Stealth-руткит нового поколения под Linux: охота на невидимку — ужас, переходящий в комедию

крис касперски a.k.a. мыщъх, no email

хакерская команда Immunity (известная своим клоном Ольги) в начале октября 2008 выпустила руткит под Linux 2.6, который средства массовой дезинформации уже окрестили принципиально новым и совершенно неуловимым, шокировав общественность мрачными картинами надвигающейся схватки антивирусов (которых под Linux практически нет) с чудовищным демоном имя которому DR-rootkit. а как все обстоит на самом деле?

введение

Объяснять кто такие руткиты и чем они занимаются, думаю нет необходимости. В грубом приближении это программы, которые прячут другие программы, а достигается это, как правило, перехватом определенных системных функций с целью фальсификации возвращаемого ими результата. Чаще всего перехват осуществляется путем правки служебных структур данных (таблица прерываний, таблица системных вызовов) или же модификацией кода самих системных функций. И то, и другое легко обнаруживается тривиальной проверкой целостности, а потому алгоритмы подобного типа уже лет пять как не актуальны, за что и получил прозвище "классических миссионерских".

Более изощренные руткиты отказываются от модификации кода и перезаписывают указатели на внутренние функции, хранящиеся в динамической памяти. Надежных способов детекции таких извращенцев до сих пор не придумано, но сложность их реализации, а так же привязанность к конкретной версии операционной системы, делает их заложниками лабораторных экспериментов без всякой надежды на успешную коммерческую реализацию. Многообразие никтов делает свое дело.

Задача: реализовать классический миссионерский алгоритм перехвата без правки кода/данных операционной системы, с одной стороны обеспечив простоту кодирования и совместимость с различными ядрами, а с другой — предотвратить обнаружение факта вторжения.

Рисунок 1 официальный сайт фирмы Immunity, выпустившей DR-руткит принципиально нового типа

>>> врезка досье на DR-rootkit

Характер нордический, твердый. Тьфу! И никакой не нордический, а очень даже опенсоурсный, написанный на Си с кучей ассемблерных вставок и заточенный под Linux 2.6, хотя концептуально совместимый и с другими никсами (однако, перенос на xBSD системы требует довольно обширных изменений в разных местах и вообще говоря далеко не тривиален).

Отличительные особенности руткита перечислены ниже:

- новый движок перехватчика, основанный на отладочных регистрах;
- не модифицирует таблицу дескрипторов векторов прерываний (IDT);
- не модифицирует таблицу системных вызовов (sys_table_global);
- предоставляет прозрачный интерфейс для установки/снятия хуков;
- реализован в виде загружаемого модуля ядра для Linux 2.6;
- ссылка для скачки: http://www.immunitysec.com/resources-freesoftware.shtml;

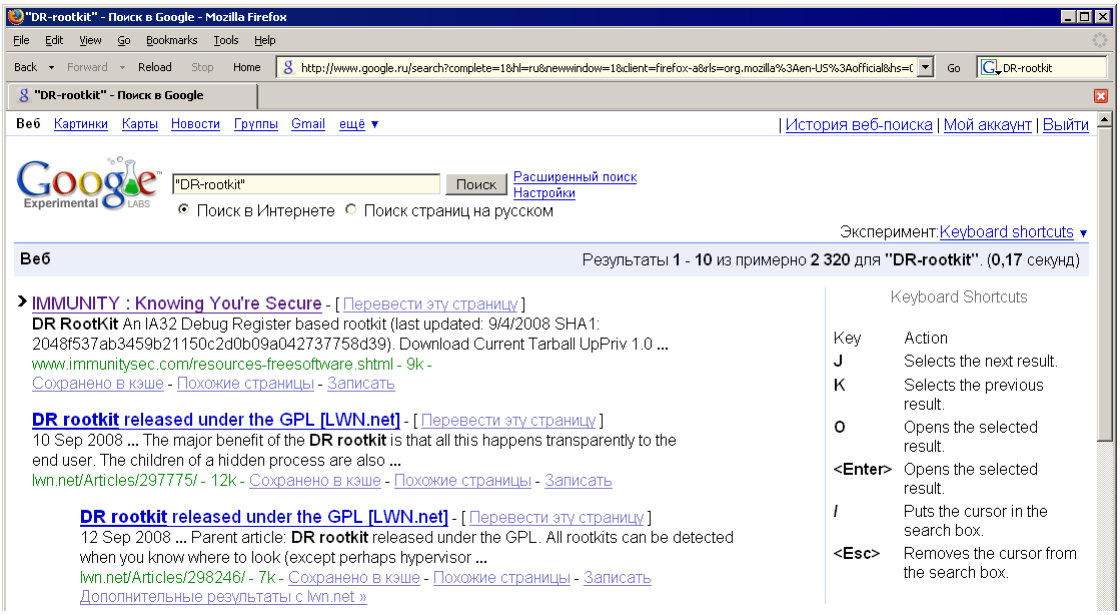

Рисунок 2 результат поиска — 2.320 страниц по запросу "DR-rootkit"

отладочные регистры на службе у контрабандистов

Идея, лежащая в основе DR-руткита на самом деле не нова, в чем разработчики и признаются, честно ссылаясь на работы halfdead'a и Pierre Falda, описывающих особенности перехвата управления под никсами, основанные на установке аппаратных точек останова на системные функции, указатели и прерывания. Windows хакеры освоили эту технологию еще со времен MS-DOS, когда термина "руткиты" вообще не существовало, а зловредные программы, скрывающие факт своего присутствия, назывались Stealth-вирусами.

Однако, многочисленные попытки создания "Голубой Пилюли" так и не увенчались успехом. Руткиты либо пались без особых усилий, либо настолько глубоко зарывались в операционную систему, что соглашались работать только со строго определенной версией, становясь непригодными для "промышленного" применения.

Собственно говоря, разработчики DR-руткита реализовали лишь базовый функционал, обнаруживаемый даже проще, чем классические миссионерские способы перехвата, а остальное пообещали дописать в "коммерческой" версии, вот только сбыться этим планам не суждено. Почему? Да потому, что на определенном этапе разработке возникнут непреодолимые технические проблемы о которых мы обязательно расскажем, но сначала разберемся с демонстрационной версией.

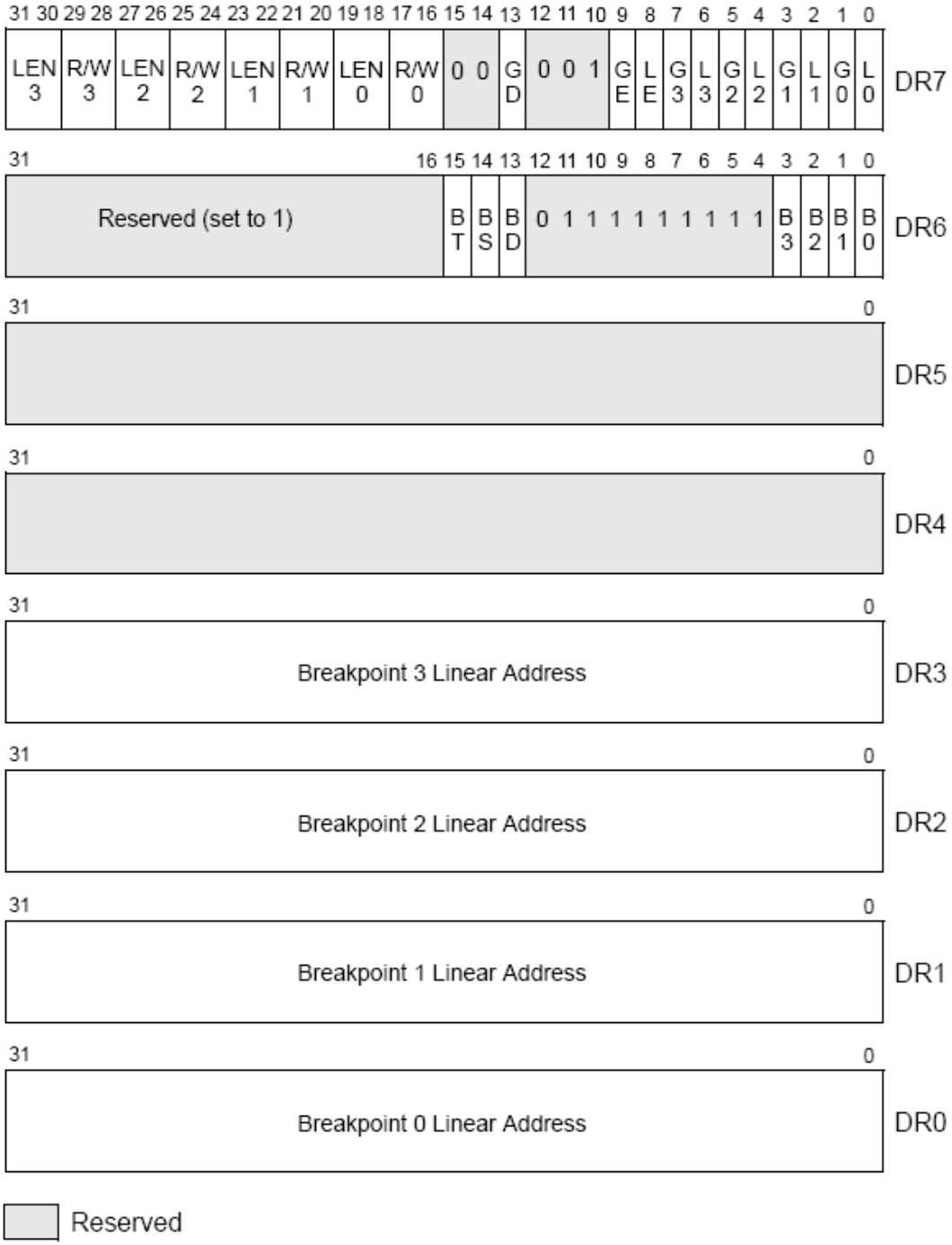

Рисунок 3 формат отладочных регистров

Ликбез на пару абзацев так же не помешает. x86-процессоры несут на своем борту четыре отладочных регистра DR0-DR3, содержащих линейные адреса (вектора прерываний) точек останова на исполнение кода и/или доступ к памяти. Управляющий регистр DR7 специфицирует атрибуты точек останова, а регистр статуса DR6 содержит информацию о текущей ситуации.

При срабатывании точки останова процессор генерирует исключение и передает управление обработку прерывания по вектору 1 (отладочное прерывание). Адреса векторов прерываний содержатся в специальной таблице, хранящейся в оперативной памяти и известной под именем IDT (Interrupt Description Table), целостность которой проверяется элементарно. На стерильной системе отладочное прерывание смотрит в ядро, а если это не так — либо установлен нестандартный отладчик уровня ядра, либо нас поимели без презерватива.

Выходит, что использование точек останова не освобождает от необходимости правки либо самой IDT, либо системного обработчика, на который указывает отладочное прерывание. В Linux системах данный обработчик указывает на функцию do_debug (реализованную в файле ./arch/i386/kernel/traps.c), которую и правит DR-руткит, причем правит весьма криво. Отыскивает инструкцию _похожую_ на CALL подменяя оригинальный целевой адрес таким образом, чтобы он указывал на хакерский обработчик. Примитив! И это они называют Stealth руткитом!!! Ладно, спишем этот недостаток на "демонстрационную" ориентацию текущей версии, хотя… как списать концептуальные просчеты? Крутись не крутись, хоть раком становись, а без модификации отладочного прерывания (или функции на которую оно указывает) не обойтись, то есть мы имеем те же самые яйца, только в профиль.

Разумеется, никто не мешает нам поставить точку останова на модифицированный код обработка прерывания, перехватывая все обращения на обращения к хакнутой ячейке памяти и подсовывая чтецу фальсифицированный результат, а всех писцов отправляя лесом (в противном случае защита может элементарно снять хакерский обработчик, перезаписав содержимое первый байт функции). Но подобные меры действуют только против пионеров. Начнем с того, что BIOS (точнее — чипсет) может скидывать содержимое оперативной памяти на диск на аппаратном уровне в обход процессора. Точки останова при этом не срабатывают. Так же, достаточно просто спроецировать банк физической памяти, где находится интересующий нас код, на соседний регион адресного пространства (опять-таки физического) и точки останова вновь не сработают. Наконец, современные процессоры обладают развитыми средствами мониторинга производительности, позволяя отслеживать количество выполненных переходов, машинных команд, обращений к памяти и т.д., а потому любая маскировка тут же становится заметной. Но это мы уже полезли в дебри. DR-руткит не предпринимает никаких попыток для маскировки факта перехвата функции do_debug, что и подтверждается нижеследующим кодом:

```
static int __get_int_handler(int offset)
{
        int idt_entry   = 0;

        // загрузка содержимого IDT таблицы посредством команды SIDT
        // с последующим определением линейного адреса отладочного прерывания
        /* off2 << 16 | off1 */
__asm__ __volatile__ ("xorl %%ebx,%%ebx             \n\t"
                      "pushl %%ebx                  \n\t"
                      "pushl %%ebx                  \n\t"
                      "sidt (%%esp)                 \n\t"
                      "movl 2(%%esp),%%ebx          \n\t"
                      "movl %1,%%ecx                \n\t"
                      "leal (%%ebx, %%ecx, 8),%%esi \n\t"
                      "xorl %%eax,%%eax             \n\t"
                      "movw 6(%%esi),%%ax           \n\t"
                      "roll $0x10,%%eax             \n\t"
                      "movw (%%esi),%%ax            \n\t"
                      "popl %%ebx                   \n\t"
                      "popl %%ebx                   \n\t"
                              : "=a" (idt_entry)
                              : "r" (offset)
                              : "ebx", "esi" );

        return idt_entry;
}

static int __get_and_set_do_debug_2_6(unsigned int handler, unsigned int my_do_debug)
{
        unsigned char *p        = (unsigned char *)handler;
...

        /* find a candidate for the call .. needs better heuristics */
        // грязный поиск машинной команды CALL offset (опокод E8h xx xx xx xx)
        // потенциально небезопасный и в определенных ситуациях приводящий
        // к ложным срабатываниям, необратимо гробящих функцию do_debug
        // и вгоняющий ядро в панику.
        while (p[0] != 0xe8)
        {
                p ++;           // если это не E8h проверяем следующий байт
        }
        DEBUGLOG(("*** found call do_debug %X\n", (unsigned int)p));
        …
        // замена старого офсета на новый, указывающий на хакерский обработчик
        // (на многопроцессорных системах возможен крах, т.к. правка кода не атомарна)
```

```
        p[1]    = (offset & 0x000000ff);
        p[2]    = (offset & 0x0000ff00) >>  8;
        p[3]    = (offset & 0x00ff0000) >> 16;
        p[4]    = (offset & 0xff000000) >> 24;
        DEBUGLOG(("*** patched in new do_debug offset\n"));

        return orig;
}

static int __init init_DR(void)
{
…
        // определение линейного адреса вектора прерывания INT 01h
        h0x01 = __get_int_handler(0x1);
        DEBUGLOG(("*** loader: handler for INT 1: %X\n", h0x01));

        /* XXX: only for debug cleanup on unload */
        h0x01_global    = h0x01;

        // правка системного обработчика отладочного прерывания
        // путем прямой правки системной функции в памяти
        /* patch the do_debug call offset in the INT 1 handler */
        __orig_do_debug = (void (*)())__get_and_set_do_debug_2_6(h0x01, \
                                        (unsigned int)__my_do_debug);
        DEBUGLOG(("*** loader: INT 1 handler patched to use __my_do_debug\n"));
…
}
```

Листинг 1 ключевой фрагмент DR-руткита, ответственный за модификацию кода функции операционной системы do_debug()

Ужас! Впрочем, для демонстрационной версии вполне сгодится. Углубляться в дальнейший анализ кода руткита нет смысла. Там все стандартно. Перехватываем диспетчер системных вызовов (на что уходит две точки останова — одна на INT 80h [old gate], другая на SYSENTER [new gate]). Третья точка останова устанавливается динамически при срабатывании любой из первых двух. Руткит анализирует какая именно системная функция вызывается и загоняет ее адрес в DR3, а при генерации отладочного прерывания, просто подменяет регистровый контекст, перенаправляя EIP на код хакерского обработчика:

```
// определение адреса системного вызова для установки динамичной точки останова
/* DR2 2nd watch on the syscall_table entry for this syscall */
dr2 = sys_table_global + (unsigned int)regs->eax * sizeof(void *);

// задание параметров точки останова
/* enable exact breakpoint detection LE/GE */
s_control   |= TRAP_GLOBAL_DR2;
s_control   |= TRAP_LE;
s_control   |= TRAP_GE;
s_control   |= DR_RW_READ << DR2_RW;
s_control   |= 3          << DR2_LEN;

DEBUGLOG(("*** dr0/dr1 trap: setting read watch on syscall_NR of %d at %X\n", \
 (unsigned int)regs->eax, dr2));

// копирование линейного адреса системного вызова в отладочный регистр DR2
/* set dr2 read watch on syscall_table */
__asm__ __volatile__ (  "movl %0,%%dr2  \n\t"
:
: "r" (dr2) );

break;
```

Листинг 2 ключевой фрагмент DR-руткита, отвечающий за установку точки останова на вызываемую системную функцию

>>> врезка: мануальная терапия

Отладочные регистры подробно описаны в главе "**Debugging And Performance Monitoring**" второго тома руководства системного программиста от Intel "**System Programming Guide, Part 2**", однако, там содержится далеко не вся информация и потому приходится листать так же и первый том ("**System Programming Guide, Part 1**"), сплошь и рядом ссылающийся на

"**Volume 1: Basic Architecture**", причем ряд тонких моментов описан только в ветхозаветном руководстве на 80486 процессор и отсутствует в более поздних редакциях.

Контекстный поиск по выражению "breakpoint" однозначно рулит, обеспечивая быструю навигацию по информации, тонким слоем "размазанной" то сотням страниц.

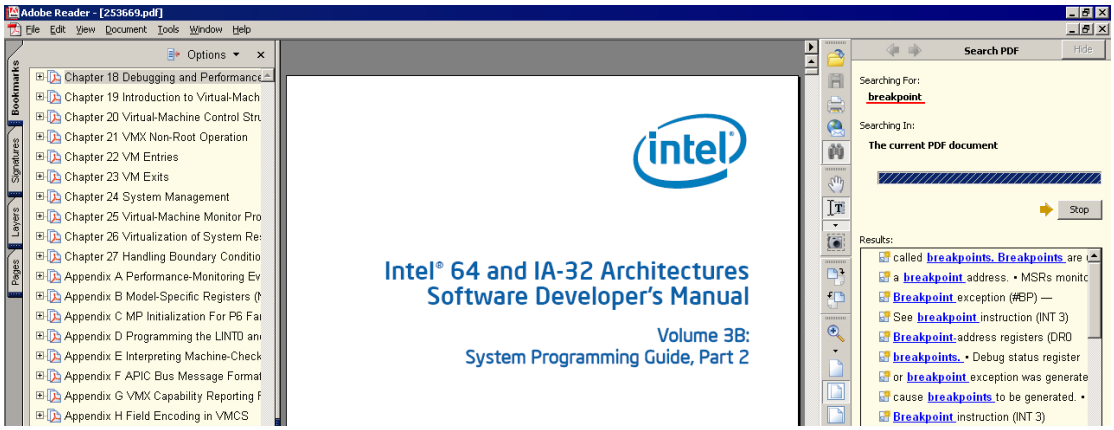

Рисунок 4 собираем информацию, размазанную тонким слоем по всей документации воедино

расстрел DR-руткита прямой наводкой

Истребители класса Stealth невидимы только для коротковолновых радаров (используемых армией США). Длинноволновые радары (морально устаревшие, но все еще не списанные) палят их без особых усилий. Поэтому для России, "Стелсы" большой угрозы не представляют.

Точно так обстоят дела и с DR-руткитом. Да, он не видим для защит, контролирующих целостность таблиц прерывания и системных вызовов, которые DR-руткит не изменяет. Но защиты рангом повыше (контролирующие целостность обработчиков прерываний и системных функций) палят DR-руткит в лет — по захаченной do_debug функции.

Поскольку, DR-руткит использует аж три отладочных регистра (из четырех имеющихся), отладчики уровня ядра с ним не работают, вызывая ужасные конфликты, обусловленные борьбой за точки останова и отладочное прерывание. Опять-таки, чисто теоретически, грамотно написанный руткит обходится всего одной точкой останова и "делится" отладочным прерыванием с отладчиком, а еще лучше — выгружает себя из памяти при активном ядерном отладчике, который позволяет "запеленговать" руткит просмотром кода операционной системы и служебных структур данных. Бороться с отладчиком, конечно, можно, но нужно ли? Ведь факт борьбы демаскирует руткита.

Кроме того, не следует забывать, что отладочные регистры могут быть прочитаны из любого ядерного модуля, состоящего всего из нескольких строк кода. Создатели DR-руткита обещают в следующей версии оказать такому способу проверки яростное сопротивление, заставив процессор генерировать исключение при любом обращении к отладочным регистрам. Процессор, действительно, может сделать это. Но вот толку с того? Это в теории все гладко и сладко. Защита читает DRx регистр для контроля его целостности. Процессор генерирует исключение, подхватываемое руткитом и возвращающего защите фиктивные данные.

Попытка практической реализации, однако, сталкивается с непреодолимым препятствием в лице операционной системы, оперирующей таким понятием как регистровый контекст. Допустим, ядро сохраняет DRx регистры для их последующего восстановления. Если руткит вернет фиктивные данные, то он и получит фиктивные данные при восстановлении контекста. ОК, блокируем восстановление контекста, прочно удерживая DRx регистры от чтения/изменения. Но… при этом они неизбежно "вырываются" с уровня ядра на прикладной уровень, вызывая непредвиденные исключения, которые руткиту придется как-то обрабатывать. Все это усложняет реализацию, делая ее системно-зависимой.

Даже если руткит сумеет корректно обработать перехват отладочных регистров, это не спасет его от расправы, поскольку тут действует правило: кто первый встал, того и тапки. В смысле: кто первый захватил отладочные регистры, тому они теперь и принадлежат. Следовательно, защитный модуль, загруженный до запуска DR-руткита, просто не позволит ему работать. Но даже если DR-руткит загрузится первым, защита, реально использующая все

четыре точки останова (а не просто читающая содержимое DRx-регистров) поставит DR-руткит перед выбором: либо дезактивировать все установленные им точки останова, совершив харакири, либо обломать защиту с установкой, тем самым разоблачив себя.

Наконец, защита может и не проверять значение отладочного прерывания, а просто создать новую таблицу прерываний и загрузить ее в процессор. Воспрепятствовать перегрузке таблицы прерываний руткит не может, а, следовательно, защита может отобрать у него отладочное прерывание, без которого руткит заглохнет как двигатель от запора. Конечно, никто не мешает руткиту перехватить одни или несколько функций операционной системы, передавая управление процедуре самовосстановления при их вызове (методика, широко использующая еще со времен MS-DOS), однако, при этом рухнет вся концепция — ведь мы же говорили о рутките, нового поколения, нашедшего "волшебный способ" перехвата, позволяющий отказаться от правки служебных таблиц и/или кода операционной системы!!!

Оказывается, в рамках данной концепции построить жизнеспособный руткит невозможно и дело ограничивается демонстрацией принципиальной возможности на уровне теоретического осмысления.

заключение

Подведем итог. Нам обещали руткит, который не модифицирует код операционной системы, используя для перехвата отладочные регистры. В действительности же, мы получили гибридный продукт, модифицирующий и отладочные регистры, и код (данные). По другому никак не получится, поскольку захват отладочного прерывания (необходимого для поддержания жизнедеятельности руткита) осуществляется по той же самой схеме, что и перехват прерывания INT 80h, используемого в качестве гейта системных вызовов, методика контроля целостности хорошо отработана.

Нам обещали руткит, который нельзя обнаружить. На самом же деле, он (как на концептуальном уровне, так и на уровне отдельно взятой реализации) обнаруживается элементарно, более того, требует, чтобы в системе отсутствовал отладчик уровня ядра, с которым он жестоко конфликтует.

Нам обещали, что все вышеперечисленные недостатки будут устранены в следующей версии, однако, эти обещая не подкреплены никакими доводами и научно не обоснованы. Говорить можно все, что угодно, а вот слабо описать алгоритм действий или дать ссылку на статьи или хоть какие-то работы в этой области? Linux в отношении руткитов существенно отстает от Windows, которая, как уже говорилось, начала эксперименты с отладочными регистрами еще со времен MS-DOS и до сих пор никому не удалось создать руткит существенно превосходящий своих коллег, использующих традиционные методики.

Короче говоря, умелое использование отладочных регистров _действительно_ улучшает качество руткита (уже хотя бы потому, что препятствует активной отладке, а, значит, замедляет анализ), но не так радикально, как это утверждается. К чести парней из Immunity — они всего лишь создали первую минимально рабочую реализацию руткита данного типа под Linux и выложили ее в открытый доступ вместе с исходными текстами, снабженными подробными комментариями.

Сенсацию из этого (в общем-то ничем не примечательного события) сделали не они, так что не будем возмущаться.

```
edit DR.c - Far
L:\research\DR-rootkit\DR.c                    DOS    Line      594/642   Col 35          19:20
static int __init init_DR(void)
{
    unsigned int h0x80          = 0;
    unsigned int h0x01          = 0;
    unsigned int table          = 0;
    unsigned int syscall_call   = 0;
    unsigned int sysenter_entry = 0;
    struct watch watches        = { 0, 0, 0, 0, 0, 0 };

    DEBUGLOG(("******* LOADING IA32 DR HOOKING ENGINE *******\n"));

    h0x80 = __get_int_handler(0x80);
    DEBUGLOG(("*** loader: handler for INT 128: %X\n", h0x80));

    table               = __get_syscall_table(h0x80, RETURN_SYSCALL_TABLE);
    syscall_call        = __get_syscall_table(h0x80, RETURN_SYSCALL_CALL);
    sys_table_global    = table;
    DEBUGLOG(("*** loader: syscall_table: %X\n", table));
    DEBUGLOG(("*** loader: syscall_call call *table(,eax,4): %X\n", syscall_call));

    h0x01 = __get_int_handler(0x1);
    DEBUGLOG(("*** loader: handler for INT 1: %X\n", h0x01));

    /* XXX: only for debug cleanup on unload */
    h0x01_global    = h0x01;

    /* patch the do_debug call offset in the INT 1 handler */
    __orig_do_debug = (void (*)())__get_and_set_do_debug_2_6(h0x01, \
                                (unsigned int)__my_do_debug);
    DEBUGLOG(("*** loader: INT 1 handler patched to use __my_do_debug\n"));

    __init_hook_table();
    DEBUGLOG(("*** loader: initialized hook_table\n"));

    /*
        Set a breakpoint on sycall handler in dr0 for 1 byte
    */

    /* for DR_RW_EXECUTE len has to be 0 (1 byte) (IA32_SDM_3B.pdf) */

1       2       3       4       5Print  6       7Prev   8Goto   9Video  10      11ViewHs 12
```

Рисунок 5 исходный код DR-руткита с комментариями