**Методы перехвата API-вызовов в Win32**

Игорь В. Филимонов

**Введение**

Данная статья написана в результате анализа известных методов перехвата API-вызовов в Windows. В некоторых широко известных примерах реализации перехвата системных функций есть небольшие ошибки, которые в некоторых случаях приводят к тому, что перехват не работает. Один из таких примеров был описан в RSDN Magazine #1, другой – в известной книге Джеффри Рихтера «Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows», 4-е издание.

Перехват системных функций операционной системы – приём, известный давно. Обычно перехватывается некоторая системная функция с целью мониторинга или изменения её поведения. Во времена DOS программисты перехватывали программные прерывания (int 21h, int 16h, int 10h). С приходом Win16 понадобились средства для перехвата API-функций. И, наконец, с появлением Win32 средства перехвата ещё раз эволюционировали, подстроившись под новую систему. Операционные системы семейства Windows никогда не содержали встроенных средств, специально предназначенных для перехвата системных функций. И понятно почему – всё-таки это немного хакерский приём. Поэтому перехват обычно осуществляется «подручными средствами», и для его реализации нужно чётко представлять многие глубинные аспекты устройства и функционирования операционной системы.

В данной статье рассматриваются методы реализации перехвата системных API-функций в 32-разрядных операционных системах Windows. Рассматриваются особенности реализации перехвата в Win9X (Windows 95/98/98SE/ME) и WinNT (Windows NT/2000/XP/2003).

**Особенности организации памяти в Windows**

Так как перехват практически всегда связан с модификацией памяти (либо кода перехватываемой функции, либо таблиц импорта/экспорта), то для его осуществления необходимо учитывать особенности архитектуры памяти WinNT и Win9X.

Каждому процессу (начиная с Windows 95) выделяется собственное виртуальное адресное пространство. Для 32-разрядных процессов его размер составляет 4 Гб. Это адресное пространство разбивается на разделы, функциональное назначение и свойства которых довольно сильно отличаются у семейств ОС WinNT и Win9Х.

Адресное пространство любого процесса в Win9Х можно разделить на три раздела:

Младшие два гигабайта (00400000-7FFFFFFF) – код и данные пользовательского режима (в диапазоне 00000000-003FFFFF расположены разделы для выявления нулевых указателей и для совместимости с программами DOS и Win16);

Третий гигабайт – для общих файлов, проецируемых в память (MMF), и системных DLL.

Четвёртый гигабайт – для кода и данных режима ядра (здесь располагается ядро операционной системы и драйверы).

Старшие два гигабайта являются общими для всех процессов. Основные системные DLL – kernel32.dll, advAPI32.dll, user32.dll и GDI32.dll загружаются в третий гигабайт. По этой причине эти четыре библиотеки доступны всем процессам в системе. Поскольку этот гигабайт общий, они существуют во всех процессах по одним и тем же адресам. Из соображений безопасности Microsoft запретила запись в область, куда они загружаются. Если же запись туда всё же произвести (а это возможно из режима ядра или недокументированными методами), то изменения произойдут во всех процессах одновременно.

В WinNT общих разделов у процессов нет, хотя системные библиотеки по-прежнему во всех процессах загружаются по одинаковым адресам (но теперь уже в область кода и данных пользовательского режима). Запись в эту область разрешена, но у образов системных библиотек в памяти стоит атрибут «копирование при записи» (copy-on-write). По этой причине попытка записи, например, в образ kernel32.dll приведёт к появлению у процесса своей копии изменённой страницы kernel32.dll, а на остальных процессах это никак не отразится.

Все эти различия существенно влияют на способы реализации перехвата функций, расположенных в системных DLL.

Перехваты можно разделить на два типа: локальные (перехват в пределах одного процесса) и глобальные (в масштабах всей системы).

**Локальный перехват**

Локальный перехват с использованием раздела импорта

Локальный перехват может быть реализован и в Win9X, и в WinNT посредством подмены адреса перехватываемой функции в таблице импорта. Для понимания механизма работы этого метода нужно иметь представление о том, как осуществляется динамическое связывание. В частности, необходимо разбираться в структуре раздела импорта модуля.

В разделе импорта каждого exe- или DLL-модуля содержится список всех используемых DLL. Кроме того, в нем перечислены все импортируемые функции. Вызывая импортируемую функцию, поток получает ее адрес фактически из раздела импорта. Поэтому, чтобы перехватить определенную функцию, надо лишь изменить её адрес в разделе импорта. Для того чтобы перехватить произвольную функцию в некотором процессе, необходимо поправить её адрес импорта во всех модулях процесса (так как процесс может вызывать эту функцию не только из exe-модуля, но и из DLL-модулей). Кроме того, процесс может воспользоваться для загрузки DLL функциями LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW или, если она уже загружена, определить её адрес при помощи функции GetProcAddress. Поэтому для перехвата любой API-функции необходимо перехватывать и все эти функции.

Существует несколько широко известных примеров реализации этого метода, в частности один из них описан в книге Джеффри Рихтера «Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows» (Jeffrey Richter «Programming Applications for Microsoft Windows»), 4-е издание. Другой пример – библиотека APIHijack, написанная Wade Brainerd на основе DelayLoadProfileDLL.CPP (Matt Pietrek, MSJ, февраль 2000). Для описания этого метода я взял за основу пример Джеффри Рихтера (с небольшими изменениями).

Для реализации перехвата был создан класс CAPIHook, конструктор которого перехватывает заданную функцию в текущем процессе. Для этого он вызывает метод ReplaceIATEntryInAllMods, который, перечисляя все модули текущего процесса, вызывает для каждого метод ReplaceIATEntryInOneMod, в котором и реализуется поиск и замена адреса в таблице импорта для заданного модуля.

|  |
| --- |
| void CAPIHook::ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,  PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) { //Получим адрес секции импорта ULONG ulSize; PIMAGE\_IMPORT\_DESCRIPTOR pImportDesc =  (PIMAGE\_IMPORT\_DESCRIPTOR)ImageDirectoryEntryToData(hmodCaller, TRUE, IMAGE\_DIRECTORY\_ENTRY\_IMPORT, &ulSize); if (pImportDesc == NULL)  return; //Здесь её нет //Найдём нужный модуль for (; pImportDesc->Name; pImportDesc++)  { PSTR pszModName = (PSTR)((PBYTE) hmodCaller + pImportDesc->Name); if (lstrcmpiA(pszModName, pszCalleeModName) == 0)  { //Нашли if (pImportDesc->Name == 0) return; //Ни одна функция не импортируется //Получим адрес таблицы импорта PIMAGE\_THUNK\_DATA pThunk =  (PIMAGE\_THUNK\_DATA)((PBYTE) hmodCaller + pImportDesc->FirstThunk); //Переберём все импортируемые функции for (; pThunk->u1.Function; pThunk++)  { PROC\* ppfn = (PROC\*) &pThunk->u1.Function; //Получим адрес функции BOOL fFound = (\*ppfn == pfnCurrent); //Его ищем? if (!fFound && (\*ppfn > sm\_pvMaxAppAddr))  { // Если не нашли, то поищем поглубже.  // Если мы в Win98 под отладчиком, то // здесь может быть push с адресом нашей функции PBYTE pbInFunc = (PBYTE) \*ppfn; if (pbInFunc[0] == cPushOpCode)  { //Да, здесь PUSH ppfn = (PROC\*) &pbInFunc[1]; //Наш адрес? fFound = (\*ppfn == pfnCurrent); } } if (fFound)  {  //Нашли!!! DWORD dwDummy; //Разрешим запись в эту страницу VirtualProtect(ppfn, sizeof(ppfn), PAGE\_EXECUTE\_READWRITE, &dwDummy); //Сменим адрес на свой WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,  sizeof(pfnNew), NULL); //Восстановим атрибуты VirtualProtect(ppfn, sizeof(ppfn), dwDummy , &dwDummy); //Готово!!! return; } } } } //Здесь этой функции не нашлось} |

При помощи функции ImageDirectoryEntryToData определяется дескриптор таблицы импорта и, если он есть, перебираются все DLL, из которых импортируются функции. Если DLL находится, то среди функций, импортируемых из неё, ищется нужная, а затем при помощи WriteProcessMemory её адрес меняется на адрес своего обработчика. Теперь он будет вызываться каждый раз, когда из данного модуля будет происходить обращение к перехваченной функции.

|  |
| --- |
| ПРИМЕЧАНИЕЕсли вы читали уже упоминаемую выше книгу Джеффри Рихтера, то могли заметить, что в функции ReplaceIATEntryInOneMod я сделал одно изменение. У него она работала так: в таблице импорта находился список функций того модуля, функция из которого импортировалась, и если в этом списке эта функция не находилась, то ReplaceIATEntryInOneMod больше ничего не делала (т. е. перехват не происходил). Я столкнулся с таким поведением, когда написал тестовую программу на Delphi для примера DriveType2 (этот пример описан ниже, в разделе «Глобальный перехват методом тотального локального перехвата», он перехватывает функцию GetDriveTypeA во всех приложениях с использованием описываемого метода). Тест, написанный на Visual C++, работал прекрасно – функция GetDriveTypeA перехватывалась. А вот программа на Delphi всё равно для всех перехватываемых мной дисков возвращала реальные значения. Я посмотрел таблицу импорта тестовой программы при помощи утилиты DUMPBIN и обнаружил, что компилятор Delphi не поместил все импортируемые функции из kernel32.dll в один список, а разбил их на 3 части, причём GetDriveTypeA оказалась в третьей. Поэтому функция ReplaceIATEntryInOneMod Джеффри Рихтера, просмотрев все функции из первого списка Kernel32.dll, не нашла функции GetDriveTypeA, хотя она и импортировалась модулем DriveTypeTest.exe. Я исправил эту функцию таким образом, чтобы она проверяла всю таблицу импорта и перебирала все списки с функциями из kernel32.dll (как оказалось, их может быть несколько). В описании формата РЕ-файла нигде не оговаривается, что каждый модуль, из которого импортируются функции, должен встречаться в секции импорта только один раз, и, видимо, некоторые компиляторы этим пользуются. |

При реализации данного метода следует учитывать, что вызовы из DllMain библиотеки, в которой находится перехватываемая функция, перехватить не удастся. Это связано с тем, что перехват может быть осуществлён только по окончании выполнения LoadLibrary, а к этому времени DllMain уже будет вызвана. Конечно, можно написать свой вариант LoadLibrary (примеры загрузки DLL «вручную» существуют) и осуществлять перехват между загрузкой DLL и вызовом DllMain, но это сильно усложняет задачу.

Основным достоинством данного метода является то, что он одинаково реализуется как в Win9X, так и в WinNT.

|  |
| --- |
| ПРИМЕЧАНИЕВ Windows NT функции Module32First и Module32Next не реализованы, и для перечисления модулей процесса вместо них придётся воспользоваться функциями из PSAPI.dll. |

Локальный перехват посредством изменения перехватываемой функции (только WinNT)

Данный метод перехвата основан на следующем: первые несколько байт перехватываемой функции заменяются на команду безусловного перехода к функции перехвата. Этот трюк достаточно просто реализуется в WinNT (как я уже упоминал, в WinNT для каждого процесса создается своя копия образов системных библиотек), но практически нереализуем в Win9X (так как в Win9X если и можно внести изменения в образ системной библиотеки, то только в адресных пространствах всех процессов сразу).

Существует множество примеров реализации этого метода. Я рассмотрю метод, предлагаемый Microsoft – Detours library.

Detours – это первая официальная библиотека, предназначенная для перехвата функций (не только системных, но и любых других). К основным понятиям Detours относятся:

целевая функция (target function) – функция, перехват которой осуществляется;

функция-перехватчик (detour function) – функция, замещающая перехватываемую;

функция-трамплин (trampoline function) – функция, состоящая из заголовка целевой функции и команды перехода к остальному коду целевой функции.

|  |
| --- |
| ПРИМЕЧАНИЕTrampoline в переводе с английского – «батут», однако словосочетание «функция-трамплин» более точно передаёт логику её работы. |

Таким образом, если целевая функция имеет следующий заголовок:

|  |
| --- |
| TargetFunction: push ebp mov ebp, esp push ebx push esi push edi ... |

то в результате перехвата получится следующее:

|  |
| --- |
| TargetFunction: jmp DetourFunction:TargetFunction+5: push edi ... TrampolineFunction: push ebp mov ebp, esp push ebx push esi jmp TargetFunction+5 ... |

Причём функция-перехватчик может вызывать функцию-трамплин в качестве оригинальной целевой функции.

Библиотека Detours предлагает два метода внедрения «трамплинов» – статический и динамический. Статический метод используется, когда адрес целевой функции известен на этапе сборки модуля. Реализуется он так:

|  |
| --- |
| #include <windows.h>#include <detours.h> //Подключим библиотеку Detours//Этот макрос создаёт функцию-трамплин для функции SleepDETOUR\_TRAMPOLINE(VOID WINAPI SleepTrampoline(DWORD), Sleep); VOID WINAPI SleepDetour(DWORD dw) //Это – функция-перехватчик{ //В этом примере она ничего не делает, просто вызывает оригинальную функцию return SleepTrampoline(dw);}void main(void){ //Здесь осуществляется перехват DetourFunctionWithTrampoline((PBYTE)SleepTrampoline, (PBYTE)SleepDetour); //... //А здесь снимается DetourRemoveTrampoline(SleepTrampoline); } |

Динамический перехват используется в случаях, когда целевая функция на этапе сборки недоступна. Реализуется он так:

|  |
| --- |
| #include <windows.h>#include <detours.h> //Подключим библиотеку DetoursVOID (\*DynamicTrampoline)(VOID) = NULL; //Это будет функция-трамплинVOID DynamicDetour(VOID) //Это – функция-перехватчик{ //В этом примере она ничего не делает, просто вызывает оригинальную функцию return DynamicTrampoline();}void main(void){ //Получим адрес целевой функции VOID (\*DynamicTarget)(VOID) = SomeFunction; //Здесь осуществляется перехват DynamicTrampoline=(FUNCPTR)DetourFunction((PBYTE)DynamicTarget, (PBYTE)DynamicDetour); //... DetourRemoveTrampoline(DynamicTrampoline); //А здесь снимается} |

При перехвате функция DetourFunction динамически создаёт трамплин и возвращает его адрес. В качестве функции SomeFunction, которая в данном примере возвращает адрес целевой функции, можно использовать DetourFindFunction, которая пытается найти нужную функцию в нужном модуле. Сначала она пытается сделать это через LoadLibrary и GetProcAddress, а в случае неудачи – использует библиотеку ImageHlp для поиска отладочных символов.

Макрос DETOUR\_TRAMPOLINE и функция DetourFunction включают в себя встроенный табличный дизассемблер, который определяет, какое количество байт из заголовка целевой функции должно быть скопировано в функцию-трамплин (не менее 5 байт (размер команды jmp), составляющих целое число команд процессора). Если целевая функция занимает менее 5 байт, то перехват оканчивается неудачей.

К достоинствам данного метода следует отнести простоту и надёжность. В отличие от метода с использованием раздела импорта, не нужно учитывать все возможные методы, которыми может быть получен реальный адрес функции. Недостаток – не удастся перехватить функцию с размером менее 5 байт или функцию со следующим заголовком:

|  |
| --- |
|  push ecx ;в функцию передаётся количество итераций циклаbegin\_loop: ;... ;здесь какой-то код ;... loop begin\_loop ret |

Вышеприведённый пример Galen Hunt, один из авторов Detours, прокомментировал следующим образом: «Существует множество теоретических примеров кода, где пролог функции меньше 5 байт, требуемых для команды jmp. Однако никто не сообщал о реальных примерах функции с такими проблемами».

|  |
| --- |
| ПРЕДУПРЕЖДЕНИЕНа момент установки/снятия перехвата нужно останавливать все остальные потоки процесса, в котором происходит перехват (или удостовериться, что они не могут вызывать перехватываемую функцию). |

Существует другой способ реализации данного метода. Вместо команды jmp в начало функции помещается команда INT 3, а управление функции-перехватчику передаётся косвенно в обработчике необработанных исключений (её адрес заносится в pExceptionInfo->ContextRecord->Eip и обработчик возвращает EXCEPTION\_CONTINUE\_EXECUTION). Так как команда INT 3 занимает 1 байт, то вышеописанная ситуация в этом случае даже теоретически невозможна.

|  |
| --- |
| ПРЕДУПРЕЖДЕНИЕОсновным недостатком данного способа является его крайне малое быстродействие (обработка исключения в Windows занимает довольно продолжительное время). Кроме того, наличие обработчика исключений в перехватываемом процессе приведёт к тому, что данный метод работать не будет. Также данный способ не будет работать под отладчиком. |

Глобальный перехват

Глобальный перехват может быть реализован различными способами. Первый способ – применение локального перехвата ко всем приложениям в системе (запущенным в момент перехвата или позже). Второй способ – «взлом системы» – подразумевает подмену кода перехватываемой функции непосредственно в DLL-файле или его образе в памяти.

Глобальный перехват методом тотального локального перехвата

Данный метод основан на следующем: если можно перехватить функцию из текущего процесса, то нужно выполнить код перехвата во всех процессах в системе. Существует несколько методов заставить чужой процесс выполнить код перехвата. Самый простой – внести этот код в DllMain некоторой библиотеки, а затем внедрить её в чужой процесс. Методов внедрения DLL также существует несколько (см. Джеффри Рихтер). Самый простой, работающий и в Win9X, и в WinNT – внедрение DLL при помощи ловушек. Реализуется он так: в системе устанавливается ловушка (при помощи функции SetWindowsHookEx) типа WH\_GETMESSAGE (эта ловушка служит для перехвата Windows-сообщений). В этом случае модуль, в котором находится ловушка, автоматически подключается к потоку, указанному в последнем аргументе SetWindowsHookEx (если указан 0, то производится подключение ко всем потокам в системе). Однако подключение происходит не сразу, а перед тем, как в очередь сообщений потока будет послано какое-нибудь сообщение. Поэтому перехват осуществляется не сразу после запуска приложения, а перед обработкой процессом первого сообщения. Так что все вызовы перехватываемой функции до обработки процессом первого сообщения перехватываться не будут. А в приложениях без очереди сообщений (например, консольных) этот способ внедрения вообще не работает.

Я написал пример, реализующий глобальный перехват функции GetDriveTypeA с использованием внедрения DLL при помощи ловушек и перехвата с использованием секции импорта.

Функция GetDriveTypeA из библиотеки kernel32.dll используется программами Windows для определения типа диска (локальный, CD-ROM, сетевой, виртуальный и т. д.). Она имеет следующий прототип:

|  |
| --- |
| UINT GetDriveType(LPCTSTR lpRootPathName); |

lpRootPathName – путь до диска (А:\, В:\ и т.д.)

GetDriveTypeA возвращает одно из следующих значений:

|  |
| --- |
| #define DRIVE\_UNKNOWN 0#define DRIVE\_NO\_ROOT\_DIR 1#define DRIVE\_REMOVABLE 2#define DRIVE\_FIXED 3#define DRIVE\_REMOTE 4#define DRIVE\_CDROM 5#define DRIVE\_RAMDISK 6 |

Перехват этой функции позволяет «обманывать» программы Windows, переопределяя значение, возвращаемое этой функцией, для любого диска.

Программа DriveType2 состоит из двух модулей: DriveType2.exe и DT2lib.dll.

DriveType2.exe реализует интерфейс, а вся работа выполняется в DT2lib.dll.

Проект DT2lib состоит из трёх основных файлов:

APIHook.cpp – этот файл написан Джеффри Рихтером (за исключением некоторых исправлений, сделанных мной. О них я расскажу ниже). В этом файле описан класс CAPIHook, реализующий перехват заданной API-функции во всех модулях текущего процесса. Здесь же автоматически перехватываются функции LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW и GetProcAddress.

Toolhelp.h – этот файл также написан Джеффри Рихтером. В нём описан класс CToolhelp, реализующий обращение к системным toolhelp-функциям. В данном случае он используется классом CAPIHook для перечисления всех модулей, подключенных к процессу.

DT2Lib.cpp – в этом файле я реализовал перехват функции GetDriveTypeA с использованием класса CAPIHook, а также установку ловушки типа WH\_GETMESSAGE, обеспечивающей подключение данного модуля (DT2lib.dll) ко всем потокам в системе.

Как же происходит перехват?

Сразу же после запуска DriveType2.exe вызывается функция DT2\_HookAllApps из DT2lib.dll, которая устанавливает ловушку.

|  |
| --- |
| BOOL WINAPI DT2\_HookAllApps(BOOL fInstall, DWORD dwThreadId) {  BOOL fOk; if (fInstall)  { chASSERT(g\_hhook == NULL); // 2 раза перехватывать ни к чему g\_hhook = SetWindowsHookEx(WH\_GETMESSAGE, GetMsgProc, ModuleFromAddress(DT2\_HookAllApps), dwThreadId); // Установим ловушку fOk = (g\_hhook != NULL); }  else  { chASSERT(g\_hhook != NULL); // Снимать-то нечего fOk = UnhookWindowsHookEx(g\_hhook); // Снимем ловушку g\_hhook = NULL; } return(fOk);} |

Функция ловушки GetMsgProc ничего не делает, а просто вызывает следующую функцию ловушки (возможно, не только наша программа установила ловушку, и это, как минимум нужно проверить). Перед тем, как поместить в очередь, ассоциированную с некоторым потоком, какое-то сообщение, система должна вызвать все установленные ловушки типа WH\_GETMESSAGE (обычно такие ловушки используются для мониторинга или изменения некоторых сообщений, однако мы ничего подобного не делаем – нам нужно просто подключиться ко всем потокам в системе). Система не может просто вызвать нашу функцию ловушки – она и получатель сообщения находятся в разных процессах, а значит, и в разных адресных пространствах. И выход из этой ситуации один – система просто подключает модуль (а это обязательно должен быть DLL-модуль), в котором находится ловушка, к тому процессу, которому посылается сообщение (что нам собственно и нужно).

После подключения DLL к потоку происходит инициализация всех переменных (каждый процесс, к которому подключается DLL, имеет копии всех глобальных и статических переменных, описанных в ней). Из глобальных переменных у нас есть 6 экземпляров класса CAPIHook (для GetDriveTypeA в DT2Lib.cpp и для LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW и GetProcAddress – в APIHook.cpp). Таким образом, при подключении DLL происходит шестикратный вызов конструктора класса CAPIHook, перехватывающего вышеперечисленные функции в текущем (то есть в том, к которому только что произошло подключение) процессе.

При завершении процесса внедрённая DLL отключается. При этом происходит вызов деструктора CAPIHook для всех экземпляров класса.

Данная функция теперь будет вызываться каждый раз, когда из данного модуля будет происходить обращение к GetDriveTypeA.

|  |
| --- |
| int WINAPI Hook\_GetDriveTypeA(PCSTR lpRootPathName){ //Вызовем оригинальную функцию int Result = ((PGetDriveTypeA)(PROC) g\_GetDriveTypeA)(lpRootPathName); if (Result > DRIVE\_NO\_ROOT\_DIR) { int Drive = toupper(\*lpRootPathName);  if (Drive >= 'A' && Drive<='Z') { Drive -= 'A'; //Индекс в массиве Drives //Если этот диск переопределен, вернём значение из массива if (Drives[Drive] < 0xFF) Result = Drives[Drive]; } } return Result;} |

Функция Hook\_GetDriveTypeA сначала вызывает оригинальную GetDriveTypeA. Затем, если возвращаемое значение больше DRIVE\_NO\_ROOT\_DIR (то есть функции был передан корректный аргумент, и она выполнилась без ошибок), то проверяется, переопределен ли диск, тип которого запрашивается. Информация о значениях перехватываемых функций в данном случае хранится в реализованном мной массиве BYTE Drives[26], что позволяет реализовать перехват 26 дисков, от A: до Z:. В этом массиве хранятся значения, возвращаемые функцией GetDriveTypeA для каждого из дисков. Итак, если значение элемента массива, соответствующего аргументу GetDriveTypeA равно 0xFF, то значение возвращается без изменений, в противном случае возвращается значение из массива. Запись значений в этот массив реализуется в DriveType2.cpp.

|  |
| --- |
| СОВЕТЕсли вы хотите, чтобы эта программа полноценно работала в WinNT, следует также перехватить функцию GetDriveTypeW. |

Ещё одна реализация данного метода описана в статье «Перехват API-функций в Windows NT/2000/XP», автор Тихомиров В. А., публиковалась в RSDN Magazine #1 (будьте осторожны, там та же ошибка, что и у Джеффри Рихтера).

|  |
| --- |
| ПРИМЕЧАНИЕУ этого метода есть ещё один существенный недостаток: некоторые коммерческие программы (например, популярный файловый менеджер Total Commander, упакованный ASPack) используют различные системы защиты (ASProtect, VBox и т. д.), шифрующие таблицу импорта защищаемого приложения. С такими программами этот метод не работает. |

Глобальный перехват может быть реализован и с помощью Detours (только в WinNT). А так как методов внедрения DLL известно несколько, то различных вариантов реализации глобального перехвата можно предложить довольно много.

Глобальный перехват методом подмены кода в DLL

Данный метод можно реализовать двумя способами: непосредственной правкой кода DLL, в которой расположена целевая функция, или подменой этой DLL другой, экспортирующей тот же набор функций. Второй способ известен под названием «Подмена с использованием оберток (wrappers)».

Первый способ позволяет реализовывать только сравнительно небольшие по размеру функции-перехватчики, так как код необходимо внедрять в свободные участки DLL – в основном в межсекционное пространство. Другой недостаток – код необходимо писать на ассемблере. Общая идеология работы этого метода та же, что и в Detours. В код целевой функции внедряется команда jmp к функции-перехватчику. Байты, скопированные «из-под» jmp’а, перемещаются в перехватчик (так как перехватчик всё равно пишется на ассемблере, в этом случае его проще сразу совместить с функцией-трамплином). Вот пример реализации этого метода.

В каталоге DriveType0 находится файл kernel32.dll, в котором я сделал следующие исправления (при помощи hiew32.exe):

По адресу 4E02 – локальный адрес .0BFF74E02 (это конец функции GetDriveTypeA) я поместил команду jmp .0BFF71080 – на первое попавшееся свободное место (в исполняемых файлах всегда много свободного места – обычно в концах секций).

По адресу .0BFF71080 (глобальный адрес 1080) я поместил следующий код:

|  |
| --- |
| .BFF71080: 3C03 cmp al,003 ;Возвращаем DRIVE\_FIXED ?.BFF71082: 750E jne .0BFF71092.BFF71084: B402 mov ah,002 ;Да..BFF71086: CD16 int 016 ;/Проверим состояние ScrollLock.BFF71088: 2410 and al,010 ;.BFF7108A: 7404 je .0BFF71090 ;\Светодиод горит ?.BFF7108C: B005 mov al,005 ;Да. Возвращаем DRIVE\_CDROM.BFF7108E: EB02 jmps .0BFF71092 ;На возврат.BFF71090: B003 mov al,003.BFF71092: 5F pop edi ;/Возврат из GetDriveTypeA.BFF71093: 5E pop esi ; (кусок кода, скопированный.BFF71094: 5B pop ebx ; из .0BFF74E02 - .BFF74E06).BFF71095: C9 leave ;.BFF71096: C20400 retn 00004 ;\ |

Таким образом, когда светодиод ScrollLock не горит, функция GetDriveTypeA работает как обычно, а если горит – то для всех Windows-приложений все локальные диски (у меня это С:\ и D:\) превращаются в CD-ROMы.

|  |
| --- |
| ПРИМЕЧАНИЕЧтобы всё это заработало, необходимо заменить файл C:\Windows\System\kernel32.dll на файл DriveType0\kernel32.dll. Сделать это можно, только загрузив компьютер в режиме MS-DOS, так как kernel32.dll – одна из системных библиотек Windows. Данный пример реализован для Windows 98. Поскольку системные библиотеки меняются в зависимости от версии Windows (и даже от номера билда), то в других операционных системах этот пример работать не будет (его нужно реализовывать для каждой версии kernel32.dll заново). |

Этот способ перехвата – один из самых мощных. Однако в коммерческих продуктах его использовать не удастся, так как он, очевидно, нарушает практически любое лицензионное соглашение.

Другой способ реализации этого метода – использование оберток (wrappers). Суть его в создании собственной DLL с тем же набором экспортируемых функций, что и оригинальная. В качестве примера могу привести следующий вариант реализации вышеприведённого примера:

Системную библиотеку Kernel32.dll переименовываем в kernel31.dll :).

Создаём библиотеку с именем Kernel32.dll, в которой реализована одна функция – GetDriveTypeA (это будет функция-перехватчик), а все остальные функции переадресуем к kernel31.dll (благо компилятор Visual C++ поддерживает переадресацию функций DLL).

Полученную библиотеку помещаем в системный каталог.

При этом функция-перехватчик может вызывать оригинальную функцию из kernel31.dll.

Основным недостатком данного способа является то, что он не годится для DLL, экспортирующих переменные.

Глобальный перехват методом подмены кода DLL в памяти (только Win9X)

Идея данного метода заключается в следующем: в Win9X системные DLL загружаются в общую для всех процессов область памяти (в третий гигабайт). Поэтому если бы удалось произвести реализацию Detours под Win9X, то изменения коснулись бы всех процессов (то есть случился бы глобальный перехват). Ситуация осложняется тем, что запись в область системных DLL в Win9X возможна или из режима ядра, или недокументированными средствами. Кроме того, в момент записи необходимо остановить все потоки, которые могут вызывать целевую функцию. Это можно сделать при помощи SuspendThread. Однако эта функция требует в качестве аргумента handle потока, а используемые для перечисления потоков функции Thread32First/Thread32Next возвращают ThreadID. Функция OpenThread, которая позволяет получить handle из ThreadID, реализована только начиная с Windows ME. По этой причине в общем виде документированными средствами из режима пользователя данный метод в Win9X нереализуем. Однако есть другой способ. Обе обозначенные проблемы (запись в область системных DLL и останов всех пользовательских потоков в системе) могут быть решены, если для перехвата использовать драйвер. Драйвер работает в режиме ядра, поэтому из него можно производить запись по любым доступным адресам. А если на момент установки/снятия перехвата поднять IRQL (уровень запроса на прерывание) до DISPATCH\_LEVEL, то прервать поток сможет только процедура обработки аппаратного прерывания (откуда вызывать пользовательские системные функции нельзя). Кроме того, применение драйвера позволяет решить ещё одну проблему. Функция-перехватчик и функция-трамплин должны располагаться в области памяти, общей для всех процессов (в старших 2 гигабайтах). Конечно, можно было бы создать файл, отображаемый в память (MMF – они в Win9X размещаются в третьем гигабайте), и поместить код этих функций туда, или разместить их в отдельной DLL с ImageBase в третьем гигабайте, но проще реализовать их непосредственно в драйвере (драйверы в Win9X размещаются в четвёртом гигабайте – в разделе кода и данных режима ядра).

Рассмотрим пример, реализующий описанный метод. Для осуществления перехвата и размещения функции-трамплина и функции-перехватчика я написал WDM-драйвер (с использованием Visual C++ 6.0, Windows 2000 DDK и Compuware DriverStudio 2.7), а также программу для взаимодействия с ним. Программа и драйвер расположены в каталоге DriveType1 (там же – инструкции по установке).

Пример DriveType1 состоит из двух частей – драйвера DTDrv.sys и установочного скрипта DTDrv.inf, а также программы DriveType.exe.

DriveType.exe компилируется из одного модуля DriveType.cpp, в котором реализованы пользовательский интерфейс и интерфейс с драйвером. Интерфейс с драйвером реализуется через функции CreateFile (открытие драйвера), DeviceIoControl (операции ввода-вывода) и CloseHandle (закрытие драйвера). Реализованы четыре команды, вызываемые через DeviceIoControl – перехват функции GetDriveTypeA, снятие перехвата, установка возвращаемого значения функцией перехвата для каждого из дисков A: .. Z:, чтение текущего состояния перехвата.

Ну а вся работа по перехвату делается в драйвере, за исключением того, что адрес функции GetDriveTypeA определяется также в DriveType.cpp и присылается в качестве параметра в команде перехвата. После получения этого адреса функция DTDRV\_IOCTL\_HOOK\_Handler (из модуля DTDrvDevice.cpp) реализует перехват рассмотренным выше способом. Функция DTDRV\_IOCTL\_UNHOOK\_Handler снимает перехват, функция DTDRV\_IOCTL\_SETDRIVE\_Handler устанавливает значение, возвращаемое перехватчиком, функция DTDRV\_IOCTL\_GETSTATE\_Handler возвращает значения перехвата и флаг перехвата.

Основные переменные, используемые DriveType.cpp:

|  |
| --- |
| unsigned char IsHook = false; //Флаг перехватаunsigned char Drives[26]; //Значения перехватаPUCHAR GDT; //Адрес GetDriveTypeA |

В Drives[26] хранятся значения, возвращаемые функцией MGetDriveType для дисков A: .. Z: (=0xFF, если информация о диске не переопределена).

Итак, функция-трамплин NewGetDriveType будет выглядеть следующим образом:

|  |
| --- |
| \_\_declspec(naked) unsigned int NewGetDriveType(LPSTR Path){ \_asm { nop //Здесь будут первые 5 байт из функции GetDriveTypeA  nop //(в Win98 3 команды ассемблера) nop nop nop jmp $ //А здесь - переход к GetDriveTypeA + 5 }} |

Изначально эта функция «пустая», так как весь её код пишется во время перехвата функцией DTDRV\_IOCTL\_HOOK\_Handler, которая, если оперировать терминами Detours, реализует динамический перехват.

|  |
| --- |
| ПРИМЕЧАНИЕКод этой функции изначально может быть любым, но он должен занимать по крайней мере 10 байт (чтобы уместились 5 байт из заголовка GetDriveTypeA и 5 байт – команда jmp). |

Собственно функция-перехватчик MGetDriveType реализована в моём примере так:

|  |
| --- |
| unsigned int MGetDriveType(LPSTR Path) //Это – функция-перехватчик{ unsigned int res = NewGetDriveType(Path); //Вызовем старый GetDriveTypeA unsigned char Letter = \*Path; if (Letter >= 'a' && Letter <= 'z') Letter- = 'a' - 'A'; //Заглавные if (Letter >= 'A' && Letter <= 'Z')  { unsigned char NewRes = Drives[Letter - 'A']; if (NewRes < 0xFF) res = NewRes; //Если диск переназначен, вернём значение из таблицы } return res;} |

Сначала вызывается функция-трамплин NewGetDriveType, которая фактически выполняет код оригинальной GetDriveTypeA (сначала выполняются первые 5 байт – это 3 команды ассемблера, затем – всё остальное). После этого определяется буква диска. Преобразование буквы в верхний регистр осуществляется вручную. Далее, если данный диск перехвачен, возвращается значение из массива Drives, в противном случае – то, которое вернула NewGetDriveType.

Перехват реализован в функции DTDRV\_IOCTL\_HOOK\_Handler следующим образом:

|  |
| --- |
| NTSTATUS DTDrvDev::DTDRV\_IOCTL\_HOOK\_Handler(KIrp I){ NTSTATUS status = STATUS\_SUCCESS; I.Information() = 0; if (IsHook)  return status; #pragma pack(push, 1) //Включим выравнивание по границе байта struct  { UCHAR Byte0;  ULONG Byte1\_4; } Patch = {0xE9}; //Код инструкции jmp #pragma pack(pop) //Вернём выравнивание по умолчанию KIRQL oldirql; KeRaiseIrql(DISPATCH\_LEVEL, &oldirql); //Поднимем IRQL до DISPATCH\_LEVEL GDT = (PUCHAR)\*(PULONG)I.IoctlBuffer(); //GDT = Адрес GetDriveTypeA RtlCopyMemory(NewGetDriveType, GDT, 5); //Заголовок NewGetDriveType Patch.Byte1\_4 = (ULONG)GDT - (ULONG)NewGetDriveType - 5; RtlCopyMemory((PVOID)((ULONG)NewGetDriveType + 5), &Patch, 5); //jmp GetDriveTypeA + 5 Patch.Byte1\_4 = (ULONG)MGetDriveType - (ULONG)GDT - 5; RtlCopyMemory(GDT, &Patch, 5); //jmp MGetDriveType IsHook = true; KeLowerIrql(oldirql); //Вернём IRQL обратно return status;} |

Если функция GetDriveTypeA ещё не перехвачена (IsHook=false), то:

Определяется адрес функции GetDriveTypeA (он присылается в качестве параметра);

По адресу NewGetDriveType копируются 5 байт из начала GetDriveTypeA;

За ними вставляется байт 0xE9 (код команды jmp) и смещение до точки GetDriveTypeA + 5;

По адресу GetDriveTypeA вставляется 0xE9 и смещение до точки MGetDriveType;

Флаг перехвата IsHook устанавливается в true.

Функция снятия перехвата возвращает всё на свои места:

|  |
| --- |
| NTSTATUS DTDrvDev::DTDRV\_IOCTL\_UNHOOK\_Handler(KIrp I){ NTSTATUS status = STATUS\_SUCCESS; I.Information() = 0; if (!IsHook) return status; KIRQL oldirql; KeRaiseIrql(DISPATCH\_LEVEL, &oldirql); //Поднимем IRQL до DISPATCH\_LEVEL RtlCopyMemory(GDT, NewGetDriveType, 5); //Вернём заголовок GetDriveTypeA на место IsHook = false; KeLowerIrql(oldirql); //Вернём IRQL обратно return status;} |

Данный метод – компромисс между гибкостью (перехват через таблицу импорта не требует написания драйвера), и мощью (по мощи он практически не уступает подмене кода в DLL), однако он реализуется только в Win9X.

**Заключение**

Итак, перехват API-вызовов – вещь, хотя и достаточно сложная, но все-таки реализуемая (причём различными способами). Методы перехвата различны и часто не переносимы из одной версии Windows в другую.

Microsoft старается сохранять совместимость программного обеспечения со старыми версиями Windows, но получается это далеко не всегда, и аспекты программирования, настолько приближённые к низкоуровневому системному программированию, очень сильно различаются для разных версий Windows. Поэтому часто приходится жертвовать эффективностью в ущерб универсальности – и наоборот.