# мануальная терапия нетрадиционного самоконтроля

крис касперски ака мыщъх

*with enough soap, we could blow up just about anything.*

Fight Club

*с достаточным количеством мыла, можно взорвать практически что угодно, даже мыщъх'а, хотя мыщъх'а лучше не трогать, — он еще пригодится.*

к-ф Бойцовский Клуб

**если защита не будет препятствовать модификации своего собственного кода — ее немедленно хакнут, а если воспрепятствует — хакнут тем более. контроль утрачивает свою силу когда становится явным. лобовые решения здесь бесполезны! чтобы выжить, защитный механизм должен хитрить, используя все преимущества страничной организации виртуальной памяти. в этой статье автор делится с читателями передовыми идеями, выдранными из лучших защитных механизмов, которые только есть**

## *введение*

Традиционные методики контроля целостности в большинстве своем сводятся либо к подсчету контрольной суммы, либо к проверке конкретного байта в таком-то месте. Оба способа просты в реализации, но слишком гуманны. Во всех этих случаях происходит явное обращение к модифицированное ячейке, что не есть хорошо! Всякий хакер, хотя бы однажды видевший soft-ice, немедленно нажмет <Ctrl-D> и поставит бряк на модифицированный байт, чтобы узнать какая зараза его контролирует. Конечно, защитный механизм может активно противодействовать отладчику, но это будет уже перебор. Грамотная антиотладка отнимает уйму времени и усилий, а безграмотная отламывается в два счета.

Необходимо найти такой способ самоконтроля, который никак не препятствует отладке, но "волшебным" образом обходит все точки останова, даже при запуске под отладчиком-эмулятором. И таки способы есть! Рассмотрим один из них.

Для этого нам понадобится hex-редактор HTE, который можно бесплатно скачать с сервера http://hte.sourceforge.net. Это могучий и послушный инструмент, намного более функциональный чем HIEW, и к тому же распространяемый в исходных текстах, что позволило портировать его под множество платформ. Правда, набор "горячих" клавиш не совпадает с HIEW'ом, что поначалу очень раздражает (как ни крути, а многолетняя привычка к HIEW'у берет свое). Впрочем, раскладку горячих клавиш нетрудно и изменить. Однако, во избежании никому не нужной путаницы, в этой статье будет использоваться оригинальная раскладка.

## *идея*

Страничная организация памяти, используемая в 386+ процессорах, абстрагирует нас от физических адресов. Один и тот же физический регион памяти может проецироваться по нескольким виртуальным адресам, но ни отладчик, ни дизассемблер об этом даже не догадываются, поскольку опираются исключительно на виртуальные адреса.

Давайте спроецируем физическую страницу F по виртуальным адресам A и B. Тогда, при записи в страницу A, все изменения немедленно отобразятся в странице B, поскольку в действительности это одна и та же страница! Как это можно использовать на практике? Создадим две переменных var_a и var_b. Первую из них мы разместим в странице A, а вторую — в странице B. Образуется что-то вроде нуль-пространственного туннеля, пригодного для контрабандой переброски данных из одного конца программы в другой. Убийственный трюк, не правда ли?! Допустим, переменная var_a отвечает за флаг регистрации. Дизассемблер покажет множество перекрестных ссылок, ведущих к разным частям защитного кода, но… среди них не будет ни одной перекрестной ссылки на var_b, а, значит, часть проверок регистрационного кода останется незамеченной и хакер будет долго ломать голову как это так.

Самое главное — точка останова, установленная на запись/чтение переменной var_a, при обращении к переменной var_b не вызывает всплытия отладчика!

К сожалению, с прикладного уровня манипуляции со страницами невозможны, ну… практически невозможны. Небольшая лазейка все-таки есть. Возьмем PE-файл. Как известно, он состоит из секций, то есть непрерывных фрагментов произвольной длины. С каждой секцией связан ряд атрибутов: name — имя; raw offset или просто offset – физическое смещение секции в файле; raw size — размер секции на диске; virtual address или сокращенно va — адрес по которому секция проецируется в память; virtual size или v_sz — размер секции в памяти. Есть и другие атрибуты, но эти самые важные.

Весь фокус в том, что один и тот же участок файла может быть спроецирован по нескольким виртуальным адресам! Как раз то, что нам нужно! К сожалению, сразу же после проецирования, страница "забывает" о своем происхождении, то есть соорудить нуль-транспортер на основе секций у нас не получится, однако, проконтролировать целостность защитного кода мы вполне сможем, но для начала — маленький ликбез.

Имя секции может быть любым, операционная система все равно его игнорирует. А вот хакеры на нестандартные имена реагируют вполне адекватно. Чтобы не выделяться, лучше использовать имена вроде ".tsl" (Thread Local Storage — локальные данные потока) или ".rsrc" (сокращение от resource – ресурсы). Тот факт, что содержимое секции не совпадает с ее называнием, операционную систему ничуть не смущает, зато на бдительность хакера воздействует самым усыпительным образом.

Физическое смещение секции в файле должно быть кратно степени выравнивания, прописанной в заголовке файла в поле File Alignment. Линкер от Microsoft по умолчанию использует выравнивание в 1000h, а наименьшая кратность выравнивания равна 200h (и хотя Windows NT поддерживает гораздо меньшие значения в 20h и даже 10h, такой файл не сможет работать в Windows 9x, поэтому, такое выравнивание в живой природе практически не встречается).

Виртуальный адрес секции должен быть выровнен на величину Section Alignment, так же указанную в заголовке. По умолчанию она чаще всего равна 1000h или 4000h.

Если физический размер секции меньше виртуального, то секция исправно грузится в память, а оставшийся "хвост" заполняется нулями. Если виртуальный размер меньше физического, проекция секции в память автоматически расширяется до физического размера. Короче говоря, из двух размеров всегда выбирается наибольший, автоматически округлясь до ближайшего Section Aliment в большую сторону.

Виртуальный образ не может содержать никаких "дыр", другими словами, на всем своем протяжении он должен быть непрерывен. Если мы попытаемся спроецировать секцию по произвольному виртуальному адресу, операционная система нас жестоко обломает. Поэтому, виртуальные адреса секций лучше не трогать. Лучше (и безопаснее) оперировать с физическими. Так мы и поступим.

## *первые эксперименты*

Рассмотрим простейшую программу, запрашивающую пароль и контролирующую целостность своего кода. Ключевой фрагмент исходного текста может выглядеть, например, так (полный вариант можно найти на прилагаемом к журналу компакт-диске в файле demo.c):

```
#define _PSW_ "godown"                // оригинальный пароль

// начало охранной зоны
begin(){ return 0;}endA();

main(int c, char **v)
{
        int CRC=0;char buf[1024];unsigned char *a;

        // переменная должна быть статической потому, что локальные
        // переменные инициируются прямо в коде, изменяя его CRC
        static int _CRC_= -1;

        // подсчет CRC
        for(a=(unsigned char*) begin; a<(unsigned char *)endA;a++)
                CRC = (CRC + *a);

        // отладочная дверь под служебные нужды
        if ((c>1) && !strcmp(v[1],"--debug")) printf("%X\n",CRC);
```

```
        // если CRC не совпадают, выйти не прощаясь!
        // внимание! нельзя выводить никаких ругательств, иначе нас засекут
        if (CRC ^ _CRC_) return 0;

        // проверка пароля
        printf("enter password:"); gets(buf);
        if (strcmp(buf, _PSW_))
                printf("wrong passwod\n");
        else
                printf("password ok\n");

}

// конец охранной зоны
endA(){ return 0;}
```

**Листинг 1 парольная защита с традиционным самоконтролем**

Откомпилировав программу своим любимым транслятором (например, Microsoft Visual C++), запустим ее с ключом "--debug" и посмотрим какую контрольную сумму она напишет (в моем случае это 47h). Присвоим это значение переменной _CRC_ и перекомпилируем исходный текст. Конечно, оставлять отладочный механизм в теле готовой программы нехорошо, однако, в качестве демонстрационного примера такой трюк вполне сойдет. Не будем обращать внимание на то, что оригинальный пароль лежит открытым текстом и кто угодно может подсмотреть его. Сосредоточимся исключительно на механизме самоконтроля.

Дизассемблирование защитного механизма сразу же показывает тот заветный условный переход, который всем рулит. Вот он, лежит по адресу 4010B1h, см. рис. 1.

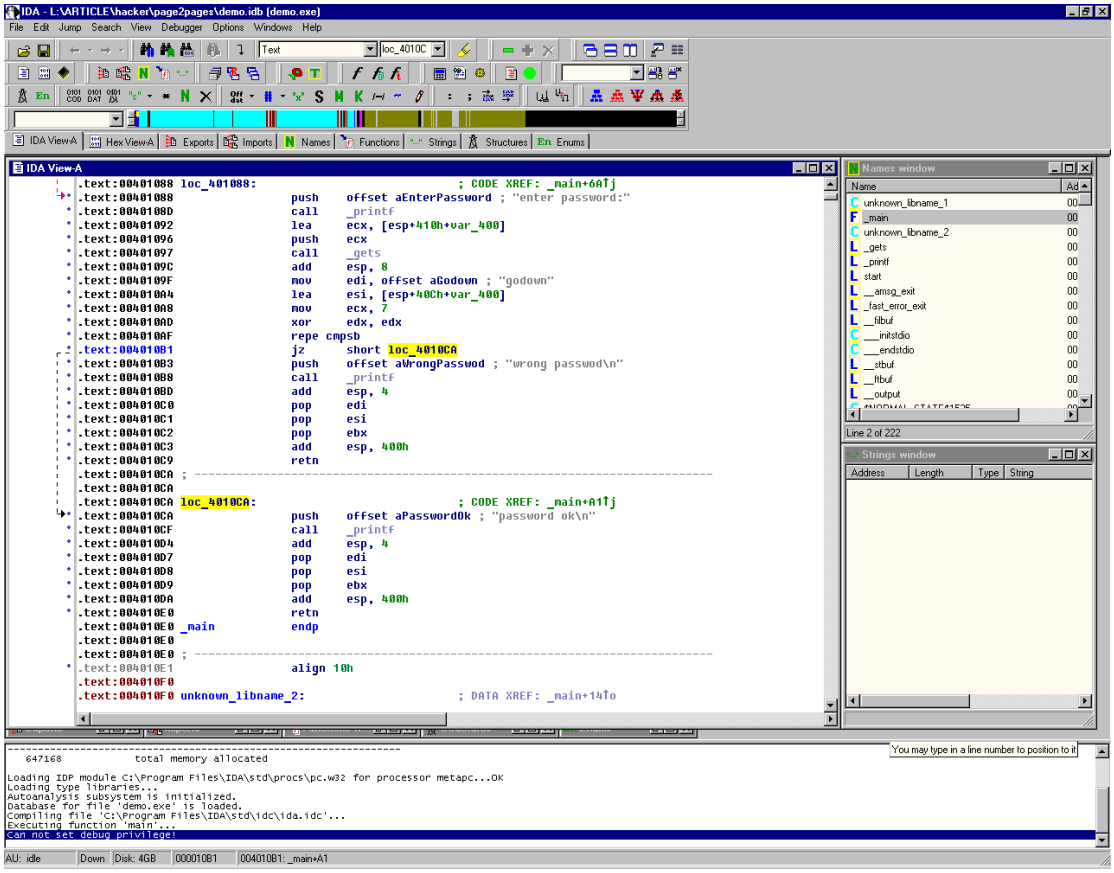


**Рисунок 1 дизассемблирование программы на предмет ее взлома**

Как видно, он ведет к той ветке кода, что выводит "password ok", поэтому если мы заменим jz loc_4010CA на jmp short loc_4010CA, любой пароль будет воспринят как правильный. Для этого необходимо заменить 74 17 (машинный код jz loc_4010CA) на EB 17 (jmp short loc_4010CA). Берем любой hex-редактор, например, HTE, создаем копию ломаемого файла (я называл ее demo.crk.exe), загружаем файл в редактор, нажимаем <F6> (mode),

выбираем режим PE-image, давим <F5> (goto), вводим адрес условного перехода (4010B1h), затем нажимаем <F4> (edit) и заменяем "74" на "EB". Клавиша <F2> сохраняет все изменения в файле, а <F10> выходит нас из редактора.

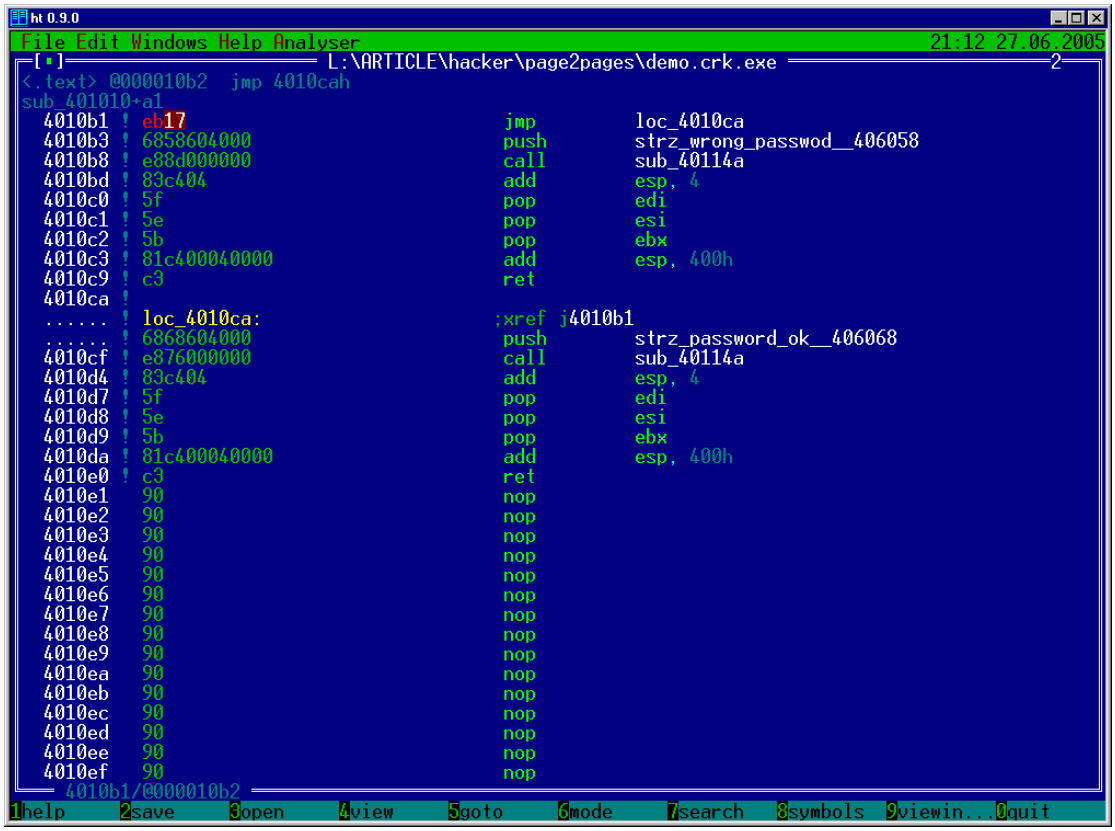


**Рисунок 2 правка условного перехода в редакторе HTE**

Запускаем хакнутую программу и… опс! Она не запускается! Так, где там наш старый добрый soft-ice? Загружаем программу в Symbol Loader, даем команду "bpm 4010B1" (установить точку доступа на обращение к ячейке 4010B1h, в которой расположен хакнутый условный переход) и по <Ctrl-D> выходим из отладчика, возвращая управление программе.

Отладчик тут же всплывает и словно волшебная лампа Алладина мгновенно переносит нас в штаб-квартиру защитного механизма, контролирующего целостность машинного кода:

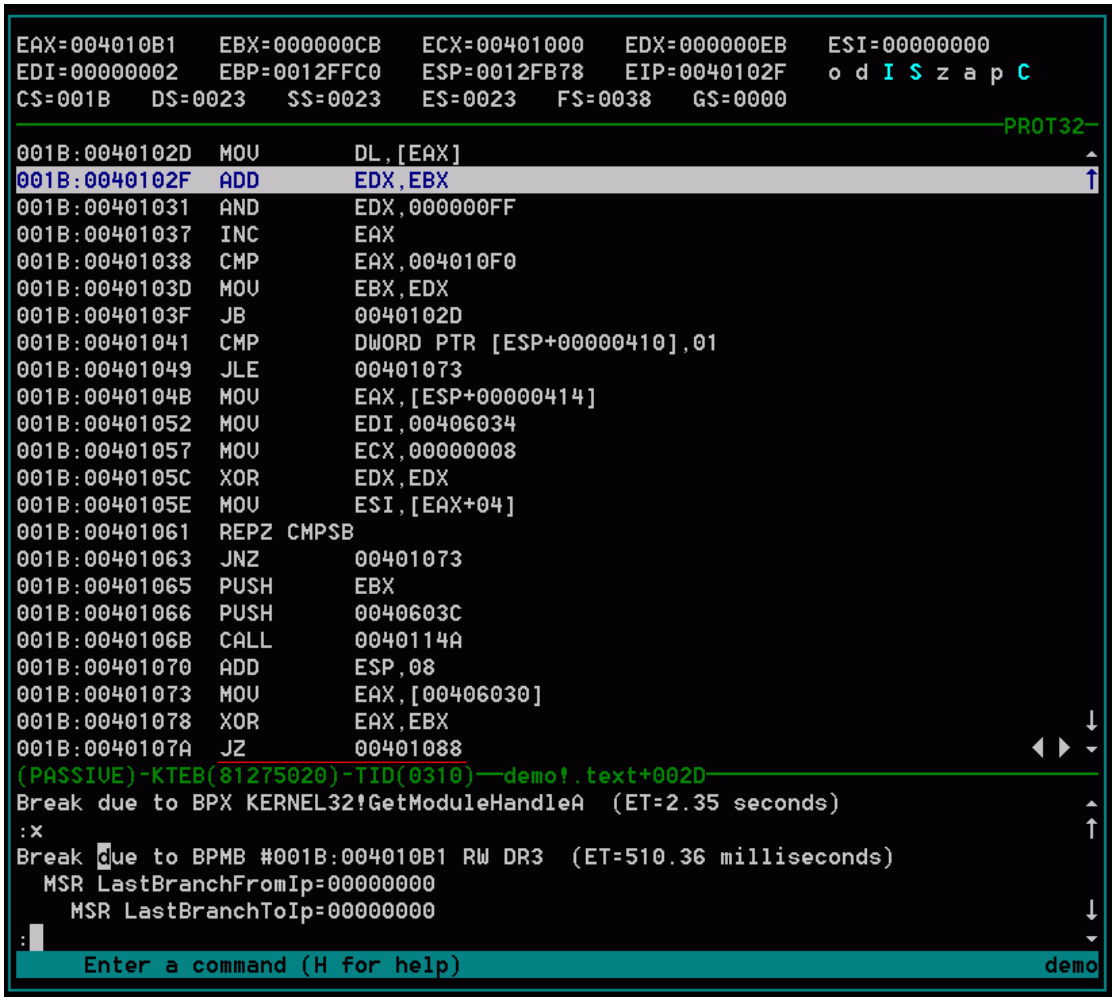


**Рисунок 3 отладчик soft-ice, обнаруживающий стандартный механизм самоконтроля в исследуемой программе**

Так-так-так, контрольная сумма накапливается в регистре EDX (MOV DL, [EAX]/ADD EDX, EBX), который тут же пересылается в регистр EBX (MOV EBX,EDX), а спустя некоторое время EBX сравнивается с какой-то ячейкой памяти (MOV EAX, [406030]/XOR EAX,EBX) и, если они идентичны друг другу, выполняется условный переход по адресу 401088h (JZ 401088). Вот этот переход и портит всю малину, препятствуя нормальной работе хакнутой программы. Если изменить XOR EAX,EBX (33 C3) на XOR EBX,EBX (33 DB или 31 DB), программа будет взломана окончательно.

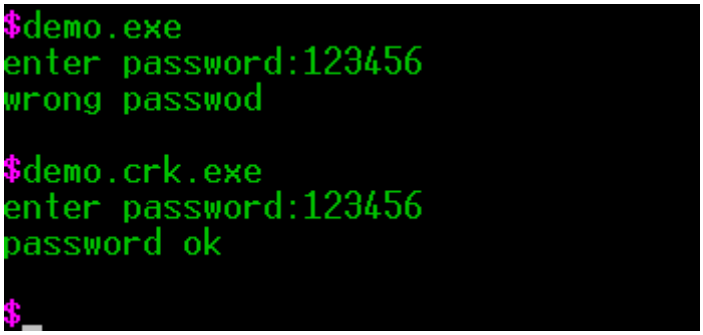


**Рисунок 4 до и после взлома**

В редакторе HTE это делается так: загружаем файл, давим <F6> (mode), выбираем pe/image, жмем <F5> (goto) и вводим адрес перехода (401078h), говорим <F4> (edit), а затем <Ctrl-A> (Assemble). Вводим "XOR EBX,EBX" и… HTE запрашивает каким именно образом мы хотим ее ассемблировать. Этой возможности нет ни у одного другого известного мне hex-

редактора! Выбрав любой вариант (оба они двухбайтные), нажимаем <F2> (save) для сохранения изменений и по <F10> выходим из редактора.

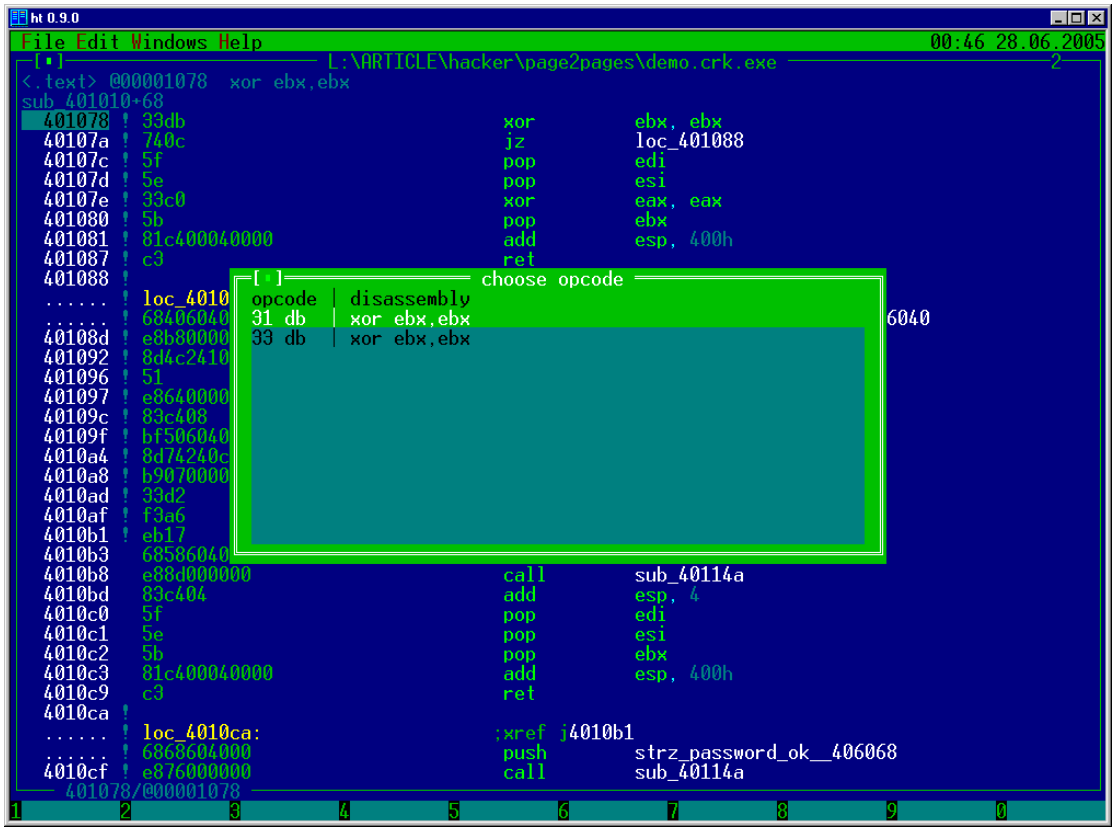


**Рисунок 5 встроенный ассемблер редактора HTE**

## *законченная реализация*

Попробуем усилить защищенность механизма самоконтроля. Добавим в начале программы следующие строки (полный вариант можно найти в файле demo.protected.c):

```
// начало нестандартной кодовую секции с именем .tsl
#pragma code_seg(".tsl")

// начало охранной зоны
begin(){ return 0;}

// фиктивная функция, чтобы секция не была пустой
demo(){}

// конец нестандартной секции
#pragma code_seg()

// конец охранной зоны
endA(){ return 0;}
```

**Листинг 2 защитный код, обслуживающий нестандартную секцию**

Прагма code_seg(имя_секции) предписывает линкеру размещать весь последующий код в секции с именем ".tsl", что он и делает. На самом деле, как мы уже говорили, это никакой не .tsl, а вполне законная секция кода, только с другим названием. Чтобы линкер не отбраковал секцию как ненужную, мы создает фиктивную функцию demo(), и окружаем ее "охранной зоной".

Прагма code_seg() отменяет действие предыдущей прагмы и весь последующий код ложится линкером в стандартную секцию .text или CODE. Поскольку, минимальный размер секции составляет 1000h (вспомним про выравнивание), то расположив endA() после code_seg(),

мы получим свое распоряжение 1000h байт. Только не перепутайте их местами, иначе ничего не получится!

Откомпилировав полученный пример, загрузим его в HTE и перейдем в режим отображения заголовка (<F6>, "PE-header"). Мы видим секцию .tls, содержащую фиктивную функцию demo, и секцию .text с подлинным кодом программы. <ENTER> распахивает содержимое атрибутов секций, а <F4> позволяет редактировать их. Очень хорошо!

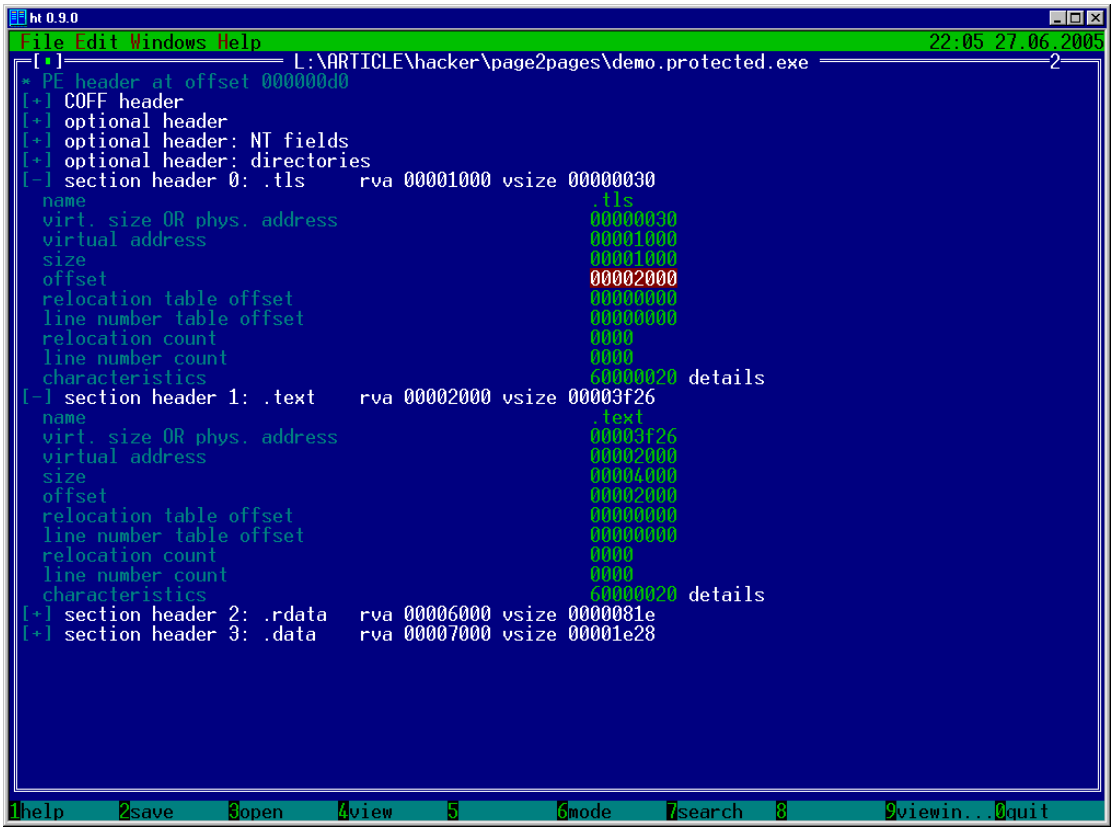


**Рисунок 6 редактирование атрибутов секции в редакторе HTE**

Секция .tls отстоит на 1000h байт от начала файла, а .text – на все 2000h. Чтобы отобразить первые 1000h байт секции .text на два региона адресного пространства необходимо изменить raw offset первой секции, передвинув ее на 1000h байт вглубь файла. Подводим курсор к строке "offset" (здесь должно быть записано 1000h), нажимаем <F4> (edit) и изменяем ее на 2000h. Сохраняемся по <F2> и выходим. Запустив отредактированный файл с ключом --debug, мы видим, что его контрольная сумма не изменилась, значит, все было сделано правильно!

Проверим — сумеем ли мы это взломать? Дизассемблер легко обнаруживает условный переход, определяющий правильность ввода пароля (в моем случае он расположен по адресу 4020A1h). Меняем jz xxx на jmp short xxx. Программа видит, что контрольная сумма изменилась и уже не срабатывает. Запускаем soft-ice и устанавливаем точку останова по адресу 4020A1h…

Черт возьми, она не срабатывает! Ну не срабатывает и все, хоть ты тресни! Хакер до отупения может перебирать все отладчики один за другим, но от этого ничего не изменится, даже если запустить программу под эмулятором. Вот так прием!

## *несколько полезных советов*

В нашем случае, охраняемый условный переход располагался в первых 1000h байтах от начала секции .text, поэтому этот трюк и сработал. Но ведь так бывает не всегда, правда? Допустим, защитный механизм расположен по адресу 403069h, что тогда?

Сначала нам необходимо определить относительный виртуальный адрес защитного механизма. Берем 403069h, вычитаем базовый адрес загрузки PE-файла (HTE показывает его в поле image base в разделе "optional header: NT fields", и в нашем случае он равен 400000h), получаем: 3069h. Округляем по величине выравнивания File Alignment в меньшую сторону. Получаем: 3000h. Это и есть та величина, которую необходимо записать в поле "offset" секции

.tls. Тогда в защитную зону попадают все байты, расположенные между адресами 3000h и 4000h.

А если нужно проконтролировать более 1000h байт? Проще всего изменить кратность выравнивания секций (у MS Link за это отвечает ключ /ALING, например, /ALING:16384 дает нам 4000h байт). Правда и размер потребляемой памяти при этом возрастает. Но кто нам запрещает создать несколько подложных секций по 1000h байт каждая? Кончено, слишком большое количество секций обязательно вызовет у хакера подозрения и он может сообразить что к чему.

В принципе, можно создать секцию данных, разместить в ней статический массив заданного размера и отобразить поверх этого массива контролируемый код, однако, тут все не так просто. Гнусный MS link и некоторые другие линкеры насильно комбинируют нестандартные секции с основной секцией данных и не дают это ограничение никак обойти. Имеются и другие проблемы, но не будет углубляться во все посторонние тонкости, а удовлетворимся тем, что есть.

## *как это ломают?*

Разумеется, предложенная защита не панацея и легко может быть взломана, особенно, если хакер с ней уже познакомился. Достаточно лишь проанализировать таблицу секций, и мы сразу обнаружим, что один и тот же участок файла отображается по нескольким виртуальным адресам.

Вернемся к нашему примеру demo.protected.exe. Загрузим программу в soft-ice и установим точку останова не на 4020A1h, а на 4010A1h. Она сработает! Хакер может обнаружить этот адрес, даже не заглядывая в таблицу секций. Достаточно посмотреть какие команды расположены в окрестностях модифицируемой ячейки и отыскать их дизассемблерном листинге. Они повторяются дважды! Ведь IDA Pro (и другие правильные дизассемблеры) эмулируют загрузку файла, что разоблачает защитный механизм с головой (см. рис 7).

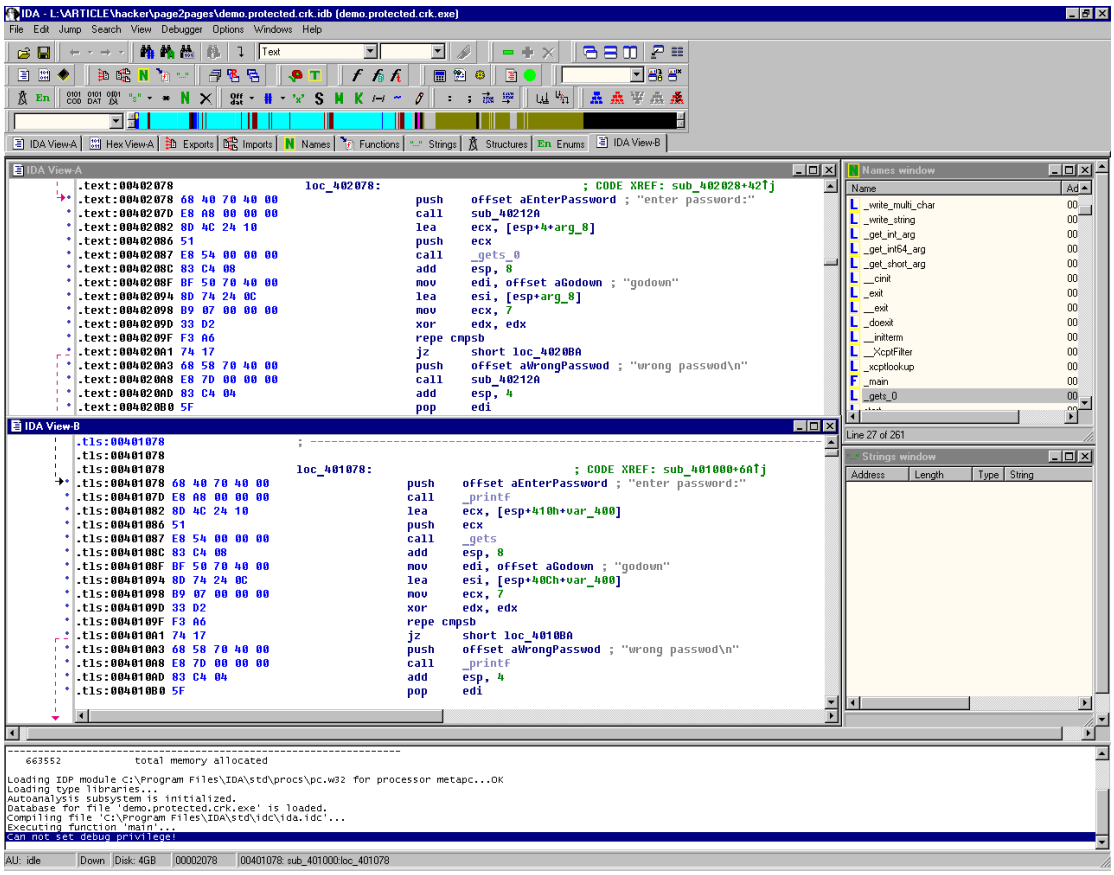


**Рисунок 7 один и те же байты расположены по различным адресам**

К тому же, защита этого типа легко обходится простым он-лайновым падчем, то есть изменением байтов не в файле, а памяти. Ведь, как уже было сказано выше, при проецировании

PE-файла в память, между "родственными" страницами утрачиваются все связи и потому изменение содержимого одной секции уже не вызывает немедленной реакции в другой. Вообще-то, он-лайновому падчу можно и противостать, поскольку чаще всего он базируется на довольно нежизнеспособной функции WriteProcessMemory из библиотеки TOOLHELP32, что интегрирована в ядро Windows 2000 и XP, но стоит ли? Если хакер всерьез разъярится, он вообще не будет править никаких байтов, а напишет генератор серийных номеров, регистрационных ключей или прочей мути, которая отличает "правильную" программу от "неправильной".

Тем не менее, написание генераторов отнимает достаточное время и на это решаются немногие. Поэтому, абсолютно бесполезным этот трюк никак не назовешь. В комбинации с другими защитными приемами его действие только усиливается, а затраты на реализацию стремятся к нулю. К тому же он абсолютно законен с точки зрения операционной системы, а это значит, что побочные эффекты практически исключены. Кстати говоря, обработка защищенной программы упаковщиками исполняемых файлов на ее самочувствие никак не влияет. Но довольно саморекламы, пускай каждый решает самостоятельно — использовать этот трюк или нет.

## *заключение*

У любого защитного механизма есть свои сильные и слабые стороны и любой из них может быть взломан. Не стоит строить иллюзий. Абсолютно надежных щитов не существует. Абсолютно надежных мечей, кстати говоря, тоже. Война с хакерами породила настоящую гонку вооружений, совершенствующую как орудия нападения, так и инструменты для отражения атак. В этой игре нет ни правил, ни ограничений. Выигрывает не тот, кто сильнее, а тот кто сумеет применить неожиданный прием, сбить с толку противника или выкинуть что-то еще.

До тех пор, пока описанная защита не станет популярной, она будет работать на благо наших программ.