

 //Создаем "финальную" нулевую запись со всеми полями равными нулю

 myDLL.Characteristics = NULL;

 myDLL.TimeDateStamp = NULL;

 myDLL.ForwarderChain = NULL;

 myDLL.Name = NULL;

 myDLL.FirstThunk = NULL;

 //И записываем её в конец новой таблицы импорта.

 (DWORD)FreePtr +=sizeof(IMAGE\_IMPORT\_DESCRIPTOR)\*DLLCounter;

 memcpy(FreePtr, &myDLL, sizeof(IMAGE\_IMPORT\_DESCRIPTOR));

 //3. Устанавливаем указатель на нашу таблицу импорта.

 // Вычисляем RVA нашей таблицы

 DWORD NewImportTableRVA = (DWORD)OldFreePtr - (DWORD)ImportSecBeg +

 sect->VirtualAddress;

 // Заносим его в DataDirectory

 pe\_opt\_head->DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress =

 NewImportTableRVA;

 pe\_opt\_head->DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].Size =

 (DLLCounter + 1) \* sizeof(IMAGE\_IMPORT\_DESCRIPTOR);

 UnmapViewOfFile(fBeg);

 CloseHandle(fMap);

**Вывод**

Данная методика позволяет внедрять свою DLL библиотеку в программы, имеющие достаточно свободного места в секции с таблицей импорта. Приведенная программа может быть доработана в следующих направлениях:

Создание таблицы импорта в другой секции (если в секции с оригинальной таблицей не хватает места)

Создание новой секции и хранение новой таблицы импорта в ней.

 Отдельное слово стоит сказать об .exe файлах, входящих в стандартную поставку Windows(таких как calc.exe, paint.exe, wordpad.exe, etc.). У них таблица импорта продублирована в начале файла, между MZ- и PE- заголовками, поэтому при модификации таких файлов необходимо в соответсвующих записях в DataDirectory обнулить адреса на эти таблицы (подробнее см. файл winnt.h, раздел Directory Entries).