on-line patching в секретах и советах

крис касперски ака мыщъх, no-email

off-line patch (он же bit-hack) это когда мы грузим программу в hiew и правим там пару байт (7xh на ECh, например). а если программа упакована? тогда у нас два пути — распаковать ее и хакнуть в off-line, или же, дождавшись завершения распаковки, модифицировать память процесса на лету, проворно обходя ловушки типа проверки CRC. вот об этом способе мы и будем говорить!

введение

Снять навороченный упаковщик/протектор чрезвычайно сложно. Качественных распаковщиков нет и приходится маньячить руками, что сильно напрягает. К тому же последние версии протекторов устраивают разные подлянки (крадут часть инструкций, внедряют p-код, эмулируют выполнение условных переходов и т. д.), в результате чего, распакованная программа работает неустойчиво и периодически падает, споткнувшись об очередную не удаленную подлянку. Поиск и удаление подлянок отнимает кучу времени и не дает никаких гарантий. Ладно, если это взлом "для себя" — взломанная программа "доводится до ума" в ходе эксплуатации. А если надо что-то срочно взломать для заказчика?

Получить дамп, пригодный для дизассемблирования (не для запуска!), относительно несложно и с этой задачей с лихвой справляется PE-TOOLS. Программы с динамической шифровкой (т. е. когда расшифровка идет небольшими порциями и отработавший свое фрагмент тут же зашифровывается вновь) обычно исследуются в отладчике.

Возлагая все надежды на навесной протектор, программисты довольно небрежно относятся к "термоядерному реактору" защитного механизма, отвечающего за контроль серийного номера, проверку кол-ва запусков, истечение испытательного строка и т. д. Большинство программ по-прежнему ломаются правкой нескольких байт, только вот… расположены эти байты глубоко под слоем упакованного кода… hiew тут уже непригоден и действовать приходится так:

запускаем ломаемый процесс на выполнение, ждем несколько секунд, чтобы все, что нужно, успело распаковаться, а затем модифицируем образ процесса прямо в памяти! Вот это и называется on-line patching'ом. Разумеется, приведенная схема далека от идеала и не учитывает ряда практических реалий, но… надо же с чего-то начинать!

простейший on-line patcher

Чтение памяти "чужого" процесса осуществляется функций `ReadProcessMemory`, а запись — `WriteProcessMemory`. Некоторые лесные сурки пишут, что нужно остановить все потоки процесса перед тем как его патчить через `SuspendThread`, а после патча возобновить их выполнение функцией `ResumeThread`. Но это не так! Патчить можно и активный процесс, но только по одной команде за раз, в противном случае возможна такая ситуация, что процесс был прерван планировщиков _между_ хакаемыми командами, а мы их заменили, при чем не факт, что границы новых команд совпадают со старыми (то есть EIP указывает на начало команды, а не в середину), в противном случае поведение ломаемой программы становится непредсказуемым и мы получаем крах, хотя вероятность этого события ничтожна мала.

Правильно делать так: остановить все потоки, а затем прочитать контекст каждого из функцией `GetThreadContext`, убедившись, что ни один из потоков в данный момент времени не выполняет хакаемый код, в противном случае необходимо либо скорректировать EIP, переустановив его на начало хакнутой команды, либо разморозить потоки и подождать еще чуть-чуть. Но, во-первых, это слишком навороченно выходит, а во-вторых, остановка/пробуждение потоков может сильно аукнуться, поскольку далеко не все программисты следят за синхронизацией.

Мы будем действовать простым, но достаточно надежным путем, срабатывающим в 99,999% случаях – запускаем процесс, ждем несколько секунд пока оно там распаковывается, читаем память активного процесса, чтобы убедиться, что по данному адресу расположено то, что нам нужно (иначе ругаемся на неверную версию ломаемой программы), и "в живую" (без всякого наркоза!) записываем сюда "исправленную" версию машинных команд.

Возьмем, например, NtExplorer от RuntimeSoftware. С помощью PEiD убедимся, что он упакован ASPack 2.11c, а, значит, прямой bit-hack невозможен. Что ж! Снимаем с программы дамп, загружаем его в дизассемблер и по перекрестным ссылкам к строке "Thank you for licensing Runtime's DiskExplorer" выходим на следующий код:

```
04E59DB        call sub_4E55B0
04E59E0        test al, al
04E59E2        jz   loc_4E5A37 ; -> облом с регистрацией
04E59E4        mov  eax, dword_582CE8
04E59E9        mov  b,[eax+10h], 1
04E59ED        mov  eax, dword_582CE8
04E59F2        call sub_4E53B8 ; запись данных в реестр
04E59F7        test al, al
04E59F9        jz   loc_4E5A15
04E59FB        push 0
04E59FD        mov  cx, word_4E5B08
04E5A04        mov  dl, 2
04E5A06        mov  eax,aThankYou;"Thank you for licensing…"
```

Листинг 1 фрагмент защитного механизма NtExporer'a

Мы видим условный переход `jz loc_4E5A37`, "шунтирующий" вывод строки об успешной регистрации. Очевидно, что забив его двумя командами NOP (если только в программе не присутствует других проверок), мы сломаем защиту и тогда любой регистрационный номер будет восприниматься как правильный.

Пишем "ломалку", алгоритм работы которой ясен из комментариев.

```
main(int c, char **v)
{
        DWORD N; STARTUPINFO si; PROCESS_INFORMATION pi; unsigned char *buf;

        // данные для патча (пример)
        unsigned char                x_old[] = {0x74,0x53}; // оригинальные байты
        unsigned char                x_new[] = {0x90,0x90}; // хакаемые байты
        void*                        x_off   = 0x04E59E2;   // арес для хака

        memset(&si,0,sizeof(si));buf=malloc(sizeof(x_old));

        // запуск процесса для взлома
        if (!CreateProcess(0,GetCommandLine()+strlen(v[0])+
                ((GetCommandLine()[0]=='\"')?3:1),0,0,0,0,0,0,&si,&pi)) return
                        printf("-ERR:run %s\x7\n",GetCommandLine()+strlen(v[0])+
                                ((GetCommandLine()[0]=='\"')?3:1));

        // ждем завершения распаковки
        for (N=0;N<69;N++) {printf("pls,wait:%c\r","-\\|/"[N%4]); Sleep(100);}

        // начинаем патчить
        printf("ok, make parch\n");

        // проверка версии ломаемой программы
        ReadProcessMemory(pi.hProcess, x_off, buf, sizeof(x_old), &N);
        if (N != sizeof(x_old)) return printf("-ERR:reading vm-memory!\x7\n");
        if (memcmp(x_old,buf,sizeof(x_old))) return printf("-ERR:incorrect ver!\x7\n");

        // падчим условный переход
        WriteProcessMemory(pi.hProcess, x_off,x_new,sizeof(x_new),&N);
        if (N != sizeof(x_new)) return printf("-ERR:writing vm-memory!\x7\n");
}
```

Листинг 2 NtExplorer.crack.c — простейшая on-line ломалка

Запускаем NtExplorer.crack.c, указав имя ломаемой программы (вместе аргументами, если они есть) в командной строке и… при этом происходит следующие. ASPack распаковывает код и передает программе управление (наша ломалка все еще ждет…), программа видит, что она ни фига не зарегистрированная, а демонстрационный строк давно истек, вот и выбрасывает диалоговое окно с требованием ввести серийный номер. К этому времени терпение у нашей ломалки кончается и пока пользователь вводит первый, пришедший ему на ум, серийный номер, `jz loc_4E5A37` успешно заменяется на NOP/NOP и при нажатии на OK, защита говорит "thanks" и продолжает выполнение программы в обычном режиме. Пользуйся — не хочу.

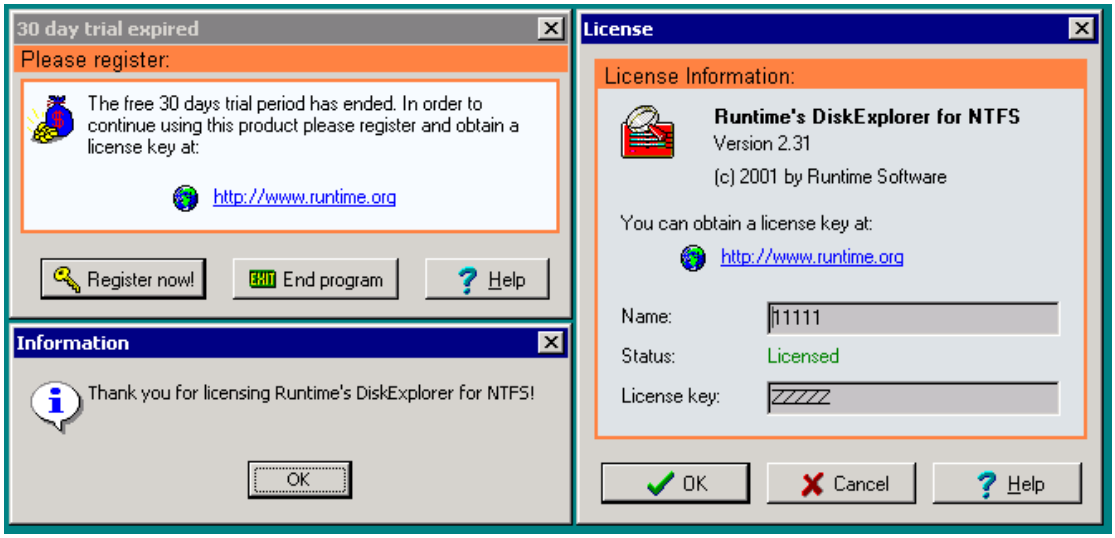

Рисунок 1 несмотря на то, что при старте программы появляется диалог, требующий регистрации, любой license key воспринимается как правильный

Естественно, при следующем запуске мерзкий диалог появится вновь, отвлекая нас от работы и заставляя вводить тупые серийные номера. А нельзя ли без этого как ни будь обойтись? Можно! И сейчас мы покажем как!

циклотрон или гонки на опережение

Продолжая исследование программы, мы обнаруживаем пару любопытных команд: `mov b,[eax+10h], 1/mov eax, dword_582CE8`, очевидно, устанавливающих флаг регистрации (что легко проверить экспериментальным путем под отладчиком).

Идея! Чтобы доломать программу окончательно, необходимо установить флаг регистрации в единицу еще _до_ того, как он будет прочитан. То есть опередить защиту! В старые время эта задача решалась пошаговой трассировкой, но теперь протекторы поумнели и просто так трассировать себя не дадут, однако, поскольку между распаковкой кода и передачей управления защите проходит какое-то время, мы вполне можем опередить защиту, если будем выполнять `ReadProcessMemory/WriteProcessMemory` в бесконечном цикле. Для надежности можно понизить приобретет ломаемого процесса, чтобы не давать ему слишком много квантов процессорного времени, однако, если слишком увлечься этим, распаковка может вообще никогда не завершиться. В большинстве случаев, для успешного взлома вообще не требуется никаких игр с приоритетами!

Вся сложность (в данном случае) в том, что местоположение флага регистрации заранее не определено. Мы знаем лишь то, что он хранится по смещению 10h от блока памяти, на который указывает двойное слово 582CE8h, инициализируемое по ходу выполнения программы. Следовательно, алгоритм наших действий будет таков: дожидаемся пока 582CE8h приобретает ненулевое значение и записываем по смещению 10h значение 01h, после чего выходим из "циклотрона" и позволяем программе продолжить свое выполнение, в заблуждении, что она успешно зарегистрирована:

```
// ждем инициализации x_off
while(!x) ReadProcessMemory(pi.hProcess,(void*)x_off,&x,sizeof(x),&N);

// ждем инициализации флага регистрации и записи результатов проверки защиты
while(count++<100)
{
        WriteProcessMemory(pi.hProcess,(void*)(x+x_idx),&foo,sizeof(foo),&N);
        Sleep(1);
}
```

Листинг 3 ключевой фрагмент NtExplorer.crack.cyclon.c

С "гонками" кода все понятно. Дождавшись совпадения хакаемого кода (что свидетельствует о завершении распаковки данной части), мы модифицируем его и отваливаем в return, поскольку никто другой модифицировать его не собирается (самомодифицирующиеся программы — не в счет, это тема для отдельного разговора).

С переменными (к которым, в частности, относятся флаги регистрации) все сложнее и они могут модифицироваться многократно. Первый раз — при конструировании объекта (если мы имеем дело с переменной-членом класса), второй раз — при явной инициализации (если только программист не забыл о ней), третий раз — при записи результатов проверки регистрационного ключа (файла, записи в реестре и т. д.). Поэтому, одного-единственного вызова `WriteProcessMemory` явно не достаточно и приходится мотать бесконечный цикл…

Цикл — дело не сложное, но слишком дурное. Неплохо бы выделить признак, что проверка регистрации уже прошла и переменная больше изменяться не будет, а, значит, ее можно не писать. Таким признаком может быть и появление главного окна программы (которое легко отследить функцией `FindWindow`), и вызов некоторой API-функции (чуть позже мы покажем как их перехватывать), и… просто время распаковки. Естественно, чем медленнее машина, тем больше ей требуется времени. В данном случае, циклу записи хватает 100 "тиков" даже при запуске NtExplorer'a под VM Ware на P-III 733 Mhz.

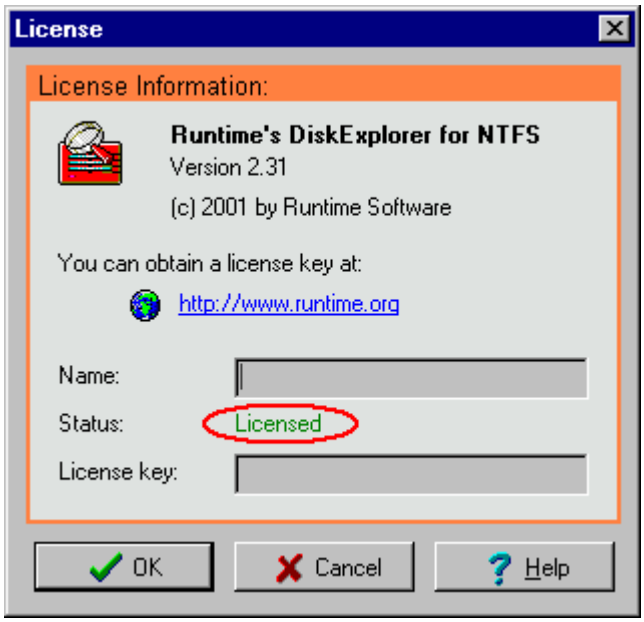

Рисунок 2 после хака флага регистрации, программа приобретает статус лицензионной даже если поля name/license key пусты

перехват API-функций как сигналинг

Сигналом к атаке, тьфу, к началу модификации может служить вызов какой-нибудь API-функции. Перехватываем функцию, вызываемую сразу же после распаковки (обычно ей становится `GetVersion`) и навешиваем на нее "сигнализатор", извещающей нас о ее вызове. Это намного надежнее и эффективнее тупого ожидания или "гонок на опережение", только следует учесть, что `GetVersion` обычно вызывается по меньшей мере дважды — первый раз из распаковщика, и второй — уже из стартового кода (start-up code) распакованной программы

Патч из стартового кода это, так сказать, хак с большого расстояния и в некоторых случаях желательно подобраться к защитному механизму как можно ближе. Для программ, защищенных ключевым файлом, хорошим решением будет перехват `CreateFileA/CreateFileW` (для 9x/NT соответственно), так же не помешает перехватить функции работы с реестром: `RegOpenKey/RegEnumKey/RegEnumValue`.

Чтобы отличить вызовы защитного механизма от всех остальных, мы можем опираться как передаваемые API-функции параметры, так и на адрес возврата. Дождавшись "своего" вызова, мы модифицируем защитный код по своему усмотрению, а в API-функции, вызываемой _после_ проверки валидности ключа, восстанавливаем все обратно. Этим мы обламываем проверки целостности, разбросанные по всей программе, гоняться за которыми нам лениво да и не фиг, когда можно просто взять и восстановить. На проверки, выполняемые _между_ вызовами API-функций, эта сентенция не распространяется и их приходится хачить вместе с остальным модифицированным кодом или… воспользоваться установкой аппаратных точек останова (см. одноименный раздел).

Алгоритм перехвата значительно упрощает тот факт, что библиотека KERNEL32.DLL во всех процессах грузится по одному и тому же адресу, а это значит, чтобы определить адрес API-функции в хакаемом процессе, достаточно определить его в своем! Оба полученных адреса будут идентичны! (В отношении остальных библиотек такой уверенности нет, USER32.DLL и GDI332.DLL _как_ _правило_ грузятся по одним и тем же адресам по всех процессах, но без 100% гарантии, а вот прикладные библиотеки могут гулять по памяти в широких пределах — все зависит от того, заняты ли базовые адреса загрузки другими библиотеками или нет).

Далее, несмотря на то, что KERNEL32.DLL проецируется на _все_ процессы, при записи внедряемого кода, соответствующие страницы памяти автоматически расщепляются и модификация затронет _только_ хакаемый процесс, никак не воздействия на все остальные (это называется "копированием при записи" — copy-on-write).

План наших действий в общих чертах выглядит так: определяем адрес выбранной API-функции в своем процессе, вызываем `VirtualAllocEx`, выделяя в хакаемом процессе блок памяти, используемый для "сигнальных" целей, запоминаем его адрес и тут же копируем его в shell-код, внедряемый в API-функцию посредством `WriteProcessMemory`, естественно, сохранив его оригинальное содержимое. Впрочем, о перехвате API-функций, мы уже неоднократно писали, так что не будем повторяться.

Рассмотрим усовершенствованный вариант нашего on-line patcher'a. Он перехватывает API-функцию `GetVersion`, внедряя на ее место shell-код следующего содержания: `inc byte ptr [p_p]/ret`, где p_p – адрес блока памяти, выделенного `VirtualAllocEx`. При каждом вызове `GetVersion` содержимое переменной p_p будет увеличиваться на единицу (оригинальное содержимое функции `GetVersion` для простоты не сохраняется) и когда оно достигнет двух, наш on-line patcher поймет, что программа распакована и пора приниматься за модификацию. Естественно, чтобы отловить этот момент, приходится непрерывно опрашивать переменную p_p, вызывая `ReadProcessMemory` в цикле, что не только некрасиво, но еще и непроизводительно. Эстеты могут воспользоваться средствами межпроцессорного взаимодействия (например, семафорами), однако, это усложнит реализацию shell-кода, но вместе с тем улучит качество on-line patcher'a.

```
unsigned char                 shell[] = {0xFE,0x05,0x56,0x34,0x12,0x00,0xC3};
                                      //  INC byte [^^^  address  ^^^]; RET
// определяем адрес GetVersion
h = LoadLibrary("KERNEL32.DLL"); p_f = GetProcAddress(h,"GetVersion");

// внедряем в программу свою переменную
p_p = VirtualAllocEx(pi.hProcess, 0, 0x1000, MEM_COMMIT, PAGE_READWRITE);

// готовим shell-код - подставляем фактический адрес переменной p_p
memcpy(&shell[2],&p_p,4);

// внедряем shell-код в программу
// здесь цикл необходим для того, чтобы дождаться момента,
// когда библиотека KERNEL32.DLL будет загружена
while (!WriteProcessMemory(pi.hProcess, p_f, shell, sizeof(shell), &N));

// ждем вызова GetVersion (непрерывный опрос переменной p_p)
// первый вызов из распаковщика,второй вызов - из самой программы
while(x<2) ReadProcessMemory(pi.hProcess, p_p, &x, sizeof(x), &N);
```

Листинг 4 фрагмент файла NtExplorer.crack-API.c, демонстрирующего patch через перехват API

аппаратные точки останова

Наилучший результат дают аппаратные точки останова, установленные на критические машинные команды/переменные защитного кода. Возвращаясь к листингу 1 — мы бы могли установить аппаратную точку по исполнению на адрес 04E59E2h (где расположена инструкция `jz loc_4E5A37`) и… вместо того, чтобы модифицировать ее, просто изменить значение регистра EIP таким образом, чтобы он указывал на следующую машинную команду, как будто условный переход не выполнялся. Тоже самое и с переменной флагом регистрации. Установить точку останова по чтению/записи и… Тогда сторожевые псы, контролирующие целостность машинного кода, ничего не смогут обнаружить! Контрольная сумма образа файла не изменится, да и сам он останется в неприкосновенности (поэтому, за такой взлом юридически очень трудно привлечь к ответственности). Красота да и только!

Подробнее о точках останова можно прочитать в руководстве Intel или в моей "технике и философии хакерских атак", копию которой можно бесплатно скачать с ftp://nezumi.org.ru. Однако, в работе с точками останова есть множество тонкостей, не отраженных в документации. Команда типа "`mov Drx, eax`" на прикладном режиме вызовет исключение, обвиняющее нас в попытке выполнить привилегированную инструкцию на ring 3. Но не спешите засаживаться за написание драйвера — отладочные регистры беспрепятственно меняются через ***контекст***! Для этого даже необязательно обладать привилегиями администратора, а отлавливать отладочные исключения можно и через SEH.

Как это осуществить на практике — показано ниже.

```
SetBreakPoint(void* p)                // установка точки останова
{
        // получаем дескриптор текущего потока
        CONTEXT ctx; HANDLE h = GetCurrentThread();

        // получаем содержимое отладочных регистров
        ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS; GetThreadContext(h, &ctx );

        // устанавливаем точку останова номер 0 по адресу p на исполнение
        ctx.Dr0 = p;
        ctx.Dr7 = ( ctx.Dr7 & 0xFFF0FFFF ) | 0x101;

        // обновляем регистровый контекст
        SetThreadContext( h, &ctx );
}

UnSetBreakPoint()                     // снятие точки останова
{
        // получаем дескриптор текущего потока
        CONTEXT ctx; HANDLE h = GetCurrentThread();

        // получаем содержимое отладочных регистров
        ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS; GetThreadContext(h, &ctx);

        // выключаем точку останова номер 0
        ctx.Dr7 = ( ctx.Dr7 & 0xFFFFFFFE );

        // обновляем регистровый контекст
        SetThreadContext( h, &ctx );
}

// функция на которую мы ставим точку останова
test(){printf("this is just a test\n");}

main()
{
        __try{
                test();               // вызываем test до установки точки останова
                SetBreakPoint(test);  // устанавливаем точку останова
                test();               // вызываем test после установки точки останова
        }

        __except(1)
        {
                printf("hello, breakpoint!\n");
                UnSetBreakPoint();    // снимаем точку останова
        }
        test();                       // вызываем test после снятия точки останова
}
```

Листинг 5 Drx.c – установка аппаратный точек останова с прикладного уровня

Устанавливать точки останова можно как в своем, так и в чужом потоке, но в последнем случае исключение поймает чужой поток, а точнее его собственный фильтр структурных исключений, который может быть переустановлен в любой момент. Навряд ли он сумеет разобраться откуда взялось это исключение и что с ним делать, поэтому нашей первой задачей будет контроль за собственным SEH-обработчиком — если ломаемая программа устанавливает новый SEH-фильтр, мы должны перекидывать наш обработчик наверх. Сделать это достаточно просто. Указатель на текущий SEH-фрейм хранится по адресу FS:[0] и нам ничего не стоит установить сюда точку останова по записи. Следует только помнить, что у каждого потока имеется свой собственный SEH, а точек останова — всего четыре. С другой

стороны, можно породить в отлаживаемом процессе своей поток (либо через `CreateRemoteThread`, вызванной из on-line patcher'a, либо с помощью `CreateThread`, вызванной из перехваченной API-функции).

Как вариант, on-line patcher может запустить ломаемую программу как отладочный процесс, получая уведомления обо всех исключениях, но протекторы страшно не любят когда их отлаживают, да и точки останова они предпочитают затирать еще в зародыше, поэтому, устанавливать их следует только на чистом коде, свободном от мин, то есть в непосредственной близости от защитного механизма, подобраться к которому позволяет перехват API-функций.

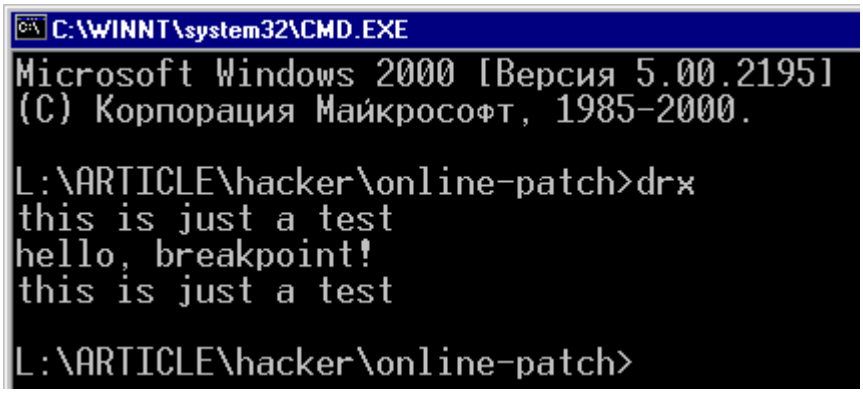

Рисунок 3 результат работы Drx.c

заключение

Мы рассмотрели основные компоненты on-line patcher'a, продемонстрировав несколько эффективных методик, и хотя до законченной "ломалки" нам еще далеко, основной фундамент уже заложен, а все остальное пытливый читатель сможет достроить и самостоятельно.