

Московский Государственный Технический Университет имени Н.Э. Баумана

Факультет: «Информатика и системы управления»

Кафедра: «Защита информации» (ИУ10)

КУРСОВАЯ РАБОТА

По дисциплине «Информатика»

Тема:

«Перехват системных вызовов»

Выполнил: Студент группы ИУ10-21 Круглов В.А.

Научный руководитель: Астрахов А.В.

Оглавление

[1. АКТУАЛЬНОСТЬ И ЗАДАЧИ 3](#_Toc419433029)

[2. ВВЕДЕНИЕ 3](#_Toc419433030)

[2.1. Назначение системных вызовов 3](#_Toc419433031)

[2.2. Принцип работы системных вызовов 4](#_Toc419433032)

[3. ПЕРЕХВАТ ПОЛЬЗОВАТЕЛЬСКИХ ВЫЗОВОВ 7](#_Toc419433033)

[3.1. API операционных систем 7](#_Toc419433034)

[3.2. Описание методов перехвата API-функций 7](#_Toc419433035)

[3.2.1. Метод инъекции кода 7](#_Toc419433036)

[3.2.2. Модификация таблицы импорта 8](#_Toc419433037)

[3.3. Реализация метода Code Injection 8](#_Toc419433038)

[3.4. Реализация метода модификации таблицы импорта 13](#_Toc419433039)

[3.5. Комбинирование методов и внедрение DLL 16](#_Toc419433040)

[4. ЗАЩИТА ОТ ПЕРЕХВАТА 20](#_Toc419433041)

[5. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21](#_Toc419433042)

[6. Приложение 22](#_Toc419433043)

[6.1. Исходный код программы, реализующей перехват функции методом инъекции кода 22](#_Toc419433044)

[6.2. Исходный код программы, реализующей перехват функции методом модификации таблицы импорта 23](#_Toc419433045)

# 1. АКТУАЛЬНОСТЬ И ЗАДАЧИ

Главной задачей моей работы является подробное изучение механизма системных вызовов ОС Windows, а также поиск методов использования уязвимостей системы, связанных с системными вызовами.

В рамках моей работы подробно будут разобраны методы перехвата API-функций Windows для ознакомления с основными механизмами перехвата системных вызовов на простых примерах.

Методы перехвата системных вызовов широко используются для разработки вредоносного ПО, которое таким образом может получить доступ к манипуляции компьютером на аппаратном уровне. К такому ПО относят так называемые руткиты (RootKit), которые, внедряясь в ядро операционной системы, могут использоваться злоумышленниками для достижения каких-либо целей, скрывая при этом следы своего нахождения на компьютере. Обнаружить такие программы зачастую очень сложно.

# 2. ВВЕДЕНИЕ

## 2.1. Назначение системных вызовов

Обязательным элементом во всех ОС является ядро.

Ядро - главная часть операционной системы, которая постоянно находится в оперативной памяти и служит для управления ОС.

Ядро ОС является низшим уровнем абстракции от аппаратного обеспечения компьютера. В его функции входит распределение ресурсов процессора и оперативной памяти между пользовательскими приложениями, обеспечение доступа к файловой системе, а также «общение» с внешними устройствами.

Начиная с Windows NT, во всех операционных системах от Microsoft реализован принцип привилегированности выполнения программ. Привилегии используются для обеспечения безопасности в ОС.

В современных ОС разделяют 4 уровня привилегий. Уровень «0» является наиболее привилегированным: в нем работает ядро операционной системы. Уровень «3» является, соответственно, наименее привилегированным: в этом уровне работают все пользовательские приложения. Операционные системы Windows используют только уровни «0» и «3», игнорируя промежуточные. Уровни «0» и «3» иногда называют режимом ядра и пользовательским режимом соответственно.

Каждый из уровней привилегий ограничен своим адресным пространством. Таким образом, пользовательские приложения не могут получать доступ к приложениям, исполняемым с более низким (0) уровнем привилегий, и не могут получать прямой доступ к ресурсам системы.

Однако многим приложениям для нормальной работы необходим доступ к системным компонентам. Для обеспечения такого доступа в ОС Windows существует механизм **системных вызовов**.

Итак, системный вызов – это обращение прикладной программы к ядру ОС для выполнения какой-либо операции, требующей численно более низкого уровня привилегий. Простейшими примерами таких операций являются операции работы с файловой системой, такие как открытие или создание файла.

Системный вызов обычно происходит в виде вызова подпрограммы или функции из системной библиотеки.

Говоря простым языком, процессор, встретив в процессе выполнения программы вызов одной из таких функций, прерывает выполнение программы, передает управление ядру ОС, сохраняя при этом место, на котором завершилось выполнение. Происходит программное прерывание, после которого необходимая процедура или функция выполняется в режиме ядра ОС, а затем продолжается выполнение прерванной программы.

## 2.2. Принцип работы системных вызовов

Уровень привилегий исполняемой программы определяется уровнем привилегий текущего сегмента кода. Каждый сегмент кода в операционной системе на процессоре x86 описывается 8-байтой структурой данных, называемой сегментным дескриптором. Сегментный дескриптор содержит начальный адрес описываемого сегмента кода, его длину и уровень привилегий, на котором будет исполняться код, содержащийся в этом сегменте.

Процессор не позволит коду с более низким (численно большим) уровнем привилегий непосредственно сделать вызов в код с более высоким (численно меньшим) уровнем привилегий.

Для того чтобы контролировать переходы между кодом с разным уровнем привилегий, Windows использует возможность x86-процессора, называемую **шлюз прерывания**.

x86-процессор имеет таблицу векторов прерывания (IDT), которая содержит информацию о том, как следует обрабатывать каждое прерывание. В реальном режиме таблица векторов прерываний x86-процессора просто содержит указатели (4 байта каждый) на процедуры обслуживания прерываний. Но в защищенном режиме таблица векторов прерываний содержит **дескрипторы шлюзов прерываний**, которые являются 8-байтовыми структурами, которые описывают, как данное прерывание должно быть обработано. Дескриптор шлюза прерывания содержит информацию о том, какой сегмент кода содержит процедуру обслуживания прерываний и ее адрес внутри сегмента.

Проверяя уровень привилегий в дескрипторе шлюза прерывания процессор гарантирует, что вызывающему приложению разрешено вызывать код ядра только в специально отведенных для этого точках (в этом и заключается суть термина "шлюз прерывания", т.е. это четко определенный шлюз, который может передать управление от кода уровня пользователя к коду уровня ядра).

Дескриптор шлюза прерывания содержит **селектор** **сегмента** (структуру данных, хранящую положение дескриптора сегмента в глобальной таблице дескрипторов (GDT)), однозначно определяющий дескриптор, который описывает сегмент кода, в котором располагается процедура обслуживания прерывания.

Прежде, чем описывать механизм системных вызовов, необходимо сказать несколько слов и о регистрах процессора.

Регистры процессора – это блок ячеек памяти, образующий сверхбыструю оперативную память (СОЗУ) внутри процессора, в большей части недоступную программисту и пользовательским программам.

В 32-битной архитектуре процессора х86 существует 16 регистров. Нас интересуют сегментные регистры и т.н. «счетчик команд».

В сегментном регистре CS (Code Segment) содержится селектор исполняемого сегмента кода.

В сегменте EIP (Extended Instruction Pointer) содержится смещение следующей исполняемой инструкции относительно текущего сегмента кода в регистре CS.

Теперь можно описать, как же именно системный вызов Windows находит свой путь из режима пользователя в режим ядра. Системные вызовы в Windows инициируются исполнением инструкции "int 2e". Команда "int" заставляет процессор сгенерировать программное прерывание, т.е. проследовать в IDT (таблицу векторов прерываний) по индексу 2e и считать находящийся там дескриптор шлюза прерывания. Используя селектор из дескриптора шлюза, процессор загрузит дескриптор сегмента кода и установит значение регистра EIP на смещение точки входа в процедуру обслуживания прерывания, взятое из дескриптора шлюза. На этом шаге процессор уже почти готов начать исполнение кода процедуры обработки прерывания в сегменте режима ядра.

Прежде чем процессор начнет исполнять ISR в кодовом сегмента режима ядра, он должен переключиться на стек ядра.

Каждый уровень привилегий x86-процессора имеет свой собственный стек. Когда происходит вызов функции более привилегированного уровня через шлюз прерывания, процессор автоматически сохраняет значения некоторых регистров, в т.ч. CS и EIP, о которых говорилось раньше.

Все системные вызовы Windows используют одно и тоже программное прерывание ''int 2e'' для переключения в режим ядра. Для информирования ядра о том, какую именно системную функцию нужно выполнить, индекс помещается в регистр EAX перед вызовом инструкции ‘’int 2e’’. В режиме ядра ISR (процедура обслуживания прерываний - Interrupt Service Routines) считывает значение регистра EAX и вызывает указанную системную функцию, если все переданные параметры проходят предварительную проверку. Параметры системного вызова передаются функции ядра в теле ISR.

Когда работа системного вызова завершается, исполняется инструкция ‘’iret’’. Исполняя эту инструкцию, процессор восстанавливает из стека сохраненные значения регистров пользовательского режима и продолжает исполнение с инструкции, следующей за инструкцией ''int 2e''.

# 3. ПЕРЕХВАТ ПОЛЬЗОВАТЕЛЬСКИХ ВЫЗОВОВ

## 3.1. API операционных систем

Каждая ОС имеет API (Application Programming Interface - интерфейс прикладного программирования) – набор готовых классов, функций, процедур, структур и констант, предоставляемых операционной системой, которые могут использоваться программистами для написания прикладных программ.

Множество системных вызовов – это главный API операционной системы.

Под пользовательскими вызовами мы в своей работе будем подразумевать некоторые функции Windows API, которые не являются системными вызовами. То есть, для их выполнения не требуется переход в режим ядра ОС.

Прежде, чем переходить к перехвату системных вызовов, я хочу научиться перехватывать API-функции пользовательского режима. Общий принцип перехвата в обоих случаях будет похожим, однако процесс перехвата вызовов в пользовательском режиме связан с гораздо меньшими трудностями и позволяет более прозрачно продемонстрировать, какие процессы при этом происходят.

## 3.2. Описание методов перехвата API-функций

Выделяют всего два основных метода:

* Модификация таблицы импорта
* Инъекция кода

Опишем каждый из них.

### 3.2.1. Метод инъекции кода

Этот метод также известен под названием Code Injection. Остановимся на нем более подробно.

Простыми словами его можно описать так: необходимо найти исходный код перехватываемой функции, стереть в нем первые несколько байт и записать в них инструкцию безусловного перехода на нашу собственную функцию.

При этом необходимо сохранить стертые инструкции на случай, если нам понадобится исполнить оригинальный код. В таком случае эти инструкции должны быть реализованы в нашей функции-перехватчике, после которых должна следовать инструкция безусловного перехода к незатронутому участку перехватываемой функции.

Для того, чтобы получить адрес в памяти перехватываемой функции, мы можем воспользоваться отладчиком.

**Отладчик** – это компьютерная программа, используемая для поиска ошибок в других программах. С ее помощью можно просмотреть адресное пространство, доступное интересующему нас приложению, и отследить пошаговое выполнение кода.

Таким образом, мы легко получим адрес какой-нибудь функции в памяти.

Примером популярного отладчика является IDA.

### 3.2.2. Модификация таблицы импорта

Этот метод является классическим. Он довольно подробно описан в книге Дж. Рихтера «Windows для профессионалов».

Помимо прочего, API подразумевает использование импортируемых функций. Импортируемая функция – это функция, фактически находящаяся в другом модуле, нежели та, которая ее вызывает.

Каждый .exe или .dll файл содержит таблицы импорта, в которых указаны все используемые DLL-библиотеки, а также перечислены используемые функции из других библиотек с указанием их адреса. Вызывая импортируемую функцию, поток получает ее адрес из таблицы импорта.

Метод перехвата в таком случае заключается в подмене адреса перехватываемой функции в таблице импорта на адрес нашей собственной функции.

## 3.3. Реализация метода Code Injection

Попытаемся разобрать описанный нами метод на простом примере.

Напишем простую программу на C++, использующую функции **CreateFile** и **CloseHandle,** и попробуем их перехватить.

Для начала разберемся в структуре самих этих функций.

Функция CreateFile может использоваться для создания файла. Она описывается следующим образом:

HANDLE CreateFile(

LPCTSTR lpFileName,

DWORD dwDesiredAccess,

DWORD dwShareMode,

LPSECURITY\_ATTRIBUTES lpSecurityAttributes,

DWORD dwCreationDisposition,

DWORD dwFlagsAndAttributes,

HANDLE hTemplateFile

);

Аргументы этой функции:

IpFileName – Полное имя создаваемого файла.

dwDesiredAccess – Тип доступа к файлу.

dwShareMode – Режим разделяемого доступа к файлу. При нулевом значении объект не может быть использован совместно несколькими программами.

lpSecurityAttributes - Адрес структуры, которая определяет, может ли возвращаемый функцией дескриптор быть унаследован дочерним процессом.

dwCreationDisposition – Определяет действие, которое необходимо совершить.

dwFlagsAndAttributes – Атрибуты файла.

hTemplateFile – Дескриптор файла-шаблона с правами доступа.

Если функция успешна, она вернет открытый дескриптор указанного файла.

Функция **CloseHandle** закрывает дескриптор файла, возвращаемый функцией **CreateFile**.

Итак, код нашей тестовой программы будет выглядеть следующим образом:

**int** \_tmain(**int** argc, \_TCHAR\* argv[])

{

HANDLE hFile = CreateFile(L"C:\\test.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

CloseHandle(hFile);

**return** 0;

}

Ставим брейкпоинт на функцию **CreateFile** и запускаем отладку, после чего открываем дизассемблированный код. Вот, что мы в нем видим:

HANDLE hFile = CreateFile(L"d:\\test.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);  
01138A8B mov esi,esp   
01138A8D push 0   
01138A8F push 80h   
01138A94 push 2   
01138A96 push 0   
01138A98 push 0   
01138A9A push 40000000h   
01138A9F push offset string L"d:\\test.txt" (11415B0h)   
01138AA4 call dword ptr [\_\_imp\_\_CreateFileW@28 (114527Ch)]   
01138AAA cmp esi,esp   
01138AAC call @ILT+1000(\_\_RTC\_CheckEsp) (11313EDh)   
01138AB1 mov dword ptr [hFile],eax

Далее нам необходимо найти инструкцию вызова нашей функции. Это инструкция

call dword ptr [\_\_imp\_\_CreateFileW@28 (114527Ch)].

Наводим на нее указатель и нажимаем F11, после чего попадаем в тело самой функции:

76D60B7D mov edi,edi   
76D60B7F push ebp   
76D60B80 mov ebp,esp   
76D60B82 push ecx   
76D60B83 push ecx   
76D60B84 push dword ptr [ebp+8]   
76D60B87 lea eax,[ebp-8]   
76D60B8A push eax   
76D60B8B call dword ptr ds:[76D11568h]

Посмотрим на начало:

76D60B7D mov edi,edi   
76D60B7F push ebp   
76D60B80 mov ebp,esp

Первая инструкция не делает абсолютно ничего, занимая целых 2 байта.

Дальнейшие две команды занимают три байта и необходимы для того, чтобы вершина стека [esp] указывала на адрес точки возврата в функцию, а не на мусорное значение, оставшееся после выполнения других программ.

Функция **CreateFile** использует соглашение вызова stdcall, как и большинство других API-функций. Благодаря этому все API-функции имеют одинаковое начало. Это позволяет нам иметь гарантированные 5 байт в начале любой из этих функций для инъекции команды jmp с указанием адреса нашей собственной функции.

Проблема в том, что не существует инструкции jmp, которая бы принимала абсолютный адрес нашей функции и имела бы размер 5 байт. Минимальный размер такой инструкции – 6 байт у команды jump far. Поэтому мы будем использовать jump near, который принимает относительный адрес (то есть разность между адресом точки назначения и следующей за jump near инструкцией). По факту, для вычисления параметра операции jump near, надо из адреса точки назначения вычесть адрес исходной точки и прибавить 5 байт (именно столько эта инструкция занимает).

size\_t \_CalculateDispacement(**void**\* lpFirst, **void**\* lpSecond)

{

**return** **reinterpret\_cast**<**char**\*>(lpSecond) - (**reinterpret\_cast**<**char**\*>(lpFirst) + 5);

}

Опкод функции jump near – это 0xe9. Таким образом, мы можем выполнить перехват следующим образом:

HANDLE WINAPI \_My\_CreateFileW(LPCWSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY\_ATTRIBUTES lpSecurity, DWORD dwCreationDisp, DWORD dwFlags, HANDLE hTemplate)

{

OutputDebugStringA(\_\_FUNCTION\_\_);

**return** (HANDLE)-1;

}

#**pragma pack(push, 1)**

**struct** jump\_near

{

BYTE opcode; // 0xe9

DWORD relativeAddress;

};

#**pragma pack(pop)**

**int** \_tmain(**int** argc, \_TCHAR\* argv[])

{

HMODULE hKernel32 = GetModuleHandle(L"kernel32.dll");

jump\_near\* lpFunc = **reinterpret\_cast**<jump\_near\*>(GetProcAddress(hKernel32, "CreateFileW"));

lpFunc->opcode = 0xe9;

lpFunc->relativeAddress = \_CalculateDispacement(lpFunc, &\_My\_CreateFileW);

HANDLE hFile = CreateFile(L"d:\\test.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

CloseHandle(hFile);

**return** 0;

}

Функция \_My\_CreateFileW имеет такую же структуру, что и оригинальная (число параметров, их типы и тип возвращаемых данных). На нее мы будем подменять оригинал.

Для того, чтобы устранить паразитные влияния оптимизации компилятором структур для выравнивания секций данных, мы воспользуемся директивой #pragma pack, которая, в нашем случае, выравнивает данные по границе байта.

Запускаем код на выполнение. Перехват осуществлен!

Теперь необходимо разобраться с тем, как вызвать оригинальную функцию. Для этого мы должны сделать следующее:

1. Сохранить указатель на начало функции.

2. Создать в памяти блок размером 10 байт с правами на выполнение кода.

3. Скопировать туда первые 5 байт исходной функции до установки туда перехватчика.

4. Создать в последних 5 байтах аналогичную инструкцию jump near, которая будет переправлять выполнение функции на оригинальный обработчик, пропуская затертые нами 5 байт.

5. Сохранить адрес 10-байтового блока и привести его к типу CreateFileWProc, который описан следующим образом:

typedef HANDLE (WINAPI \*CreateFileWProc)(LPCWSTR, DWORD, DWORD, LPSECURITY\_ATTRIBUTES, DWORD, DWORD, HANDLE);

6. Теперь, если нам нужно вызывать оригинал – мы просто пользуемся этим указателем.

Теперь посмотрим, что получилось в дизассемблированном коде после всех этих манипуляций.

Снова найдем вызов оригинальной функции:

000D8AD0 call dword ptr [\_\_imp\_\_CreateFileW@28 (0E527Ch)]

И нажмем F11. В ее начале увидим:

76D60B7D jmp \_My\_CreateFileW (0D13E8h)

Это 5-байтовый jump near, установленный нашим перехватчиком.

Нажав еще раз F11, попадем в тело нашей функции.

HANDLE WINAPI \_My\_CreateFileW(LPCWSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY\_ATTRIBUTES lpSecurity, DWORD dwCreationDisp, DWORD dwFlags, HANDLE hTemplate)  
000D8910 push ebp   
000D8911 mov ebp,esp   
000D8913 sub esp,0C0h   
000D8919 push ebx   
000D891A push esi   
000D891B push edi   
000D891C lea edi,[ebp-0C0h]   
000D8922 mov ecx,30h   
000D8927 mov eax,0CCCCCCCCh   
000D892C rep stos dword ptr es:[edi]

Дойдем в нем до вызова оригинальной функции:

000D8960 call dword ptr [\_Std\_CreateFileW (0E4240h)]

И нажмем F11.

00060000 mov edi,edi   
00060002 push ebp   
00060003 mov ebp,esp   
00060005 jmp 76D60B82

Мы попали в так называемый трамплин – код, который исполняет вырезанную часть оригинальной функции и передает управление обратно в оригинал. Проверим. Дойдем до последней инструкции и снова нажмем F11:

76D60B7D jmp \_My\_CreateFileW (0D13E8h)   
76D60B82 push ecx   
76D60B83 push ecx

Вспомним самое начало и поймем, что после инструкции jmp действительно следует продолжение тела оригинальной функции.

Всё работает так, как надо.

## 3.4. Реализация метода модификации таблицы импорта

В Windows все исполняемые файлы, файлы динамических библиотек и объектного кода имеют формат PE (Portable Executable). Это формат представляет собой некую структуру данных, содержащую несколько обязательных элементов: заголовки и секции. В заголовках содержится информация, описывающую различные свойства исполняемого файла и его структуру. Секции содержат данные, которые размещаются в адресном пространстве процесса во время загрузки исполняемого файла в память.

Все PE-файлы как правило имеют таблицу импорта (IAT). Эта таблица заполняется фактическими адресами импортируемых функций из других модулей на стадии запуска исполняемого файла.

Перехват импортируемой функции можно осуществить, подменив адрес вызываемой функции в таблице импорта исполняемого файла на адрес нашей функции-обработчика. При этом, так как таблица импорта заполняется адресами функций каждый раз при загрузке исполняемого файла, модификация будет действовать до завершения работы файла.

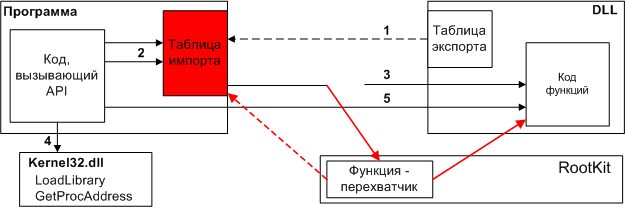


Рис. 3.1. Схема работы перехватчика

Следует отметить, что перехват с помощью модификации таблицы импорта может производиться как на уровне самого приложения, так и на уровне библиотек, которые оно использует (каждая из них имеет свою таблицу импорта). Такой перехват гораздо сложнее обнаружить.

Мы, однако, не ставим себе цели разработать ПО для применения в полевых условиях, поэтому разберем метод на простом примере.

Наша задача - реализовать набор функций, которые будут принимать в качестве параметров:

* Имя модуля (B), вызовы из которого в модуль (A) мы собираемся перехватывать
* Имя модуля (A), вызов функций которого мы собираемся перехватывать
* Имя функции модуля (A) импортируемой модулем (B) для перехвата
* Адрес нашего обработчика

Реализуем функцию, которая будет возвращать статус перехвата (удачно / не удачно) и адрес оригинального обработчика. Последнее необходимо, поскольку после выполнения нашего обработчика должна выполниться оригинальная функция.

bool PatchImportTable(LPCSTR lpszModule, // B

LPCSTR lpszDLL, // A

LPCSTR lpszEntry, // Имя импортируемой функции

PVOID pHook, // Наш обработчик

PVOID \*ppOriFunc); // Сюда возвращаем оригинальный адрес

Первый этап - это определение адреса загрузки модуля B. Для этого воспользуемся функциями **EnumProcessModules** и **GetModuleInformation.** Первая строит список всех загруженных модулей для приложения, а вторая возвращает адрес загрузки интересующего нас модуля.

Второй этап – нахождение таблицы импорта в интересующем нас файле и нахождение уже в этой таблице записи об интересующей нас функции.

Таблица импорта представляет собой 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.

Эта запись, очевидно, является структурой. Один из элементов этой структуры - PIMAGE\_THUNK\_DATA FirstThunk – является ссылкой на массив указателей, который и заполняется адресами импортируемых функций при загрузке исполняемого файла в память.

Теперь можем получить адрес заголовка PE-файла:

PIMAGE\_NT\_HEADERS pPEHdr; // PE header address

pPEHdr = GetNtHeader(pImage);

И находим адрес таблицы импорта IAT:

ULONG uDirBaseRVA = GetDirBaseRVA(pPEHdr, IMAGE\_DIRECTORY\_ENTRY\_IMPORT);

ULONG uDirSize = GetDirSize (pPEHdr, IMAGE\_DIRECTORY\_ENTRY\_IMPORT);

ULONG uIATDirBase = GetDirBaseRVA(pPEHdr, IMAGE\_DIRECTORY\_ENTRY\_IAT);

ULONG uIATDirSize = GetDirSize (pPEHdr, IMAGE\_DIRECTORY\_ENTRY\_IAT);

Далее находим запись IMAGE\_IMPORT\_DESCRIPTOR, соответствующую библиотеке A. После этого в PIMAGE\_THUNK\_DATA FirstThunk находим перебором импорт-дескриптор (IMAGE\_IMPORT\_BY\_NAME) для функции lpszEntry и соответствующий ему вход в IAT. Если вход был найден (наша функция вернула TRUE), сохраняем адрес оригинальной функции и пишем вместо него адрес перехватчика. Перехват выполнен.

## 3.5. Комбинирование методов и внедрение DLL

При реализации описанных методов перехвата API-функций могут возникнуть некоторые трудности. Одна из проблем может заключаться в том, что в адресном пространстве процесса попросту не хватит места для нашего с вами кода.

Такая проблема может быть решена путем оформления нашего кода в файл формата .dll (динамически подключаемую библиотеку) и внедрения этой библиотеки в адресное пространство интересующего нас приложения.

Внедрение DLL также можно выделить в отдельный способ осуществления хакерской атаки, наряду с перехватом вызовов.

Помимо этого, внедрение DLL тесно связано с перехватом функций методом модификации таблицы импорта. Динамические библиотеки DLL, как и файлы .exe, являются PE-файлами. Значит, для DLL-библиотек также справедливо все, что было сказано о модификации таблиц импорта для любых процессов.

С другой стороны, внедрение DLL можно осуществить с помощью модификации таблицы импорта приложения. Правда, несколько другим образом, чем тот, что описывался ранее в моей работе.

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. Стало быть, достаточно добавить к списку используемых программой DLL-библиотек свою, у которой функция DllMain содержит необходимый нам код.

Есть несколько способов осуществить внедрение DLL.

Первый способ связан с использованием реестра. Увы, он будет работать только в ОС Windows линейки NT (NT, 2000, XP и т.д.).

Большинство приложений в этих ОС используют библиотеку user32.dll. При запуске приложения, использующего её, к нему подключаются также все библиотеки, перечисленные в специальном ключе реестра: **HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit\_DLLs.**

Мы легко можем записать в этом ключе реестра и нашу библиотеку. После этого операционной системе потребуется перезагрузка.

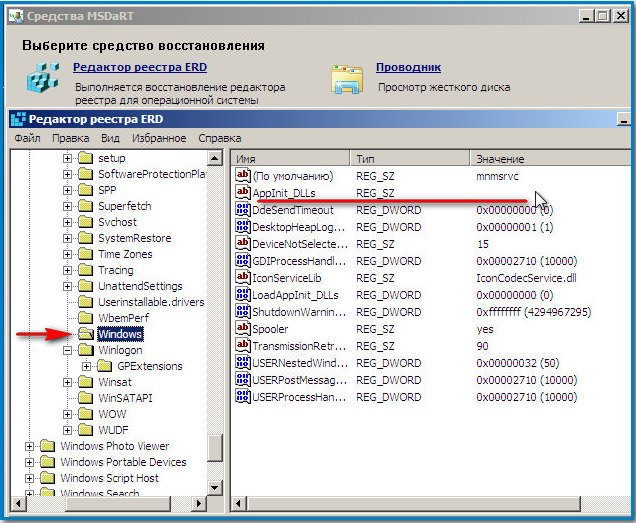


Рис. 3.2. Регистр Windows XP

Такой способ внедрения библиотеки имеет ряд существенным недостатков. Как уже было сказано выше, он не будет работать на современных версиях ОС Windows. Кроме того, что этот способ прост в исполнении, он также прост в обнаружении, что делает затруднительным его использование в реальных задачах.

Второй способ позволит нам внедрить свою DLL в процесс с помощью т.н. ловушек.

Приложения получают массу сообщений, уведомляющих о различных событиях, например, нажатие на клавишу, когда одно из окон приложения активно. В Windows реализован способ, позволяющий перехватывать оконные сообщения чужого процесса. Именно его мы и используем для внедрения DLL в чужое адресное пространство.

Чтобы создать ловушку, используем функцию **SetWindowsHookEx(WH\_GETMESSAGE, GetMbgProc, hiribtDll, 0),** которая принимает на вход 4 аргумента. Первый – тип перехватываемого события (например, WH\_KEYBOARD для нажатия на клавишу), второй – указатель на процедуру-обработчик, третий – виртуальный адрес библиотеки, которая содержит процедуру-обработчик, и последний параметр – это идентификатор потока, который мы хотим захватить, либо 0 для захвата всех потоков.

Мы создаем процесс, который вызывает создание ловушки на событие **WH\_GETMESSAGE** (процесс А). Запускаем процесс, в который хотим внедрить библиотеку (процесс В), и инициализируем совершение этого события. Далее происходит следующее:

Система проверяет, установлена ли ловушка для данного потока и данного события. Затем проверяет, спроецирована ли библиотека **hiribt.dll** в адресное пространство процесса В, и если нет – делает это. Система проверяет, не совпадают ли значения hinstDll (параметр hinstDll определяет адрес библиотеки в адресном пространстве процесса) этой DLL, относящиеся к процессам А и В. Если hinstDll в обоих процессах одинаковы, то и адрес **GetMsgProc** в этих процессах тоже одинаков. Тогда система может просто вызвать **GetMsgProc** в адресном пространстве процесса А. Если же hinstDll различны, система определяет адрес функции **GetMsgProc** в адресном пространстве процесса В по формуле:

**GetMsgProc В = hinstDll В + (GetMsgProc А - hinstDll А)**.

Далее вызывается **GetMsgProc** в адресном пространстве процесса В.

Очевидно, в библиотеке **hiribtDll** помимо процедуры обработки события должен содержаться код, который мы хотим выполнить в результате перехвата функции (ведь именно ради этого мы все и затеяли).

Следующий способ подключения своей DLL-библиотеки к чужому процессу использует создание удаленных потоков. Этот способ наиболее сложный в реализации, но и наиболее гибкий. Не будем рассказывать о нем очень подробно, опишем лишь базовый механизм его работы.

Большинство Windows-функций позволяет процессу управлять лишь самим собой, исключая тем самым риск повреждения одного процесса другим. Однако есть и такие функции, которые дают возможность управлять чужим процессом. Внедрение DLL этим способом предполагает вызов функции **LoadLibrary** потоком целевого процесса для загрузки нужной DLL. Так как управление потоками чужого процесса сильно затруднено, мы должны создать в нем свой удаленный поток, который и загрузит нужную нам DLL.

Создать удаленный процесс нам поможет функция **CreateRemoteThread.** Она принимает в качестве аргументов, среди прочих, адрес функции потока и идентификатор процесса, в котором создается удаленный поток.

# 4. ЗАЩИТА ОТ ПЕРЕХВАТА

Помимо методов перехвата API-функций стоит рассмотреть еще и методы защиты от перехвата. Познакомимся с некоторыми из них.

Метод первый: сравнение начала функции перед ее вызовом с опкодами машинных инструкций.

//0x90 - nop

//0xE9 - jmp

//0xE8 - call

if (\*mainFunc == 0xE9 || \*mainFunc == 0x90 || \*mainFunc == 0xE8 ...)

printf("Hook detected!");

Метод простой и легко обходится более хитрыми переходами.

Второй метод: используя таблицы экспорта в системных библиотеках, сравнивать оригинальную длину интересующих нас API-функций c длиной их дизассемблированного кода.

Перечисленные методы перехвата API-функций так или иначе заключаются в модификации памяти. Третий способ защиты предполагает обнаружение таких правок посредством специальных проверок. Проблема в том, что API-функции, ответственные за сканирование памяти, тоже могут быть перехвачены, что заставит их возвращать ложный результат.

Еще один метод заключается в побайтовом восстановлении модифицированной функции из ее библиотеки.

Также применяется имитация вызова системных функций, которые простыми методами перехвачены быть не могут. Этот метод является самым сложным и самым надежным из всех перечисленных.

# 5. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Дж. Рихтер. Windows для профессионалов. СПб.: Питер, 2000.

2. Зайцев О.В. ROOTKITS, SPYWARE/ADWARE, KEYLOGGERS & BACKDOORS: обнаружение и защита. СПб.: БХВ-Петербург, 2006.

3. Барри Брей. Микропроцессоры Intel. Архитектура, программирование и интерфейсы. Шестое издание. СПб.: БХВ-Петербург, 2005

4. Перехват API-функций [Электронный ресурс] / Коллективный блог Хабрахабр. URL: <http://habrahabr.ru> (Дата обращения – 23.04.2015)

5. Установка хука на вызов функции через таблицу импорта [Электронный ресурс] / Windows Programming Tips. Заметки разработчика. URL: <http://tips.efmsoft.com> (Дата обращения – 03.05.2015)

6. RootKit - принципы и механизмы работы. [Электронный ресурс] / Информационная безопасность. URL: <http://z-oleg.com> (Дата обращения – 03.05.2015)

7. Лукач Ю.С. Структура исполняемых файлов Win32 и Win64 [Электронный ресурс] / Сборник ресурсов «Компьютерные науки» математико-механического факультета УрГУ. URL: <http://cs.usu.edu.ru/> (Дата обращения – 04.05.2015)

8. Жесткое внедрение DLL в Windows-программы. [Электронный ресурс] / Информационная безопасность без паники и всерьез. URL: <https://bugtraq.ru> (Дата обращения – 04.05.2015)

# 6. Приложение

## 6.1. Исходный код программы, реализующей перехват функции методом инъекции кода

#pragma once

// Файлы заголовков Windows:

#include <windows.h>

// Файлы заголовков C RunTime

#include <stdlib.h>

#include <stdio.h>

#include <malloc.h>

#include <memory.h>

#include <tchar.h>

// указатель на функцию-двойник (с таким же соглашенеим вызова) CreateFileW

typedef HANDLE(WINAPI \*CreateFileWProc)(LPCWSTR, DWORD, DWORD, LPSECURITY\_ATTRIBUTES, DWORD, DWORD, HANDLE);

CreateFileWProc Std\_CreateFileW;

// функция-перехватчик

HANDLE WINAPI \_My\_CreateFileW(LPCWSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY\_ATTRIBUTES lpSecurity, DWORD dwCreationDisp, DWORD dwFlags, HANDLE hTemplate)

{

printf\_s("Called CreateFileW in module kernel32.dll! Arguments:\n");

printf\_s("lpFileName = %ls\ndwDesiredAccess = %d\ndwShareMode = %d\nlpSecurity = %d\ndwCreationDisp = %d\ndwFlags = %d\nhTemplate = %X", lpFileName, dwDesiredAccess, dwShareMode, lpSecurity, dwCreationDisp, dwFlags, hTemplate);

return Std\_CreateFileW(L"E:\\hacked.txt", dwDesiredAccess, dwShareMode, lpSecurity, dwCreationDisp, dwFlags, hTemplate);

}

// подсчет смещения

size\_t \_CalculateDispacement(void\* lpFirst, void\* lpSecond, DWORD offset)

{

return reinterpret\_cast<char\*>(lpSecond)-(reinterpret\_cast<char\*>(lpFirst) + offset);

}

#pragma pack(push, 1) //Включаем выравнивание полей по 1 байту

struct jump\_near // 5 bytes: jmp(1 byte) offset(4 bytes)

{

BYTE opcode; // 0xE9 == jmp

DWORD relativeAddress;

};

#pragma pack(pop) //Выравнивание по дефолту

int \_tmain(int argc, \_TCHAR\* argv[])

{

//получим адрес kernel32.dll в памяти

HMODULE hKernel32 = GetModuleHandle(L"kernel32.dll");

//оригинальный адрес сохраняем

DWORD\* original\_adress = reinterpret\_cast<DWORD\*>(GetProcAddress(hKernel32, "CreateFileW"));

//printf("original\_adress = %X\n", original\_adress);

//сохраним оригинальную последовательность байт в начале функции

const unsigned short mem\_length = 5;

BYTE original\_bytes[mem\_length];

memcpy(original\_bytes, original\_adress, mem\_length);

/\*printf("original\_bytes = ");

for (int i = 0; i < mem\_length; i++)

printf("%X ", original\_bytes[i]);

printf("\n");\*/

//Создаем указатель на 5 байт по адресу функции CreateFileW

jump\_near\* lpFunc = reinterpret\_cast<jump\_near\*>(GetProcAddress(hKernel32, "CreateFileW"));

//unprotect

DWORD dwOldProtect; // PAGE\_EXECUTE\_READ = 0x20

VirtualProtect(lpFunc, sizeof(jump\_near), PAGE\_EXECUTE\_READWRITE, &dwOldProtect);

//в 1 байт ставим команду jmp

lpFunc->opcode = 0xe9;

//в остальные 4 загружаем смещение между CreateFileW и нашей инъектируемой

//при этом учитываем смещение на 5 байта из-за инъекции jmp

lpFunc->relativeAddress = \_CalculateDispacement(lpFunc, &\_My\_CreateFileW, mem\_length);

//re-protect

VirtualProtect(lpFunc, sizeof(jump\_near), dwOldProtect, NULL);

// выделим память для восстановления оригинальной последовательности байт и прыжка в оригинал

Std\_CreateFileW = reinterpret\_cast<CreateFileWProc>(malloc(10));

//unprotect

VirtualProtect(Std\_CreateFileW, 10, PAGE\_EXECUTE\_READWRITE, &dwOldProtect);

//printf("Std\_CreateFileW\_adress = %X\n", Std\_CreateFileW);

// скопируем оригинальные байты

memcpy(Std\_CreateFileW, original\_bytes, 5);

// + 1 по арифметике указателя == 5 байт!!!!

jump\_near\* jmp\_back = reinterpret\_cast<jump\_near\*>(Std\_CreateFileW) + 1;

//printf("jmp\_back = %X\n", jmp\_back);

jmp\_back->opcode = 0xe9;

jmp\_back->relativeAddress = \_CalculateDispacement((BYTE\*)Std\_CreateFileW + 10, (BYTE\*)original\_adress + 5, 0);

//re-protect

VirtualProtect(Std\_CreateFileW, 10, dwOldProtect, NULL);

HANDLE hFile = CreateFile(L"C:\\test\test.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

CloseHandle(hFile);

system("pause");

return 0;

## 6.2. Исходный код программы, реализующей перехват функции методом модификации таблицы импорта

//----------------------------- Open process ---------------------------------

HANDLE hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, GetCurrentProcessId());

if (!hProcess)

return false;

//------------------------ Retrieve module (B) handle ------------------------

DWORD cbNeeded;

HMODULE hMods[1024];

if (EnumProcessModules(hProcess, hMods, sizeof(hMods), &cbNeeded))

{

HMODULE hModule = NULL;

for (ULONG uModule = 0; uModule < (cbNeeded / sizeof(HMODULE)); uModule++)

{

char szModName[MAX\_PATH];

// Get the full path to the module's file.

if (GetModuleFileNameExA(hProcess,

hMods[uModule],

szModName,

sizeof(szModName)))

{

\_strlwr(szModName);

if (strstr(szModName, lpszModule))

{

hModule = hMods[uModule];

break;

}

}

}

// -------------------- Get base address of module (B) -----------------------

if (hModule)

{

MODULEINFO mi;

memset(&mi, 0, sizeof(mi));

BOOL f = GetModuleInformation(hProcess, hModule, &mi, sizeof(mi));

...

}

}

//------------ Look for descriptor which match to module name ----------------

PIMAGE\_IMPORT\_DESCRIPTOR pid; // Import directory

pid = PIMAGE\_IMPORT\_DESCRIPTOR(uDirBaseRVA + uBase);

for (; !pdwRet; pid++)

{

if (!pid->OriginalFirstThunk)

break; // No more descriptors

if (!\_stricmp(LPCSTR(uBase + pid->Name), pszModuleName))

{

//---------------------- Look for specified ordinal --------------------------

DWORD \* pdwThunk = (DWORD \*)(pid->OriginalFirstThunk + uBase);

DWORD dwIdx = 0;

for (; \*pdwThunk; pdwThunk++, dwIdx++)

{

if (\*pdwThunk & IMAGE\_ORDINAL\_FLAG)

continue;

PIMAGE\_IMPORT\_BY\_NAME p = PIMAGE\_IMPORT\_BY\_NAME(\*pdwThunk + uBase);

if (IsBadReadPtr(p, sizeof(IMAGE\_IMPORT\_BY\_NAME)))

break;

if (!\_stricmp((LPCSTR)p->Name, pszFunction))

{

pdwThunk = (DWORD \*)(pid->FirstThunk + uBase) + dwIdx;

//------------- Check if Thunk is lying inside of IAT bounds -----------------

if ((DWORD)pdwThunk >= uIATDirBase + uBase &&

(DWORD)pdwThunk < uIATDirBase + uBase + uIATDirSize)

{

pdwRet = pdwThunk;

}

break; // Assumes, only one match

}

}

}

}