# Примеры реальных взломов (фрагмент книги "Техника и философия хакерских атак 2.000")


## Крис Касперски
## kk@sendmail.ru


В настоящей книге все атаки рассматривались исключительно на примерах, специально написанных для демонстрации того или иного алгоритма программ ‘CrackMe’. При этом многие из них были слишком искусственными и далекими от реальных защитных механизмов. Это было удобно для изложения материала, но не отражало реальных существующих защит.

Поэтому, я решил включить в приложения некоторые примеры реальных взломов. Все эти программы широко распространены и отражают средний уровень защиты условно-бесплатных программ. Заметим, что он существенно ниже чем многие из предложенных в книге реализаций.

Напоминаю, что взлом в той или иной мере конфликтует с российским и международным законодательством. Поэтому, необходимо помнить, что взлом не освобождает от регистрации и может быть использован только в образовательных целях, но никак не для получения какой-либо выгоды. В этом случае конфликтов с законодательсвом не возникнет.

# Компилятор Intel C++ 5.0.1

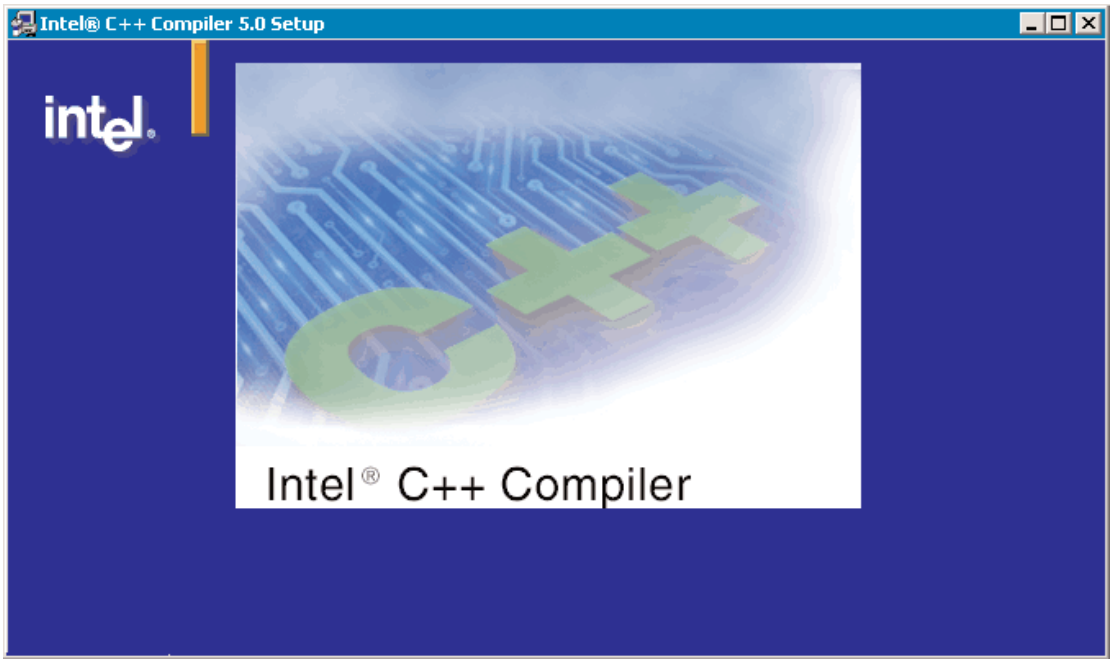


Рисунок 0x001. *Логотип компилятора Intel C++*

Прежде, чем приступать к обсуждению аспектов стойкости защиты компилятора **Intel C++ 5.0.1**, считаю своим долгом заявить, что я глубоко восхищен этим великолепным программным продуктом и ломать его, на мой взгляд, по меньшей мере кощунственно. Впрочем, сегодня только ленивый не найдет в Сети кряк (один только Google по запросу "Intel C++ crack" выдает свыше 12 тысячи ссылок!), так что никакого вреда от данной публикации не будет.

Немного грустных новостей для начала. Приобрести легальную версию данного компилятора для жителей России оказывается чрезвычайно затруднительно. И вопрос упирается даже не в то "сколько он стоит" (а стоит он, если мне не изменяет память что-то в районе тысячи долларов), — компания Intel просто игнорирует данный сегмент рынка. Обращения в российское представительство компании с просьбой предоставить (за деньги!) данный компилятор для его же описания (читай — рекламы и продвижения) в книге "Техника оптимизации программ", положительных результатов не дали. Даже после того, как к этому вопросу подключились прямо-таки скажем не мелкие отечественные издательства BHV и Солон — Р. Ладно, не хотят продавать — ну и не надо! Благо хоть с сервера компании можно свободно утянуть 30-дневный триал. Негусто, конечно, но для сравнительного тестирования — вполне достаточно (а для других целей мне этот компилятор и не нужен!).

Впрочем, все оказалось не так просто! С web-сервера компилятор за просто так не отдался, после заполнения регистрационной формы меня вежливо поблагодарили и сообщили, что сейчас ко мне на "мыло" упадет письмо с триальной лицензией и инструк-

цией по ее установке. Это "сейчас" заняло у севера аж несколько дней (такое впечатление, что анкеты просматриваются вручную). OK! Лицензия получена! Начинаем скачивать файл.... Как это так докачка не поддерживается?! А вот не поддерживается и все! Учитывая, что у меня лишь хлипкий Dial-Up по каналу в 19.200 (да и тот по межгороду), скачать полста мегабайт без единого разрыва просто нереально. К тому же, работа над книгой уже близиться к завершению и вносить в нее еще один компилятор (а значит, переписывать кучу текста заново) мне становится просто в лом. Да и Intel C++ это далеко не самый популярный в кругах российских программистов компилятор и книга без него как ни будь уж переживет (хотя, посмотреть как Intel оптимизирует код под свои процессоры очень хотелось, да и документация по компилятору вдохновляла)[1].

Разозлившись на весь свет (и на парней из Intel в частности), я отправился на ftp-сервер компании, откуда наскоро, всего за каких-то три дня, слил полнофункциональную (хотя и шибко несвежую) версию компилятора, находящуюся по следующему адресу: **ftp://download.intel.com/software/products/downloads/C5.0.1-15.exe.** (приятно, что ftp докачку исправно поддерживал и многократные разрывы никаких проблем не вызывали). Польстившись на размер, я скачал именно пятую версию компилятора, которая была в полтора раза легче шестой (под которую у меня имелась неиспользованная триальная лицензия) и аж в два раза компактнее седьмой — новейшей на момент написания этих строк — версии, ломать которую, из "политических" соображений, я все равно бы не рискнул, так зачем же ее зря качать?

Теперь, собственно, мы и подходим к известному философскому вопросу: этично ли ломать программный продукт уважаемой тобой компании или без этого можно обойтись? Да если бы без этого было возможно обойтись, я бы — честное слово — без тени сожаления выложил за этот замечательный продукт пачку вечнозеленых, но, увы... компания не проявляет ко мне как покупателю никакого интереса и, кроме как ломать, ничего другого просто не остается!

Итак, инсталлируем Intel C++ и, предварительно скопировав в просроченную лицензию от шестой версии в папку \Intel\Licenses, запускаем головной файл программы:

```
...\Program Files\Intel\C501\Compiler50\ia32\bin>icl.exe
Intel(R) C++ Compiler for 32-bit applications, Version 5.0.1   Build 010525Z
Copyright (C) 1985-2001 Intel Corporation.  All rights reserved.

icl: error: could not checkout FLEXlm license
checkout failed: No such feature exists (-5,357)
```

Как и следовало ожидать: "could not checkout **FLEX lm** license" (*"не могу проверить FLEX lm лицензию"*) — компилятор ругается и прекращает свою работу. Ага, стало быть, программа защищена FLEX'ом — достаточно известным в хакерских кругах менеджером лицензий от компании **Globetrotter Inc**, представляющим собой достаточно продвинутую защиту интегрированного типа. Разработчик защищаемого приложения получает в свое распоряжение SDK, содержащее как тривиальные функции проверки валидности ключевого файла (лицензии), так и развитые средства динамической шифровки файла. При грамотном подходе к защите запустить защищенную программу без наличия соответствующей ей лицензии доподлинно невозможно. Если часть про-

[1] Самое смешное, что когда я все-таки скачал компилятор через своих московских знакомых (ну, для Москвы 45 мегабайт это вообще ни что) он наотрез отказался работать, мотивируя свое поведение тем, что срок демонстрационной лицензии уже истек...

граммы зашифрована, то пытаться расшифровать ее без ключа — дохлое дело. Правда не факт, что парни из Intel действительно использовали шифрование, к тому же, зашифрованные фрагменты иногда удается восстановить по косвенным данным. Это смотря, что еще зашифровано!

Разумеется, при наличии триальной лицензии шифровка снимается без труда, но в том-то все и дело, что триальной лицензии у меня не было! Тем не менее надежда меня не покидала и, перекусив для смелости батоном докторской колбасы, сдобренной значительным количеством кетчупа, я запустил свой любимый дизассемблер IDA, и... не знаю у кого как, а у меня вид консольной IDA, распахнутой на весь экран, всегда вызывает чувство благоговения. OK, ну-ка посмотрим, где скрываются те текстовые строки, которые выводятся при отсутствии лицензии на экран. Результат: ни "No such feature exists", ни "could not checkout" в ASCII-строках (т. е. тех строках, что сумел распознать автоматический анализатор IDA) **не найдено**. Хорошо, зайдем с другого конца. Нажимаем <F4> для переключения в hex-режим и давим <ALT-T> для поиска текстовых строк в "сыром" виде. Что ж, на этот раз поиск "could not checkout" увенчался успехом!

```
.data1:0042D9C0  63 6F 75 6C 64 20 6E 6F-74 20 63 68 65 63 6B 6F "could not checko"
.data1:0042D9D0  75 74 20 46 4C 45 58 6C-6D 20 6C 69 63 65 6E 73 "ut FLEXlm licens"
.data1:0042D9E0  65 00 00 00 63 6F 75 6C-64 20 6E 6F 74 20 6C 6F "e...could not lo"
.data1:0042D9F0  63 61 74 65 20 46 4C 45-58 6C 6D 20 72 65 67 69 "cate FLEXlm regi"
.data1:0042DA00  73 74 72 79 20 6B 65 79-00 00 00 00 63 6F 75 6C "stry key....coul"
```

Нажимаем <F4> еще один раз для возврата в режим дизассемблера, подводим курсор к адресу 42D9C0h и нажимаем <A> для преобразования цепочки байт в ASCII-строку. В результате мы получаем:

```
.data1:0042D9C0 aCouldNotChecko db 'could not checkout FLEXlm license',0
```

А как узнать, кто же выводит строку-ругательство на экран? Нет ничего проще! Вновь переключившись в режим дизассемблера, по <F4>, давим <ALT-T> для поиска последовательности "C0 D9 40 00" — адрес строки, представленный в обратном (с учетом порядка следования старших байтов) виде. О-па! Мы видим код, наподобие следующего:

```
.data:00420CE8                  db 50h
.data:00420CE9                  db 0DEh ; ▌
.data:00420CEA                  db  42h ; B
.data:00420CEB                  db    0 ;
.data:00420CEC                  db    1 ;
.data:00420CED                  db    0 ;
.data:00420CEE                  db    0 ;
.data:00420CEF                  db    0 ;
.data:00420CF0                  db 2Ch
.data:00420CF1                  db 0DEh ; ▌
.data:00420CF2                  db  42h ; B
.data:00420CF3                  db    0 ;
.data:00420CF4                  db    2 ;
.data:00420CF5                  db    0 ;
.data:00420CF6                  db    0 ;
.data:00420CF7                  db    0 ;
```

Косвенный вызов строки! Ну, собственного, этого и следовало ожидать (иначе с чего бы это, автоматический анализатор IDA их не распознал?). Хорошо, преобразуем

двойные слова в смещения, руководствуясь при этом тем, что число "42h" должно выпадать на младший байт старшего слова (иначе адрес ссылки уйдет за диапазон предельно допустимых значений) и получаем:

```
.data:00420DE8 dd offset aCouldNotLoca_0 ; "could not locate FLEXlm registry direct"
.data:00420DEC dd 21h
.data:00420DF0 dd offset aCouldNotLocate ; "could not locate FLEXlm registry key"
.data:00420DF4 dd 22h
.data:00420DF8 dd offset aCouldNotChecko ; "could not checkout FLEXlm license"
.data:00420DFC dd 23h
```

Попробуем теперь найти ту су..., в общем тот код, что обращается к указателю (на ругательную строку) расположенному по адресу 420CE8h? Не надо спешить! По виду полученной таблицы смещений можно с уверенностью заключить, что прямого обращения к ее элементам не будет. Можно предположить, что числа, стоящие возле ссылок на строки — коды ошибок, а сами строки — соответствующие тексты сообщений. Если так, то с вероятностью близкой к единице разработчиками программы использовалась относительная адресация, т. е. для вычисления эффективного адреса элемента, ее смещение в таблицы суммируются с базовым адресом таблицы, — единственным адресом, который загружается явно.

Прокручивая экран дизассемблера вверх, мы внезапно натыкаемся на длинную последовательность нулей, интерпретируемую нами как начало таблицы:

```
.data:00420CDE                db    0 ;
.data:00420CDF                db    0 ;
.data:00420CE0 off_420CE0     dd offset unk_42DE80      ; DATA XREF: sub_403370+5E↑r
.data:00420CE4 dword_420CE4   dd 0                      ; DATA XREF: sub_403370+19↑r
.data:00420CE4                                          ; sub_403370+39↑r
.data:00420CE8                dd offset aCouldNotFindDi ; "could not find directory"
.data:00420CEC                dd 1
```

Ага! Есть две перекрестных ссылки! Это хорошо! Теперь поднимемся по ним вверх, прямиком к вызывающему их коду? Можно, конечно, поступить и так, но есть и более универсальное решение: запустив Soft-Ice, мы устанавливаем точку останова на чтение ячейки 420DE8h (если вы еще не забыли, это адрес элемента таблицы, ссылающийся на искомую ругательную строку). Теперь, кто бы к ней не обращался, Soft-Ice обязательно всплывет, и ведь действительно он всплывает! Пару раз отдав команду "P RET", поднимающую нас из дебрей глубоко вложенных процедур поближе к свету. Наконец, мы взбирается на вершину стека, и очередной "P RET" приводит к завершению программы. OK, повторяем все заново, делая на этот раз на один "P RET" меньше. Записываем любой из близлежащих адресов (пусть это будет для определенности адрес 4031C4h) и натравливаем на него IDA.

```
.text:004031C4       call    lc_checkout
.text:004031C9       test    eax, eax
.text:004031CB       jz      short loc_403215
.text:004031CD       cmp     eax, 0FFFFFFF6h
.text:004031D0       jz      loc_41B000
.text:004031D6       cmp     eax, 0FFFFFFB7h
.text:004031D9       jz      loc_41B01A
.text:004031DF
.text:004031DF loc_4031DF:                           ; CODE XREF: .text1:0041B015↓j
.text:004031DF                                       ; .text1:0041B026↓j
```

```
.text:004031DF       mov   [esp+240h+var_240], 23h
.text:004031E6       call  sub_405B00
.text:004031EB       mov   eax, dword_424C9C
.text:004031F0       mov   [esp+240h+var_240], eax
.text:004031F3       mov   [esp+240h+var_23C], offset aCheckoutFailed ; "checkout failed"
.text:004031FB       call  lc_perror
.text:00403200       mov   eax, dword_424C9C
.text:00403205       mov   [esp+240h+var_240], eax
.text:00403208       call  lc_get_errno
.text:0040320D       mov   [esp+240h+var_240], eax
.text:00403210       call  sub_405BA0
.text:00403215
.text:00403215 loc_403215:                                ; CODE XREF: sub_403000+1CB↑j
.text:00403215       mov   eax, dword_424C9C
.text:0040321A       mov   edx, dword_421E3C
.text:00403220       mov   [esp+240h+var_240], eax
.text:00403223       mov   [esp+240h+var_23C], edx
.text:00403227       call  lc_auth_data
.text:0040322C       mov   edx, eax
.text:0040322E       mov   eax, dword_424C9C
.text:00403233       call  sub_40A6F8
.text:00403238
```

Вот это да! — восклицаем мы, пришибленно уставившись на экран. Многое мы ожидали от IDA, но вот чтобы она, так запросто, представила символьные имена защитных функций, все говорящие за себя: lc_chekout, lc_perror, lc_auth_data... Черт, возьми, как? Вдохновленные смутной надеждой, мы неуверенно подгоняем курсор к lc_chekout и нажимаем на <ENTER>.

```
.idata:0041D12C ; Imports from LMGR327A.dll
.idata:0041D12C ;
.idata:0041D12C extrn __imp_lc_init:dword          ; DATA XREF: lc_init↑r
.idata:0041D130 extrn __imp_lc_expire_days:dword
.idata:0041D130                                    ; DATA XREF: lc_expire_days↑r
.idata:0041D134 extrn __imp_lc_free_job:dword      ; DATA XREF: lc_free_job↑r
.idata:0041D138 extrn __imp_lc_checkin:dword       ; DATA XREF: lc_checkin↑r
.idata:0041D13C extrn __imp_lc_auth_data:dword     ; DATA XREF: lc_auth_data↑r
.idata:0041D140 extrn __imp_lc_get_errno:dword     ; DATA XREF: lc_get_errno↑r
.idata:0041D144 extrn __imp_lc_perror:dword        ; DATA XREF: lc_perror↑r
.idata:0041D148 extrn __imp_lc_checkout:dword      ; DATA XREF: lc_checkout↑r
.idata:0041D14C extrn __imp_lc_set_attr:dword      ; DATA XREF: lc_set_attr↑r
.idata:0041D150
```

Святой Кондратий! И ***это*** они еще называют защитой?! Все защитные функции вынесены в отдельную динамическую библиотеку (наверное, чтобы взломщику разбираться было легче?) — LMGR327A.DLL, в названии которой угадывается "Library MaNaGeR", причем, это штатные функции FLEX lm, описание которых можно найти в его же SDK (хоть SDK на FLEX ln с компилятором и не поставляется, найти его в Сети — плевое дело).

Отыскав в текущем каталоге этот самый LMGR327A.DLL, мы открываем его HIEW'ов на предмет полного переписывания функции lc_checkout. Ну, насчет "переписывания" автор, ясное дело, загнул. Всего-то и требуется, — заставить lc_checkout всегда возвращать нуль, для чего первые две команды ее тела должны выглядеть приблизительно так: "XOR EAX, EAX / RETN". Записываемся, и с дрожью в сердце, запускам

icl.exe на выполнение. Критическая ошибка приложения? А чего мы хотели?! Ведь теперь функция lc_auth_data получает неверные данные и гробит все к черту. Впрочем, не будем спешить. Беглое исследование процедуры sub_40A6F8, как будто, не выявляет никаких следов шифрования и поэтому ее можно смело удалить, не забыв тоже самое, "на всякий пожарный" случай, проделать и с lc_auth_data (самое простое — впихнуть в ее начало RETN). Сохраняемся, запускам icl.exe и... компилятор работает! Все! Больше тут нечего ломать!

Самое забавное, что размер защитного механизма (413 Кб) в **два с половиной** раза превышает размер защищенной с его помощью программы (176 Кб)! Как говорится — no comment.


Рисунок 0x002. *Логотип Intel Fortran Compiler*

Ситуация с этим компилятором, в кратце, такова. В процессе работы над третьим томом "Образа мышления IDA" я исследовал большое количество компиляторов на предмет особенностей их кодогенерации и вытекающих отсюда трудностей восстановления исходного кода. Не избежал этой участи и "Intel Fortran Compiler", обнаруженный на диске "Научись сам программировать на FORTRAN". Краткая аннотация на буклете гласила "*Intel FORTRAN Compiler 4.5 — новейшая версия знаменитого компилятора. Для регистрации программы смотрите поддиректорию CRACK*". Ну, на счет "новейшего" составители диска явно приврали, т .к. на тот момент уже вышла седьмая версия, да и CRACK оказался некорректным. Вместо того, чтобы ломать защиту, он ломал сам компилятор, необратимо его гробя. К счастью, оригинальный ifl.exe на диске все-таки имелся и это давало возможность заставить работать компилятор мне самому. В конце концов, использовать в коммерческих целях этот, бесспорно, замечательный программный продукт я все равно не собирался, а для серии тестовых прогонов не то, что месяца (положенного мне по праву) даже нескольких дней было вполне предостаточно, поэтому, с этической точки зрения, ничего кощунственного я не совершал (просто мне очень уж не хотелось тянуть ~160 метров из Интернета, с моим междугородним Интернетом это действительно проблематично).

Итак, запускаем оригинальный файл компилятора на выполнение и лицезрим, как он спускает на нас Полкана (ругается в смысле):

```
KPNC$C:\Program Files\Intel\compiler45\bin>ifl1.exe >1
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error: The evaluation period has expired.

    The evaluation period for this trial version of the
    Intel(R) Fortran Compiler has expired.  For product ordering
    information, please refer to the product release notes or visit the
    Intel Developer web site at the following URL:

                http://developer.intel.com/vtune
```

Ни слова о FLEX lm! (см. "Компилятор Intel C++ 5.0.1") и файл LMGxxx.DLL отсутствует. Странно! Похоже, что Fortran Compiler защищен иначе, что, собственно, и не удивительно, поскольку их делали разные группы.

Что ж, запускаем IDA и натравливаем на нее исполняемый файл, который, кстати, занимает всего 176,128 Кб, что с точностью до байта соответствует размеру Intel C++ 5.1 Compiler. Странно! Но, как бы там ни было, ASCII-строки "The evaluation period has expired" автоматический анализатор IDA в тексте дизассемблируемого файла так и не нашел. Что ж, тогда мы сделаем это сами. <F4>, <ALT-T>, "The evaluation period" и...

```
.data1:0042A220  54 68 65 20 65 76 61 6C-75 61 74 69 6F 6E 20 70 "The evaluation p"
.data1:0042A230  65 72 69 6F 64 20 68 61-73 20 65 78 70 69 72 65 "eriod has expire"
.data1:0042A240  64 2E 0A 0A 20 20 20 20-54 68 65 20 65 76 61 6C "d.◙◙    The eval"
.data1:0042A250  75 61 74 69 6F 6E 20 70-65 72 69 6F 64 20 66 6F "uation period fo"
.data1:0042A260  72 20 74 68 69 73 20 74-72 69 61 6C 20 76 65 72 "r this trial ver"
.data1:0042A270  73 69 6F 6E 20 6F 66 20-74 68 65 0A 20 20 20 20 "sion of the◙    "
```

Теперь, вновь нажимаем <ALT-T> для поиска последовательности "20 A2 42 00" — адрес начала строки, заданный в обратном виде. Результат не заставляет себя долго ждать:

```
.data:00419390  60 A3 42 00 4F 00 00 00-20 A2 42 00 50 00 00 00 "`гB.O... вB.P..."
.data:004193A0  00 A2 42 00 51 00 00 00-E0 A1 42 00 52 00 00 00 ".вB.Q...рбB.R..."
.data:004193B0  C0 A1 42 00 53 00 00 00-A0 A1 42 00 54 00 00 00 "LбB.S...абB.T..."
.data:004193C0  60 A1 42 00 55 00 00 00-40 A1 42 00 56 00 00 00 "`бB.U...@бB.V..."
.data:004193D0  20 A1 42 00 57 00 00 00-00 A1 42 00 58 00 00 00 " бB.W....бB.X..."
```

Переключаемся обратно в дизассемблер, трижды жмем <D> для преобразования цепочки байт в двойное слово, затем <O> для перевода его в смещение и... в результате таких манипуляций получаем приблизительно такую же таблицу, как и в нашем предыдущем случае с Intel C++

```
.data:00419390 dd offset aSNoteTheEvalua ; "%s: NOTE: The evaluation period for thi"
.data:00419394 dd 4Fh
.data:00419398 dd offset aTheEvaluationP ; "The evaluation period has expired.\n\n    "
.data:0041939C dd 50h
.data:004193A0 dd offset aCommandLineErr ; "Command line error"
.data:004193A4 dd 51h
.data:004193A8 dd offset aCommandLineWar ; "Command line warning"
.data:004193AC dd 52h
```

А посему и действовать мы будем точно так же: поставим бряк на адрес 0419390h и дождемся пока отладчик не получит управления. Кстати, на счет отладчика. В момент написания этих строк у автора как раз закачивалась седьмая версия компилятора Intel C++ и от использования soft-ice пришлось воздержаться (в момент своей активации soft-ice полностью "замораживает" операционную систему, что пагубно влияет на Интернет, а точнее на установленные TCP/IP соединения). И вместо soft-ice автор решил для разнообразия использовать **Microsoft WDB**, который кстати, справился со своей задачей ничуть не хуже.

Запускам WDB на выполнение, нажимаем <Ctrl-E>, указываем имя загружаемого файла, переходим в окно команд ("Command Window") и устанавливаем точку остановка на адрес 0419398h для чего отдаем команду "BA r4 0x0419398" (что расшифровывается как: "Break on Access of Read 4 bytes long"). Затем для продолжения выполнения программы пишем "G" и с полсекунды ждем...

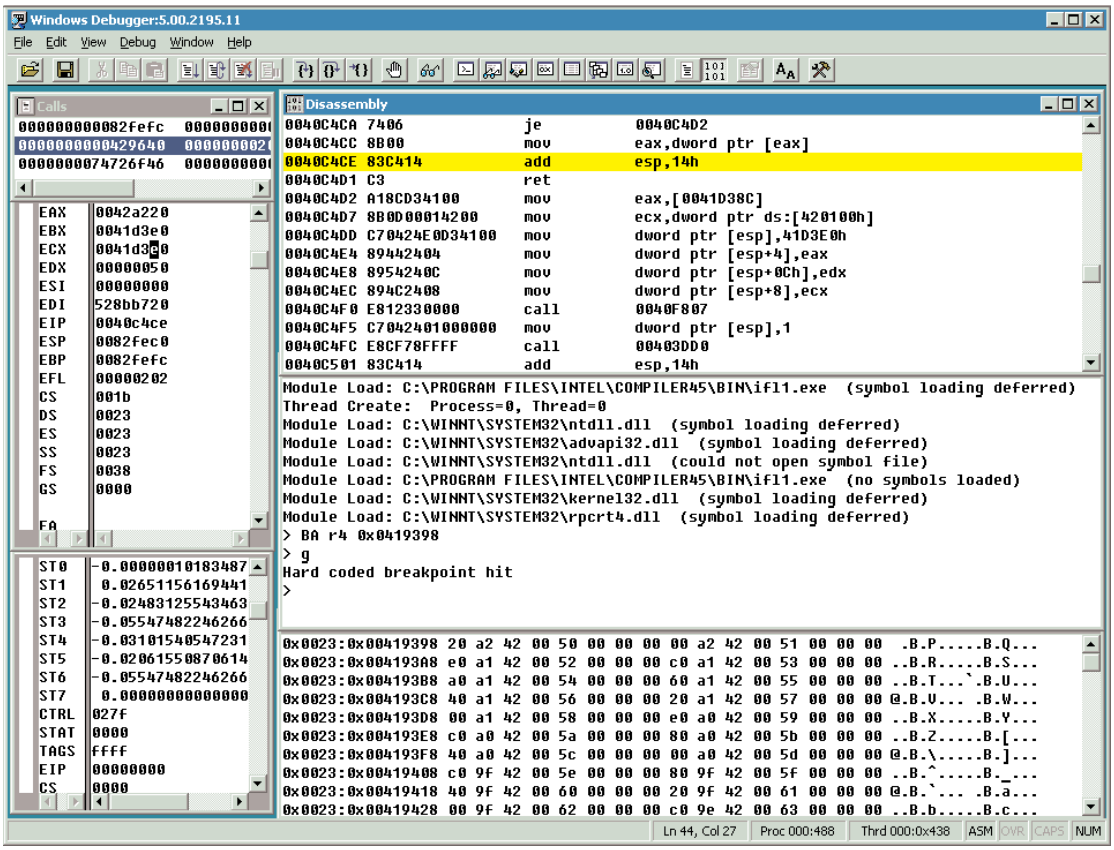


Рисунок 0x003. *Внешний вид отладчика MS WBD в процессе ломания программы*

Ага, отладчик говорит "Hard coded breakpoint hit" ("*сработала аппаратная точка останова*") и приостанавливает выполнение отлаживаемой программы. Сама же отлаживаемая программа к этому моменту уже успела вывести на экран:

```
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error:
```

Обратите внимание на строки, выделенные жирным шрифтом! Очевидно, они свидетельствуют о том, что мы попали не в самое начало защитной процедуры, а где-то в ее середину. Кстати, а что у нас там лежит на стеке? Смотрим (~View Stack, см. рис. 0x003). Всего три адреса, — довольно неглубокий уровень вложения, не так ли? Причем (обратив свой взор к окну дизассемблера), сейчас уровень вложения еще понизится, т. к. следующей командой мы выходим из этой процедуры:

```
0040C4CC 8B00            mov         eax,dword ptr [eax]
0040C4CE 83C414          add         esp,14
0040C4D1 C3              ret
```

Теперь неспешно трассируем код, попеременно поглядывая то на дизассемблированный листинг, то на консоль отлаживаемой программы. Следующая трассируемая функция (внутрь которой мы не заходим, а "заглатываем" ее одним нажатием <F10>) выводит на экран "The evolution period has expired", но не завершает программу, а продолжает ее выполнение. Что ж! Тогда и мы продолжим (трассировку)! Вызов функции

040F5FEh проходит без каких либо внешних проявлений. Так и не поняв, зачем она собственно нужна, мы поднимается на еще один уровень вверх, куда нас забрасывает завершающий функцию RET.

```
00403C7C E833880000       call        0040C4B4
; отсюда ^^^^^^^^^^^^^^^^^^^^ мы только что вышли

00403C81 89442404         mov         dword ptr [esp+4],eax
00403C85 891C24           mov         dword ptr [esp],ebx
00403C88 896C2408         mov         dword ptr [esp+8],ebp
00403C8C E8A8BB0000       call        0040F839
; эта процедура выводит "The evaluation period has expired."

00403C91 C70424C04C4200   mov         dword ptr [esp],424CC0h
00403C98 895C2404         mov         dword ptr [esp+4],ebx
00403C9C E85DB90000       call        0040F5FE
; эта процедура ничего не делает

00403CA1 83C414           add         esp,14h
00403CA4 5B               pop         ebx
00403CA5 5D               pop         ebp
00403CA6 C3               ret
```

...и таким Макаром мы трассируем код до тех пор, пока не наткнемся на следующую конструкцию:

```
0040105A E8E1800000       call        00409140 ; отсюда мы только что вышли по RETN
0040105F 0FB6C0           movzx       eax,al
00401062 85C0             test        eax,eax
00401064 0F84C4000000     je          0040112E
```

Что в ней необычного? А то, что это первая, встретившаяся нам материнская процедура, которая анализирует код возврата дочерней функции. В нашем случае регистр EAX содержит значение "ноль" и, стало быть, следующий условный переход выполняется. Но не тот ли это переход который нам нужен? Что ж, сейчас мы это узнаем — нажимаем клавишу <F10> еще несколько раз... Оп-ля! Наш условный переход перебрасывает нас на ту ветку программы, которая спустя несколько команд скоропостижно сдыхает, захлопывая окно программы. А что произойдет, если команду "JE" в строке 401064h заменить на противоположную (или, как вариант, просто удалить этот условный переход)? Пробуем...

Компилятор по прежнему смачно ругается на "evaluation expired", но... он работает! Работает!! Работает!!! По соображениям экономии экранного места (в самом деле, ругательство занимает чуть ли не половину экрана и смотрится крайне некрасиво) мы забиваем вызов процедуры 0409140h командами NOP. Проверяем — сработало ли? Ну... это смотря как посмотреть. Трехэтажный мат действительно исчез, но вот лаконичная строка "Evaluation Copy" так и осталась. Найдем что за код ее выводит? Зачем? — лучше найти саму эту строку и тем же HIEW'ом ее переписать во что ни будь более привычное, например: "hacked by mother-fucker guy". Переписываем, и... пользуемся компилятором в свое удовольствие, не забывая, однако о том, что по истечении 30-дневного срока вы будете должны его стереть, в противном случае вы поступите очень и очень нехорошо, да и незаконно.

# Intel C++ 7.0 Compiler

...компилятор Intel C++ 7.0 докачался глубокой ночью, часу где-то в пятом утра. Спать хотелось неимоверно, но и любопытство: была ли усилена защита или нет, тоже раздирало. Решив, что до тех пор пока не разберусь с защитой, я все равно не усну, я, открыв новую консоль, и переустановив системные переменные TEMP и TMP на каталог C:\TEMP, наскоро набил неприлично длинное имя инсталлятора W_CC_P_7.0.073.exe в командной строке (необходимость в установке переменных TEMP и TMP объясняется тем, что в Windows 2000 они по умолчанию указывают на очень глубоко вложенный каталог, а инсталлятор Intel C++ — да и не только он — не поддерживает путей такого огромного размера).

Сразу же выяснилось, что политика защиты была кардинально пересмотрена и теперь наличие лицензии проверялось уже на стадии установки программы (в версии 5.x установка осуществлялось без проблем). OK, даем команду dir и смотрим на содержимое того, с чем нам сейчас предстоит воевать:

```
>dir
Содержимое папки C:\TMP\IntelC++Compiler70
17.03.2003  05:10        <DIR>           html
17.03.2003  05:11        <DIR>           x86
17.03.2003  05:11        <DIR>           Itanium
17.03.2003  05:11        <DIR>           notes
05.06.2002  10:35                45 056 AutoRun.exe
10.07.2001  12:56                    27 autorun.inf
29.10.2002  11:25                 2 831 ccompindex.htm
24.10.2002  08:12               126 976 ChkLic.dll
18.10.2002  22:37               552 960 chklic.exe
17.10.2002  16:29                28 663 CLicense.rtf
17.10.2002  16:35                   386 credist.txt
16.10.2002  17:02                34 136 Crelnotes.htm
19.03.2002  14:28                 4 635 PLSuite.htm
21.02.2002  12:39                 2 478 register.htm
02.10.2002  14:51                40 960 Setup.exe
02.10.2002  10:40                   151 Setup.ini
10.07.2001  12:56                   184 setup.mwg
              19 файлов       2 519 238 байт
               6 папок      886 571 008 байт свободно
```

Ага! Программа установки setup.exe занимает всего сорок с хвостиком килобайт. Очень хорошо! В такой объем серьезную защиту навряд ли спрячешь, а если даже так этот крохотный файл ничего не стоит проанализировать целиком до последнего байта дизассемблерного листинга. Впрочем, не факт, что защитный код расположен именно в setup.exe, он может находится и в другом месте, вот например... ChkLic.dll/ChkLic.exe, занимающими в совокупности немногим менее семисот килобайт. Постой, какой такой ChkLic? Это сокращение от Check License, что ли?! Гм, у ребят из Intel, очевидно, серьезные проблемы с чувством юмора. Уж лучше бы они назвали этот файл "Hack Me", честное слово! Ладно, судя по объему, ChkLic — это тот самый FLEX lm и есть, а с ним мы уже сталкивались (см. "Intel C++ 5.0 Compiler") и приблизительно представляем как его ломать.

Даем команду "dumpbin /EXPORTS ChkLic.dll" для исследования экспортируемых функций и... крепко держимся за Клаву, чтобы не упасть со стула:

```
Dump of file ChkLic.dll

File Type: DLL

  Section contains the following exports for ChkLic.dll

           0 characteristics
    3DB438B4 time date stamp Mon Oct 21 21:26:12 2002
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 000010A0 _CheckValidLicense
```

Черт побери! Защита экспортирует всего одну-единственную функцию с замечательным именем **CheckValidLicense**. "Замечательным" потому, что назначение функции становится понятным ее названия и появляется возможность избежать кропотливого анализа дизассемблерного кода. Ну вот, отбили весь интерес... Уж лучше бы они ее по ординалу экспортировали что ли, или, по крайней мере, окрестили ее каким ни будь отпугивающим именем типа DES Decrypt.

...размечтались! Ладно, вернемся к нашим баранам. Давайте рассуждать логически: если весь защитный код сосредоточен непосредственно в ChkLic.dll (а, судя по "навесному" характеру защиты, это действительно так), то вся "защита" сводится к вызову CheckValidLicense из Setup.exe и проверке возвращенного ею результата. Поэтому для "взлома" достаточно лишь пропадчить ChkLic.dll, заставляя функцию ChekValidLicense всегда возвращать... Да, кстати, что она должна возвращать? Точнее: какое именно возвращаемое значение соответствует успешной проверке лицензии? Нет, не торопитесь дизассемблировать setup.exe для определения ведь возможных вариантов не так уже и много: либо FALSE, либо TRUE. Вы делаете ставку на TRUE? Что ж, в каком-то смысле это логично, но с другой стороны — а почему мы, собственно, решили, что функция CheckValidLicense возвращает именно флаг успешности операции, а не код ошибки? Ведь должна же она как-то мотивировать причины отказа устанавливать компилятор: файл с лицензией не найден, файл поврежден, лицензия просрочена и так далее? Хорошо, попробуем возвратить ноль, а если это не прокатит, возвратим единицу.

ОК, пристегивайтесь, поехали! Запускаем HIEW, открываем файл ChkLic.dll (если же он не открывается — трижды помянув сусликов, временно скопируем его в корневую или любую другую директорию, не содержащую в своем имени спецсимволов, которые так не нравятся hiew'у). Затем, обратившись еще раз к таблице экспорта, полученной с помощью dumpbin, определяем адрес функции CheckValidLicense (в данном случае 010A0h) и через <F5>, "10A0" переходим в ее начало. Теперь режем по "живому", перезаписывая поверх старого кода `"XOR EAX, EAX/RETN 4"`. Почему именно `"REN 4"`, а не просто "RET"? Да потому, что функция поддерживает соглашение stdcall, о чем можно узнать, взглянув в HIEW'е на ее эпилог (просто пролистывайте экран дизассемблера вниз до тех пор, пока не встретите RET).

Проверяем... Это работает!!! Несмотря на отсутствие лицензии, инсталлятор, не задавая лишних вопросов, начинает установку! Стало быть, защита пала. Ой, не верится нам, что все так просто и чтобы не сидеть, тупо уставившись в монитор в ожидании завершения процесса инсталляции программы, мы натравливаем на setup.exe свой любимый дизассемблер IDA. Первое, что бросается в глаза — отсутствие CheckValidLicense в списке импортируемых функций. Может быть, она файл ChkLic.exe как-то запускает? Пробуем найти соответствующую ссылку среди автоматически распознанных строк: "~View Names", "ChkLic"... Ага, строки "Chklic.exe" здесь вообще нет, но зато обнаруживается "Chklic.dll". Понятно, значит, библиотека ChkLic загружается явной компоновкой через LoadLibrary. И переход по перекрестной ссылке подтверждает это:

```
.text:0040175D                  push    offset aChklic_dll ; lpLibFileName
.text:00401762                  call    ds:LoadLibraryA
.text:00401762 ; загружаем ChkLic.dll ^^^^^^^^^^^^^^^^
.text:00401762 ;
.text:00401768                  mov     esi, eax
.text:0040176A                  push    offset a_checkvalidlic ; lpProcName
.text:0040176F                  push    esi             ; hModule
.text:00401770                  call    ds:GetProcAddress
.text:00401770 ; получаем адрес функции CheckValidLicense
.text:00401770 ;
.text:00401776                  cmp     esi, ebx
.text:00401778                  jz      loc_40192E
.text:00401778 ; если такой библиотеки нет, то выходим из программы установки
.text:00401778 ;
.text:0040177E                  cmp     eax, ebx
.text:00401780                  jz      loc_40192E
.text:00401780 ; если такой функции в библиотеке нет, то выходим из установки
.text:00401780 ;
.text:00401786                  push    ebx
.text:00401787                  call    eax
.text:00401787 ; вызываем функцию ChekValidLicense
.text:00401787 ;
.text:00401789                  test    eax, eax
.text:0040178B                  jnz     loc_4019A3
.text:0040178  ; если функция возвратила не ноль, то выходим из программы установки
```

Невероятно, но эта до ужаса примитивная защита построена именно так! Причем, полуметровый файл ChkLic.exe вообще не нужен! И чего ради стоило тащить его из Интернета? Кстати, если вы надумаете сохранять дистрибьютив компилятора (внимание: я не говорил "распространять"!), то для экономии дискового места ChkLic.* можно стереть — либо пропадчив setup.exe, навсегда отучив его к ним обращаться, либо же просто создав свою собственную ChkLic.dll, экспортирующую stdcall функцию CheckValidLicence вида: `int CheckValidLicence(int some_flag) { return 0;}`

Так-с, пока мы все это обсуждали, инсталлятор закончил установку компилятора и благополучно завершил свою работу. Интересно запустится ли компилятор или все самое интересное только начинается? Лихорадочно спускаемся вниз по разветвленной иерархии вложенных папок, находим icl.exe, который как и следовало ожидать, находится в каталоге bin, нажимаем <ENTER> и... Компилятор, естественно, не запускается, ссылаясь на то, что "icl: error: could not checkout FLEX lm license", без которой он не может продолжить свою работу.

Выходит, что Intel применила многоуровневую защиту и первый уровень оказался грубой защитой от дураков. Что ж! Мы принимаем этот вызов и, опираясь на свой предыдущий опыт, машинально ищем файл LMGR*.DLL в каталоге компилятора. Бесполезно! На этот раз такого файла здесь не оказывается, зато выясняется, что icl.exe сильно прибавил в весе, перевалив за отметку шестиста килобайт... Стоп! А не прилинковали ли разработчики компилятора, этот самый FLEX lm статической компоновкой? Смотрим: в Intel C++ 5.0 сумма размеров lmgr327.dll и icl.exe составляла 598 Кб, а сейчас одни лишь icl.exe занимает 684 Кб. С учетом поправки на естественное старческое "ожирение", цифры очень хорошо сходятся. Значит, все-таки FLEX lm! Ой-ой! А ведь теперь, — без символических имен функций ломать защиту будет намного труднее... Впрочем, не будем раньше времени паниковать! Давайте думать, только спокойно! Навряд ли команда разработчиков полностью переписала весь код, взаимодействующей с этой "конвертной" защитой. Скорее всего, ее "усовершенствование" одной лишь сменой типа компоновки и закончилось. А раз так, то шансы взломать программу по прежнему велики!

Памятуя о том, что в прошлый раз защитный код находится в функции main, мы, определив ее адрес, просто устанавливаем точку останова и, дождавшись всплытия отладчика, тупо трассируем код, попеременно поглядывая то на отладчик, то на окно вывода программы: не появилась ли там ругательное сообщение? При этом, все встретившиеся нам условные переходы, мы отмечаем на отдельном листке бумаги (или откладываем в своей собственной памяти, если вы так хотите), не забыв указать выполнялся ли каждый условный переход или нет... Стоп! Что-то заболтались мы с вами, а ведь ругательное сообщение уже выскочило! OK, хорошо! Посмотрим, какой условный переход ему соответствовал. Наши записи показывают, что последним, встретившимся переходом, был условный переход JNZ, расположенный по адресу 0401075h и "реагирующий" на результат, возвращенной процедурой sub_404C0E:

```
.text:0040106E                  call    sub_404C0E
.text:00401073                  test    eax, eax
.text:00401075                  jnz     short loc_40107F
.text:00401077                  mov     al, 1
.text:00401079                  mov     byte ptr [esp+40h+var_18], al
.text:0040107D                  jmp     short loc_4010BA
.text:0040107F ; ---------------------------------
.text:0040107F
.text:0040107F loc_40107F:                              ; CODE XREF: _main+75↑j
.text:0040107F                  mov     eax, offset aFfrps ; "FFrps"
.text:00401084                  mov     edx, 21h
.text:00401089                  call    sub_404C0E
.text:0040108E                  test    eax, eax
.text:00401090                  jnz     short loc_40109A
```

Очевидно, что sub_404C0E и есть та самая защитная процедура, которая осуществляет проверку лицензии на ее наличие. Как ее обхитрить? Ну, тут много вариантов... Во-первых, можно, вдумчиво и скрупулезно проанализировать содержимое sub_404C0E на предмет выяснения: что именно и как именно она проверяет. Во-вторых, можно просто заменить `JNZ short loc_40107F` на `JZ short loc_40107F` или даже NOP, NOP. В-третьих, команду проверки результата возврата `TEST EAX, EAX` можно превратить в команду установки нуля: `XOR EAX, EAX`. В-четвертых, можно пропадчить

саму sub_404C0E, чтобы она всегда возвращала ноль. Не знаю, как вы, но мне больше всех приглянулся способ номер три. Меняем два байта и запускаем компилятор. Если никаких других проверок его "лицензионности" в защите нет, то программа заработает и, соответственно, наоборот. (Как мы помним, в пятой версии таких проверок было две). Поразительно, но компилятор больше не ругается и работает!!! Действительно, как и следовало ожидать — его разработчики ничуть не усилили защиту, а, напротив, даже ослабили ее!

# Record Now

...и угораздило же меня приобрести "писец" (то бишь CD-RW) в OEM-поставке! И ведь спрашивал продавца: а где, позвольте, тут пишущий софт или, по крайней мере, драйвера? На что продавец, удивленно так пожимая плечами, ответил: какие драйвера? Втыкаете — работает. А пишущие программы подходят любые, вот купите в соседнем магазине диск с Neuro CD. Мне — обладателю retail-"писца" от PHILPS, еще тогда это показалось странным, поскольку, я хорошо помнил, что диск с драйверами в коробке PHILPS'а был, а Easy CD Creator — непосредственно сам пишущий софт, — располагался совсем на другом диске. Но ведь как-то же справляются с OEM-продукцией другие люди, подумал я, и... купил.

Наскоро воткнув новехонький 40-скоростной NEC в свой компьютер, я был немало удивлен, когда Neuro CD наотрез отказался признать его "писцом". Не помог тут и Easy CD Creator, взятый с Филечкиного CD. Провозившись битый час и, ничего ровным счетом так и не выяснив, я, зверски разозленный на продавца, решил сделать ход конем, установив NEC на компьютер с "девяносто восьмой" Windows, вернув PHILPS'а себе. Никаких изменений! Собравшись было отдавать привод назад продавцу, я неожиданно вспомнил, что в одном из последних номеров Компьютер Пресс был обзор пишущих программ, причем демонстрационные версии всех этих программ содержались на прилагаемом к журналу компакт-диске. Из всех программ NEC'овый писец опознала лишь одна: **Record NOW**, которая, к счастью не имела никаких функциональных ограничений, исключая, правда 30-днейвый триальный период. Причем, программа оказалась такой уютной и удобной, что расставаться с ней мне не захотелось, но и расставаться со своими деньгами мне не хотелось тоже.

Как выглядит защита? При каждом запуске программа выводит противный nag-screen, напоминающий сколько дней ей еще "жить" осталось, и тем самым страшно нервирующий. Хорошо, ищем фразу "Number of days remaining in evaluation" во всех файлах программы и, если наша искалка поддерживает юникод, быстро выясняется, что данный текст содержится в файле lockers.dll, открыв который любым редактором ресурсов, мы обнаруживаем в нем тот самый заветный диалог! Остается выяснить: кто же выводит этот диалог на экран? Ищем строку "lockres.dll" во всех файлах программы. OK, это lockout.dll. Да... и эти разработчики не в ладах с юмором. Запускам dumpbin и смотрим список экспортируемых функций:

```
Dump of file lockout.dll

File Type: DLL

  Section contains the following exports for lockout.dll

           0 characteristics
    3C855E8D time date stamp Wed Mar 06 03:10:53 2002
        0.00 version
           1 ordinal base
          23 number of functions
          23 number of names
```

```
ordinal hint RVA      name

      3    0 0000CFF0 ?DESDecrypt@@YAKPBDPAD0@Z
      4    1 0000CC40 ?DESEncrypt@@YAKPBDPAD0@Z
      1    2 00003520 EvalModeTest
      2    3 00003930 EvalModeTestVB
      6    4 0000B230 _ezLICENSE_Check_Delphi@16
      7    5 0000B1A0 _ezLICENSE_Check_VB@16
      9    6 0000BC20 _ezLICENSE_ChkExpire_Delphi@16
     10    7 0000BB90 _ezLICENSE_ChkExpire_VB@16
     12    8 00009DB0 _ezLICENSE_ChkFileCRC_Delphi@8
     13    9 00009D40 _ezLICENSE_ChkFileCRC_VB@8
     15    A 0000BA30 _ezLICENSE_Clear_Delphi@12
     16    B 0000B9B0 _ezLICENSE_Clear_VB@12
     18    C 0000A320 _ezLICENSE_GetRestNumber_Delphi@16
     19    D 0000A290 _ezLICENSE_GetRestNumber_VB@16
     22    E 0000A6C0 _ezLICENSE_Upgrade_Delphi@20
     23    F 0000A610 _ezLICENSE_Upgrade_VB@20
      5   10 0000B2C0 ezLICENSE_Check
      8   11 0000BCA0 ezLICENSE_ChkExpire
     11   12 00009E20 ezLICENSE_ChkFileCRC
     14   13 0000BAA0 ezLICENSE_Clear
     17   14 0000A3B0 ezLICENSE_GetRestNumber
     20   15 00009C30 ezLICENSE_GetVersion
     21   16 0000A770 ezLICENSE_Upgrade
```

Сурово! Во-первых, обращает на себя внимание пара функция DES Encrypt/DES Decrypt, что-то (как и следует из ее названия) зашифровывающая/расшифровывающая. Во-вторых, тройственный подход к наименованию функций наводит на мысль, что мы имеем дело с "конвертной" защитой, разработанной независимо от защищенной с ее помощью программы и поддерживающий все основные языки программирования: Си/Си++, Дельфи и конечно же, Visual Basic, узнаваемый по суффиксу VB. В-третьих, такое обилие всевозможных проверочных функций предвещает, что исследование защиты и защищенной программы окажется делом отнюдь не легким! Причем, в те три сотни килобайт, которые занимает файл lockout.dll можно много всяких ловушек и хитростей понапхать, так что на скорый успех нам рассчитывать не приходится. Но... глаза страшатся, а руки делают. Начнем с того, что посмотрим какие именно функции защитной библиотеки использует программа.

Вот тебе и раз! Защищенная-то программа состряпана на Визуальном Бейсике, о чем красноречиво свидетельствует единственная, явно загружаемая ею библиотека MSVBVM60.DLL! Ах, так?! Хорошо, пойдем напролом. Просто удаляем lockout.dll из каталога программы и подсовываем ей любую другую DLL, предварительно переименованную в данную. Запускаем программу. На экране незамедлительно появляется сообщение об ошибке: среда Visual Basic'а ругается, что не может найти функцию EvalModeTestVB. Что ж, это уже кое-что! Загружаем lockout.dll в дизассемблер, находим в нем эту самую "Eval", быстро выясняем что она является "переходником" к EvalModeTest, которая... которая... Ой-ой-ой, которая занимает до черта килобайт и содержит в себе крайне запутанный с большим количеством глубоко вложенных друг в друга процедур программный код. Да чтобы проанализировать все это и месяца не хватит! А кто сказал, что этот код вообще следует анализировать?! Достаточно просто подсунуть

нужный код возврата и все! Весь вопрос в том какой именно код нужный. Беглый просмотр содержимого функции показал, что существуют как минимум три различных кода возврата: "0", "2" и "3". Если это так, то скорее всего одному из них соответствует состояние "программа не зарегистрирована, но лицензия еще не истекала", "программа не зарегистрирована и лицензия уже истекла", и, наконец, "программа зарегистрирована". Что ж, на перебор трех вариантов не уйдет много времени! Взяв в руки HIEW, переписываем код защитной функции "с нуля": `XOR EAX, EAX/RETN`.

Возвращаем lockout.dll на ее прежнее место, запускаем Record NOW и... не можем поверить свои глазам — программа исправно работает! "Исправно" — в том смысле, что nag-screen уже не выводится и по истечении положенных тридцати дней писец по-прежнему живет, а не умирает.

Хорошо, а если бы разработчик защищенного приложения, не поленился бы воткнуть проверку на успешность загрузки функции EvalModeTestVB и при ее отсутствии немотивированно прекращал свою работу? Смогли бы мы тогда узнать: какие функции библиотеки lockout используется, а какие нет? Уговорили! Взломаем программу другим путем! Подгоняем курсор к MyCDPro.exe и, нажав на <F3>, пытаемся найти lockout.dll прямым контекстным поиском. Вот, пожалуйста:

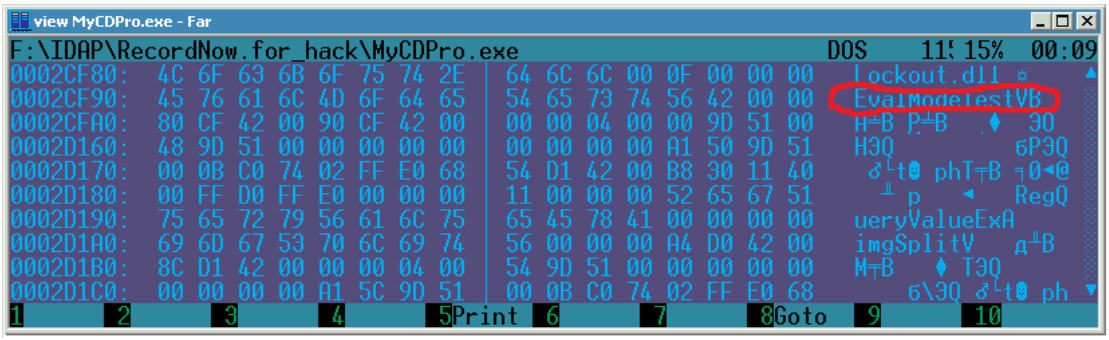


Рисунок 0x004. *Поиск ссылки на lockout.dll в защищенной программе*

Прямым текстом: "lockout.dll" и рядышком с ней EvalModeTestVB. Имена остальных защитных функций в исследуемой программе отсутствуют. Самое забавное, что в модуле lockout.dll присутствует огромное количество строк типа: "'*User has turned back their clock, so calculating days based on last and init*", "*The CRC file is valid*", "*Failed to update the Last Accessed time*", — т. е. защита составлена довольно грамотно и в состоянии как следует за себя постоять. Если, конечно, разработчик защищаемого приложения использовал все, предоставленные ей возможности, сполна. Увы, этого не произошло и на этот раз...

# UniLink v1.03 от Юрия Харона

Баста! Надоело! Все эти уродские защиты... (см. описания четырех предыдущих взломов) только портят настроение и еще, чего доброго, вызывают у читателей смутное сомнение: а не специально ли автор подобрал такие простые программы? Может быть, он, автор, вообще не умеет ничего серьезного ломать?! Уметь-то он (вы уж поверьте) умеет, но публично описывать взлом "серьезных" программ — боязно, а в "несерьезных" хороших защит мне как-то и не попадалось. Хотя, стоп! Ведь есть же такой программный продукт как UniLink, созданный опытнейшим системщиком Юрием Хароном (хорошо известным всем членам тусовки FIDO7.SU.C-CPP; если же вы никогда не заглядывали туда ранее, не поленитесь, сходите на Google, поднимите архив конференции и почитайте. Уверяю вас, вы не пожалеете). Достаточно сказать, что один лишь bag-list на UniLink — настоящий кладезь информации, перечисляющий больше количество ошибок операционной системы и ее окружения.

Наша цель — отучить UniLink ругаться на trial expired при запуске (из уважения к Харону необходимо отметить, что взлом проводится исключительно из спортивного интереса и природного любопытства. Какие либо корыстные цели тут не причем — линкер абсолютно бесплатен и может быть свободно скачан по следующему адресу: **ftp://ftp.styx.cabel.net/pub/UniLink/ulnbXXXX.zip**, где XXXX — номер версии). Цитирую со слов Харона "*Любая бета через полтора месяца начнёт "ругаться", что мол она expired :). Сделано это, просто как напоминание. в силу заинтересованности в том, что бы тестировались последнии билды*". Так что, ломая линкер помните, что взлом еще не освобождает от beta-тестирования ;-).

Несмотря на бесплатность линкера, Харон очень неплохо его защитил. Во всяком случае у меня на полный анализ защиты (включая развернутое описания взлома и отвлечения на повседневную текучку) ушла добрая неделя! Сейчас, когда пишутся эти строки, даже жалко, что защита так быстро сломалась и то интересное, во что еще можно вонзить свои зубы, закончилось. Впрочем, лучше отложим всю эту ностальгию до лучших времен в сторону, и вспомним, как эта неделя "эротических развлечений с защитой" собственно и начиналась...

...Привычным движением руки загружаем исполняемый файл линкера в свою любимую IDA 4.1.7. и... IDA грязно ругается по поводу того, что... "*can't find translation for virtual address 00000000, continue?*". Хм, ну что нам еще остается делать, — покорно жмем "Yes", чтобы сделать "continue". Увы! Наш фокус не увенчался успехом — на экране возникает еще одно ругательство "*File read error at 0004C7AC (may be bad PE structure), continue?*". Обречено жмем "Yes" и... ...IDA просто исчезает. Да-да! Именно **исчезает**, даже не успев перед смертью выдать сообщение о критической ошибке!!![2]

Интересный формат файла, однако! Пытаясь выяснить что же в нем содержится такое нехорошее, что так не понравилось IDA, мы решаем натравить на него утилиту dumpbin. Щас! Разбежались — при попытке вывести таблицу импорта, dumpbin выдает сообщение о внутренней ошибке "*DUMPBIN: error: Internal error during DumpImports*",

[2] В последующих версиях IDA это было исправлено.

и только что успев скинуть контекст, аварийно прекращает свою работу. Вот, значит, как?! Ну, защита, держись! Сейчас мы заглянем внутрь тебя "вручную" каким ни будь низкоуровневым инструментом. Ну, например, HIEW'ом...

Облом-с! При попытке сделать "prepare import data" HIEW скручивает дулю и, выдав нам на прощание трогательно красное окошко с надписью "Import name No free memory" банально виснет. Конкурирующий с ним QVIEW умирает и вовсе без каких либо пояснений. Утилита "PEDUMP" от Мэта Питтрека (известнейшего исследователя недр Windows) хоть и не виснет, но выдает сообщение о критической ошибке приложения и автоматически прибивается операционной системой. Так, чем еще можно исследовать внутренний формат PE-файла? На ум приходит **efd** (***E**xecutable **F**ile **D**umper*) от Ильфака, но даже эта утилита не справляется. — Выдав сообщение "*Can't find translation for 000002F6 (758.)*", она просто прекращает свою работу. **Dump PE** от Clive Turvey поступает аналогично. Дизассемблер **De Win** от Милюкова — виснет. **Win DASM** не виснет, но и не дизассемблирует. Даже знаменитый **PROCDUMP** распаковывать этот файл отказывается, правда позволяет сделать rebuild PE-заголовка, однако, после такой операции полученный файл становится неработоспособным. В общем, этот список можно продолжать бесконечно.

Кошмар! Защиты, срывающие крышу отладчику, — это я еще понимаю, но вот чтобы так агрессивно сопротивляться дизассемблеру! Причем, не какому-то одному, конкретно взятому дизассемблеру, а всем дизассемблерам сразу. И в это же самое время защита ухитряется работать в любой, Windows-совестимой операционной системе, включая NT и w2k, а, значит, никаких грязных хаков не использует. Харон по определению гений!

Вот мы и столкнулись с тем самым случаем, когда приходится дизассемблировать не готовым дизассемблером, а своими собственными руками и головой![3] Тяпнув для храбрости пивка, запускаем Иду и загружаем нашего подопытного в ***бинарном режиме***, то есть без анализа заголовков файла. Файл, естественно, успешно загружается. Теперь, открываем свой MSDN на странице "**Microsoft Portable Executable and Common Object File Format Specification**" и вдумчиво читаем все, что там написано. Без четкого представления о структуре и порядке загрузки PE-файлов, Харонову защиту нам ни за что не сломать. Если чтение фирменных спецификаций вызывает проблемы, попробуйте обратится к сторонним источникам. В том же MSDN содержится масса статей, посвященных исследованию PE-формата, в частности: "**The Portable Executable File Format from Top to Bottom**" by Randy Kath, русский перевод которой ("Исследование переносимого формата исполнимых файлов сверху вниз") легко найти в Сети. На худой конец можно обойтись и одним лишь заголовочным файлом WINNT.H, входящим в штатный комплект поставки любого windows-компилятора (но разобраться с "голым" WINNT.H сумеет лишь гений!)

Наша задача состоит в том, чтобы вручную проанализировать все заголовки, все секции и все поля исследуемого файла, пытаясь определить: что же такого необычного есть в каждом из них. Спрашиваете: "*необычное*" — это вообще как? Навскидку

[3] Вообще-то, анализировать PE-заголовок руками я ринулся чисто с перепугу. Тот же EXEVIEW от Randy Kath пусть и не совсем корректно обрабатывает защищенный файл, но по крайней мере не виснет и не завершает свою работу. К тому же он распространяется вместе с исходниками (см. MSDN) и у нас есть возможность оперативно исправить баг.

можно предположить по крайней мере три варианта: а) защита использует документированные, но малоизвестные возможности PE-файлов, не поддерживаемые распространенными дизассемблерами; б) защита использует недокументированные особенности (и/или поля) PE-файлов, не поддерживаемые дизассемблерами, но корректно обрабатываемые операционной системой; в) разночтения спецификаций PE-формата привели к тому, что разработчики ОС трактовали отдельные поля заголовков по-своему, а разработчики дизассемблеров — по-своему, в результате чего появилась возможность создать такой извращенный файл, корректно загрузить который сумеет одна лишь система, а все остальные исследовательские программы конкретно обломаются на его анализе.

Из пункта "а" со всей очевидностью следует, что для анализа защищенного файла одной лишь документации явно недостаточно, ведь нам требуется не только убедиться в соответствии всех полей исследуемого файла фирменной спецификации, но и выяснить насколько эти поля вообще типичны. Другими словами нам необходим практический опыт работы с PE-файлами, а если его нет, — что ж, возьмите несколько заведомо неизвращенных PE-файлов и основательно проштудируйте их от пола до потолка.

С пунктом "б" справится сложнее. Допустим, в фирменной спецификации такое-то поле помечено как неиспользуемое, а в защищенном файле здесь прописано некоторое значение. Как быть? (Дизассемблировать загрузчик операционной системы не предлагать). Да очень просто! Берем hiew старой версии — той, которая ничего не знает о PE и никак его не анализирует, и перебиваем "неиспользуемое" поле нулями или любым другим значением, пришедшимся нам по вкусу. Если это не нарушит работоспособности защищенного файла, — по всей видимости это поле действительно не используется и, соответственно, наоборот.

Пункт "в" еще более сложен. Никакие прямолинейные решения тут не действуют и все, что нам остается — вдумчиво читать каждую букву исходной спецификации и... нет! не стремиться "понять" ее, а пытаться представить себе: как она вообще должна быть понята, чтобы загрузчик операционной системы работал, а дизассемблер — нет. Дайте волю своему воображению, напрягите интуицию — всех многих тонкостей PE-форматов составители документации просто не описали. С другой стороны, сами разработчики ОС данный формат не с потолка брали и по тем же самым спецификациям его и реализовывали. Задумайтесь, а как бы вы реализовали загрузку PE-файла в память? Какие бы комбинации свойств PE-файла вы могли бы использовать для его защиты?

Первое, что нам приходит в голову — инициализация некоторых критических ячеек памяти посредством добавления их адреса в таблицу перемещаемых элементов. А что, это мысль! Особенно привлекательной в этом плане выглядит таблица перемещаемых элементов из old exe — заглушки, расположенной перед PE-файлом и большинством дизассемблеров просто игнорируемой. Но обращает ли системный загрузчик внимание на эти элементы или нет, — вот ведь в чем вопрос! Хорошо, давайте посмотрим на восстановленный old exe заголовок, извлеченный нами из защищенного файла.

```
seg000:00000000 ; OLD EXE HEADER
seg000:00000000   cc                    db 'MZ'
seg000:00000002   e_cblp                dw 405
seg000:00000004   e_cp                  dw 1
seg000:00000006   e_crlc                dw 0
seg000:00000008   e_cparhdr             dw 4
```

```
seg000:0000000A   e_minalloc              dw 33
seg000:0000000C   e_maxalloc              dw 33
seg000:0000000E   e_ss                    dw 16h
seg000:00000010   ccaaa                   dw 512
seg000:00000012   e_csum                  dw 0
seg000:00000014   e_ip                    dw 106
seg000:00000016   e_cs                    dw 0
seg000:00000018   e_lfarlc                dw offset RelocationTable
seg000:0000001A   e_ovno                  dw 0
seg000:0000001C   ae_res                  db 'UniLink!'
seg000:00000024   e_OEMid                 dw 0
seg000:00000026   e_OEMinfo               dw 1
seg000:00000028   e_res2                  db 14h dup(0)
seg000:0000003C   e_lfanew                dd offset IMAGE_NT_SIGNATURE_PE ; "PE"
```

Баста карапузики! Нас обломали! Никаких перемещаемых элементов в DOS-заглушке нет, о чем поле e_ovno красноречиво и свидетельствует (в дизассемблерном листинге оно выделено жирным шрифтом). Да и во всех остальных отношениях, old exe заголовок выглядит вполне корректным и приличным. Ладно, лиха беда начало! Отталкиваясь от значения поля e_lfanew, переходим по содержащемуся в нем смещению на заголовок PE-файла.

```
seg000:00000198 ; NEW EXE HEADER
seg000:00000198 IMAGE_NT_SIGNATURE_PE db 'PE',0,0       ; DATA XREF: seg000:0000003C
seg000:0000019C Machine               dw 14Ch           ; IMAGE_FILE_MACHINE_I386
seg000:0000019E NumberOfSection       dw 3              ; три секции
seg000:000001A0 TimeDateStamp         dd 3D4EE158h      ; временная метка
seg000:000001A4 PointerToSymbolTable  dd 0              ; указатель на таблицу символов
seg000:000001A8 NumberOfSymbols       dd 0              ; кол-во символов ноль, т.е. нет
seg000:000001AC SizeOfOptionalHeader  dw 0C0h           ; размер опционального заголовка
seg000:000001AC ;            а вот это  ^^^^  уже интересно: зная, за концом опциональ-
seg000:000001AC ; ного заголовка сразу же следуют заголовки сегментов, пытаемся проверить
seg000:000001AC ; корректность этого поля "на глаз": складываем 0x1B0 (начало опциональ-
seg000:000001AC ; ного заголовка) с 0xC0 (указанный размер заголовка) и получаем 0x270.
seg000:000001AC ; смотрим - по этому смещению в файле расположено слово ".text", значит,
seg000:000001AC ; размер заголовка указан правильно. Но… в то же самое время 0xC0 - это
seg000:000001AC ; крайне нетипичный размер для опционального заголовка и все исследуемые
seg000:000001AC ; мной файлы, содержали совсем другое значение, - а именно 0xE0. за счет
seg000:000001AC ; чего же "наш" заголовок оказался меньше? очевидно, защищенный файл со-
seg000:000001AC ; держит урезанный массив data directory, что теоретически должно воспри-
seg000:000001AC ; ниматься всеми дизассемблерами нормально, но вот полной увечности у нас
seg000:000001AC ; в этом нет. Как быть? Представляется логичным найти (или создать) PE-
seg000:000001AC ; файл с урезанной data directory и натравить на него дизассемблер (ту
seg000:000001AC ; же IDA) - интересно зависнет он или нет? А вот как создать такой файл,
seg000:000001AC ; не имея под руками соответствующего линкера? Просто пропадчить заголо-
seg000:000001AC ; вок в готовом PE-файле нельзя, т. к. за концом data directory загрузчик
seg000:000001AC ; ожидает увидеть каталог сегментов, а при "искусственном" уменьшении
seg000:000001AC ; размера заголовка там окажется "хвост" от data directory, что приведет
seg000:000001AC ; дизассемблер в сильное замешательство. "вырезать" кусочек data directo-
seg000:000001AC ; ry из файла так же невозможно, ведь при этом посыплются все смещения,
seg000:000001AC ; что так же приведет к непредсказуемой реакции дизассемблера при попыт-
seg000:000001AC ; ке анализа такого файла. А если… Постойте-ка! ведь можно просто сдви-
seg000:000001AC ; нуть каталог сегментов на место "освободившихся" после усечения заго-
seg000:000001AC ; ловка элементов data directory?! а знаете, это должно сработать! OK,
seg000:000001AC ; вооружившись hiew'ом усекаем размер заголовка любого заведомо нормаль-
```

```
seg000:000001AC ; ного файла до 0xC0 и перемещаем каталог сегментов на 0x20 байт "вверх".
seg000:000001AC ; Запускаем сам файл. Работает? Работает! Загружаем файл в дизассемблер…
seg000:000001AC ; Работает!!! OK, значит, размер заголовка в 0xC0 действительно допустим
seg000:000001AC ; продолжаем анализ….
seg000:000001AE Characteristics       dw 30Fh   ; IMAGE_FILE_RELOCS_STRIPPED|
seg000:000001AE                                 ; IMAGE_FILE_EXECUTABLE_IMAGE|
seg000:000001AE                                 ; IMAGE_FILE_LINE_NUMS_STRIPPED|
seg000:000001AE                                 ; IMAGE_FILE_32BIT_MACHINE |
seg000:000001AE                                 ; IMAGE_FILE_DEBUG_STRIPPED
seg000:000001AE ; атрибуты файла несколько нетипичны. обычно встречается 0x10F, а не 0x30F
seg000:000001AE ; (т.е. в нормальных файлах отсутствует флаг  IMAGE_FILE_DEBUG_STRIPPED
seg000:000001AE ; даже когда они не содержат никакой отладочной инфы), но с другой сто-
seg000:000001AE ; роны, так даже и правильнее. Эксперименты показывают, что исправление
seg000:000001AE ; 0x10F на 0x30F в остальных файлах (ес-но без дебужной инфы) проходит
seg000:000001AE ; безболезненно, значит, собака зарыта не здесь
```

Вот мы и выяснили, что PE-заголовок защищенного файла не содержит абсолютно ничего интересно, и если кто и завешивает HIEW и срывает IDA крышу, то уж точно не он. Что ж, сделав короткий перерыв (для "пивка"), продолжим наше утомительное исследование формата PE-файла, на сей раз взявшись за так называемый ***опциональный заголовок*** (***optional header***), следующий за концом PE-заголовка.

```
seg000:000001B0 ; ОПЦИОНАЛ ХИДЕР
seg000:000001B0 ; ==============
seg000:000001B0 Magic                   dw 10Bh         ; NORMAL EXE (все OK)
seg000:000001B2 MajorLinkerVersion      db 1            ; версия линкера
seg000:000001B3 MinorLinkerVersion      db 3            ; версия линкера
seg000:000001B4 SizeOfCode              dd 49817h       ; размер кода
seg000:000001B4                                         ; выглядит вполне нормально.
seg000:000001B4                                         ; т. е. при длине exe-файла в
seg000:000001B4                                         ; 0x4C7AA байт, потребности в
seg000:000001B4                                         ; 0x49817 байт вполне
seg000:000001B4                                         ; удовлетворяются
seg000:000001B4
seg000:000001B8 SizeOfInitializedData   dd 3008h        ; размер секции
seg000:000001B8                                         ; инициализированных данных
seg000:000001B8                                         ; выглядит вполне нормально
seg000:000001B8
seg000:000001BC SizeOfUninitializedData dd 0            ; нет секции
seg000:000001BC                                         ; неинициализированных данных
seg000:000001C0 AddressOfEntryPoint     dd 46673h       ; адрес точки входа
seg000:000001C4 BaseOfCode              dd 1000h        ; базовый адрес сегмента кода,
seg000:000001C4                                         ; забегая вперед, отметим,
seg000:000001C4                                         ; что этот адрес в точности равен
seg000:000001C4                                         ; адресу сегмента .text, так что
seg000:000001C4                                         ; тут все законно
seg000:000001C4
seg000:000001C8 BaseOfData              dd 4B000h       ; базовый адрес сегмента данных,
seg000:000001C8                                         ; проверка подтверждает его
seg000:000001C8                                         ; корректность
seg000:000001C8
seg000:000001CC ImageBase               dd 400000h      ; image base абсолютно нормальный
seg000:000001D0 SectionAlignment        dd 1000h        ; выравнивание секций по границе
seg000:000001D0                                         ; в 4Кб, что OK
seg000:000001D0
```

```
seg000:000001D4 FileAlignment           dd 200h         ; выравнивание файла по границе
seg000:000001D4                                         ; в 512 байт, что OK
seg000:000001D8 MajorSysVersion         dw 4            ; версия требуемой системы, OK
seg000:000001DA MinorSysVersion         dw 0            ; OK
seg000:000001DC MajorImageVersion       dw 1            ; версия приложения, OK
seg000:000001DE MinorImageVersion       dw 0            ; OK
seg000:000001E0 MajorSubsystemVersion   dw 4            ; версия подсистемы, OK
seg000:000001E2 MinorSubsystemVersion   dw 0            ; OK
seg000:000001E4 Win32VersionValue       dd 0            ; OK
seg000:000001E8 SizeOfImage             dd 52000h       ; размер образа файла в памяти
seg000:000001E8                                         ; выглядит вполне достоверно
seg000:000001E8
seg000:000001EC SizeOfHeaders           dd 400h         ; размер всех заголовков, OK
seg000:000001F0 CheckSum                dd 0            ; нет контрольной суммы, OK
seg000:000001F4 Subsystem               dd 3            ; кол-во секций, OK
seg000:000001F4                                         ; (дальше мы их все найдем)
seg000:000001F4
seg000:000001F8 SizeOfStackReserve      dd 100000h      ; кол-во резервируемой памяти
seg000:000001F8                                         : под стек, OK
seg000:000001F8
seg000:000001FC SizeOfStackCommit       dd 2000h        ; кол-во выделенной под стек
seg000:000001FC                                         ; памяти, OK
seg000:000001FC
seg000:00000200 SizeOfHeapReserve       dd 100000h      ; кол-во резервируемой под кучу
seg000:00000200                                         ; памяти, OK
seg000:00000200
seg000:00000204 SizeOfHeapCommit        dd 1000h        ; кол-во выделенной под кучу
seg000:00000204                                         ; памяти, OK
seg000:00000204
seg000:00000208 LoaderFlags             dd 0            ; не используется, OK
seg000:0000020C NumberOfRvaAndSizes     dd 0Ch          ; кол-во элементов в
seg000:0000020C                                         ; IMAGE_DATA_DIRECTORY
```

...и опциональный заголовок не содержит ничего интересного, но вот IMAGE DATA DIRECTORY, расположенная за ним следом, — дело другое и буквально с третий по счету строки мы выходим на след защиты:

```
seg000:00000210 IMAGE_DATA_DIRECTORY    dd 0            ; EXPORT dir
seg000:00000214                         dd 0
seg000:00000218
seg000:00000218 Import Table
seg000:00000218                         dd offset IMPORT_TABLE ;
```

Вот она — ссылка на таблицу импорта, ту самую таблицу, которая приводит к буйному замешательству огромное количество дизассемблеров и срывает крышу всем PE-утилитам вместе взятым. Посмотрим на нее?

```
seg000:0004B000 IMPORT_TABLE            dd 94010F0Eh    ; DATA XREF: seg000:00000218-o
seg000:0004B000                                         ; flags
seg000:0004B004                         dd 4000696h     ; date start
seg000:0004B008                         dd 54414C46h    ; foward index
seg000:0004B00C                         dd offset unk_39A39
seg000:0004B010                         dd 8965410h     ; import addres
seg000:0004B014
```

Пошла вода в хату! Оказывается, в таблице импорта вместо нормальных полей содержится какой-то голимый "мусор", который кое-что проясняет. С такой таблицей импорта дизассемблеры работать просто не могут и... если проверка корректности содержимого таблицы импорта отсутствует, они — виснут, в противном же случае, — аварийно прерывают свою работу с сообщением об ошибке.

Но это совершенно не объясняет как с такой защитой ухитряется работать загрузчик операционной системы? Уж не имеем ли мы дело с некоторыми недокументированными особенностями? Или, быть может, по этим "мусорным" адресам в оперативной памяти расположено что-то особенное? Последнее навряд ли! Поскольку защита успешно функционирует во всех windows-подобных системах, представляется сомнительным, что содержимое данных адресов всегда и везде одно и то же (кстати, беглая проверка отладчиком, это допущение с треском опровергает). Недокументированные возможности? Хм, непохоже... да если так — где прикажите искать реально импортируемые адреса?! Ладно, двигаемся дальше, может быть нам и повезет...

```
seg000:00000268 ; Bound Import
seg000:00000268                         dd offset bound_import_table
seg000:0000026C                         dd 1Ch
```

Ага! Держи Тигру за хвост! Защита использует документированное, но малоизвестное поле **bound import**, — представляющее собой альтернативный механизм импорта функций из DLL. Смотрим, что у нас там...

```
seg000:000002E8 ; bound import table
seg000:000002E8 TimeDateStamp          dd 0FFFFFFFFh         ; DATA XREF: seg000:0000268
seg000:000002EC OffsetModuleName       dw 0Eh                ; относительное смещение
seg000:000002EC                                              ; строки, содержащей имя
seg000:000002EC                                              ; импортируемой DLL
seg000:000002EC                                              ; 0x2E8 + 0xE == 0x2F6
seg000:000002EC                                              ; где мы обнаруживаем
seg000:000002EC                                              ; "kernel32.dll", что
seg000:000002EC                                              ; очевидно, уже не мусор!
seg000:000002EC
seg000:000002EE NumberOfModuleForward  dw 0                  ; ничего не импортируем?!
seg000:000002F0 Reserverd              dw 0
seg000:000002F2                        dd 0
seg000:000002F6 aKernel32_dll          db 'kernel32.dll',0   ; DATA XREF: seg000:049E0C
```

Вот **это** уже явно не мусор, а вполне удобоваримая таблица импорта, загружающая динамическую библиотеку kernel32.dll, и импортирующая.... Как это так — никаких функций?! Странно... Но ведь защита все-таки работает (пусть час от часу становится все менее и менее понятно ***как***). Хорошо, давайте рассуждать логически. Программ, не импортирующих никаких функций, под Windows NT существовать в принципе не может. Даже если защита использует native API (т. е. обращается к системным функциям напрямую через прерывание 2Eh), операционный загрузчик окажется не в состоянии загрузить такое приложение, поскольку ему необходимо, чтобы на адресное пространство загружаемого процесса была спроецирована библиотека kernel32.dll. Это в Windows 9x, где системные библиотеки автоматически отображаются на адресные пространства процессов, "голые" файлы работают безо всяких проблем, а в NT, отображающий только явно загруженные библиотеки, такой фокус уже не проходит. А, знаете, это многое объясняет! Теперь становится понятно в частности почему таблица импорта

не содержит в себе ни одной функции — они просто не нужны! Ссылка на kernel32.dll присутствует лишь затем, чтобы спроецировать эту библиотеку на адресное пространство процесса, как этого требует системный загрузчик. Хорошо, но как быть с "мусором" в стандартной таблице импорта? Как ни крути, а такие извращения системный загрузчик скорее удавится, чем обработает... Увы, нам нечего ответить на этот вопрос и, скрепя сердце, его вновь приходится откладывать, надеясь, что последующий анализ отделит свет от тьмы и все расставит по своим местам...

```
seg000:00000270 ; НАЧАЛО СЕГМЕНТОВ
seg000:00000270 a_text              db '.text',0,0,0
seg000:00000278 vir_size_text       dd 49817h           ; размер секции text в памяти
seg000:0000027C virt_addr_text      dd 1000h            ; адрес проекции на память
seg000:00000280 szRawData_text      dd 49810h           ; размер в файле
seg000:00000284 pRawData_text       dd 400h             ; смещение начала секции в файле
seg000:00000288 pReloc_text         dd 0
seg000:0000028C pLineNum_text       dd 0
seg000:00000290 nReloc_text         dw 0
seg000:00000292 nLineNum_text       dw 0
seg000:00000294 FLAG_TEXT           dd 60000020h        ; code | executable | readable
```

Вот мы и добрались до каталога сегментов! IMAGE HEADER секции ".text" выглядит вполне типично и никаких подозрений у нас не вызывает, но вот следующая за ним секция ".data" очень многое прояснеет...

```
seg000:00000298 a_data              db '.data',0,0,0
seg000:000002A0 vir_size_data       dd 3008h            ; размер секции .data в памяти
seg000:000002A4 vir_addr_data       dd 4B000h           ; адрес проекции на память
seg000:000002A8 szRawData_data      dd 14h              ; размер в файле
seg000:000002AC pRawData_data       dd 49E00h           ; смещение в файле
seg000:000002B0 pReloc_data         dd 0
seg000:000002B4 pLineNum_data       dd 0
seg000:000002B8 nReloc_data         dw 0
seg000:000002BA nLineNum_data       dw 0
seg000:000002BC FLAG_DATA           dd 0C0000040h       ; readable | writeable
```

"Ну и что здесь интересного?" — спросит иной читатель. А вот что — присмотритесь повнимательнее ***куда именно*** грузится содержимое данной секции. Если верить выделенной жирным шрифтом строке, — то по адресу IMAGE_BASE + 0x4B000. Ничего не напоминает? Во-первых, адрес 0x4B000 "волшебным" образом совпадает с адресом "мусорной" таблицы импорта (те, кто поимел сект с защитой этот адрес надолго запомнят, кстати, Харону не мешало бы его немножко замаскировать, чтобы он не так бросался в глаза). Во-вторых, изобразив процесс проецирования секций графически (см. рис. 0x005), мы с удивлением обнаружим, что секция .data расположена не следом за секцией .text (как это обычно и бывает), а находится ***внутри*** нее. Действительно, давайте подсчитаем: виртуальный адрес секции .text равен 0x1000, а ее размер — 0x49817, и последний байт секции приходится на адрес 0x59817, что превышает виртуальный адрес секции .data, равный 0x4B000.

Так вот оно что! Поскольку, секции отображаются на память в порядке их перечисления в каталоге (недокументированно, но факт!), то содержимое секции .data затирает область адресов 0x4B000 — 0x4E008! А что там у нас расположено?! ТАБЛИЦА ИМПОРТА!!! В дисковом файле по смещению 0x4B000 действительно расположен чи-

стейшей воды мусор (и это косвенно подтверждается тем, что изменения первых 0x14 байт работу программы не нарушают), а истинная таблица импорта расположена непосредственно в секции .data, которой соответствует смещение 0x49E00 дискового файла. Заглянем; что у нас там?!

```
seg000:00049E00 RealImportTable     dd offset IAT             ; OriginalFirstThunk
seg000:00049E04 TimeDateStamp1      dd 1
seg000:00049E08 ForwarderChain      dd 0FFFFFFFFh             ; no forward
seg000:00049E0C Name                dd offset aKernel32_dll   ; "kernel32.dll"
seg000:00049E10 FirstThunk          dd offset IAT
```

Вот, это действительно похожее на таблицу импорта со ссылкой на IAT. Кстати, не мешает посмотреть, что за функции импортирует IAT. Подгоняем курсор к "IAT" и, нажав, на <ENTER> смотрим:

```
seg000:0004B014 IAT                 dd 47440600h        ; DATA XREF: seg000:00049E00-o
seg000:0004B014                                         ; seg000:00049E10↑o
seg000:0004B018                     dd 50554F52h
seg000:0004B01C                     dd 69A8Bh
seg000:0004B020                     dd 0FF03FF11h
seg000:0004B024                     db    2             ;
seg000:0004B025                     db  4Ch             ; L
```

Мать родная! Ну почему ты не родишь меня обратно?! Опять вместо символических имен или на худой конец — ординалов, нам попадается этот проклятый мусор! Хотя, — подождите минуточку, — давайте попробуем определить что будет расположено по данному адресу после загрузки программы. Возвращаясь к описанию секции .data, мы обнаруживаем, что упустили один очень важный момент. Виртуальный размер секции .data (0x3008 байт) намного больше ее физического размера (0x14 байт) и потому, регион 0x4B014 — 49E008 будет заполнен нулями, а ведь "мусорная" IAT как раз и расположена по адресу 0x4B014! Следовательно, после загрузки ее содержимое окажется заполнено одними нулями, что соответствует пустой таблице импорта функций. Фу-х! Невероятно, но мы действительно в этом разобрались!!! Кстати, подобный прием и широко используется авторами упаковщиков исполняемых файлов.

```
seg000:000002C0 seg000:000002C0 b_rsrc  db '.rsrc',0,0,0
seg000:000002C8 vir_size_rsc            dd 27ACh          ; размер секции rsrc в памяти
seg000:000002CC vir_addr_rsc            dd 4F000h         ; адрес проекции на память
seg000:000002D0 szRawData_rsc           dd 27ACh          ; размер в файле
seg000:000002D4 pRawData_rsc            dd 4A000h         ; смещение секции в файле
seg000:000002D8 pReloc_rsc              dd 0
seg000:000002DC pLineMun_rsc            dd 0
seg000:000002E0 nReloc_rsc              dw 0
seg000:000002E2 nLineNum_rsc            dw 0
seg000:000002E4 FLAG_RSC                dd 50000040h      ; initalized data |
seg000:000002E4                                           ; shareable |readable
```

Аналогичным образом поступает и секция .rsrc, внедрясь в середину секции .text (но секцию .data она не перекрывает), причем, для ослепления некоторых дизассемблеров тут используется еще один хитрый примем: указанный "физический" размер секции .rsrc "вылетает" за пределы дискового файла. Системному загрузчику — хоть бы что, а вот некоторые исследовательские утилиты от этого и крышей поехать могут.

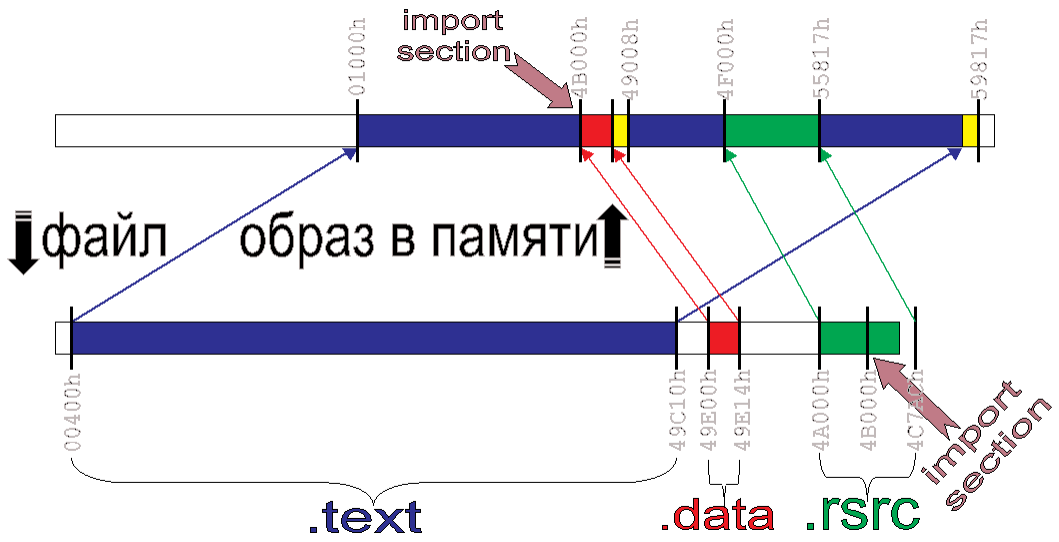


Рисунок 0x005. *Динамическое замещение таблицы импорта в процессе загрузки PE-файла*

Настало время проверить наши предположения на практике. Давайте загрузим эту извращенную программу отладчиком и посмотрим что содержится в памяти по адресу IMAGE_BASE + 0x4B000 = 0x44B000: мусор или нормальная таблица импорта? Отладчик soft-ice (как это и следовало ожидать) обламывается с отладкой этого извращенного файла, просто проскакивая точку входа, а вот WDB сполна оправдывая репутацию фирмы Microsoft (это не ирония!), пусть и не без ругательств, но все-таки загружает наш подопытный файл и послушно останавливается в точке входа.

```
Module Load: F:\IDAP\HARON\ulink.exe  (symbol loading deferred)
Thread Create:  Process=0, Thread=0
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (could not open symbol file)
Module Load: F:\IDAP\HARON\ulink.exe  (could not open symbol file)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (could not open symbol file)
Stopped at program entry point
```

Обратите внимание на выделенную жирным шрифтом строку. Отладчику показалось, что отлаживаемая программа импортирует некоторые функции... из самой себя! Но мы-то, излазившие защищенный файл вдоль и поперек, хорошо знаем, что за исключением kernel32.dll, никаких других экспортируемых и/или импортируемых библиотек здесь нет и такое поведение отладчика, судя по всему, объясняется все тем же самым "мусором". OK, переключаем свое внимание на окно с дампом памяти, заставляя ее отобразить содержимое таблицы импорта:

```
0x0023:0x0044B000 14 b0 04 00 01 00 00 00 ff ff ff ff f6 02 00 00 ................
0x0023:0x0044B010 14 b0 04 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0023:0x0044B020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Ура! Открываем на радостях пиво! Содержимое памяти доказательно подтверждает, что загрузка файла действительно происходит именно так, как мы и предполагали! Хорошо, но что же нам теперь делать? То бишь, найти-то причину помешательства дизассемблеров мы нашли, но вот как ее нейтрализовать? Ну, это не вопрос! Достаточно

лишь скопировать 0x14 байт памяти с адреса 0x49E00 по адресу 0x4B000 и скорректировать ссылку на IAT, направив ее на любое, заполненное нулями, место.

...HIEW теперь заглатывает защищенную программу и даже не пикает! А IDA... а IDA по прежнему отказываться обрабатывать этот файл и с завидным упорством слетает. В чем же причина? Вы, конечно, будете смеяться, но истинный виновник есть ни кто иной как Microsoft! Если бы не ее жутко прогрессивная платформа NET... А, впрочем, чего это я разворчался? Сами смотрите:

```
(•) Microsoft.Net assembly [pe.ldw]
( ) Portable executable for IBM PC (PE) [pe.ldw]
( ) MS-DOS executable (EXE) [dos.ldw]
( ) Binary file
```

Вот это да! С роду такого не было! Чтобы IDA да не правильно опознала формат файла!!! Перемещаем радио-кнопку на одну позицию вниз (ведь мы имеем дело отнюдь не с Microsoft Net assembly, а с PE!) и... IDA успешно открывает файл. Причем, с восстановлением таблицы импорта можно было и не возиться, — IDA просто ругнулась на мусор и все! Но кто ж знал?! Задним умом все мы крепки...

Короче, возвращаясь к нашим баранам (в данном случае — к терпеливо ожидающему нас отладчику) в точке входа, дизассемблерный текст выглядит так:

```
00446673 55              push    ebp
00446674 68AECF4200      push    42CFAEh
00446679 8BDC            mov     ebx,esp
0044667B 2403            and     al,3
0044667D 7203            jb      00446682
0044667F FE4302          inc     byte ptr [ebx+2]
00446682 D7              xlat    byte ptr [ebx]
00446683 27              daa
00446684 81042453970000  add     dword ptr [esp],9753h
0044668B 1AC9            sbb     cl,cl
0044668D 9F              lahf
0044668E FF33            push     dword ptr [ebx]
00446690 FC              cld
00446691 C3              ret
```

Не очень-то это похоже на осмысленный код программы! Может быть, это снова мусор? Маловероятно, — ведь отладчик использует штатный системный загрузчик PE-файлов и потому показывает образ файла таким, какой он в действительности есть, ну... если, конечно, защита тем или иным образом не противостоит отладке. Ладно, отставив разговорчики в строю, начинам трассировать код и... с первых же строк впадаем в некоторое замешательство. Защита опрашивает начальное значение регистра EAX, которое (если верить отладчику!) как будто бы равно нулю, но полной уверенности в этом у нас нет. Еще со времен старушки MS-DOS многие отладчики славились тем, что самостоятельно инициализировали регистры после загрузки, чем и выдавали себя (в частности, при нормальной загрузке файла регистр SI содержал в себе адрес первой исполняемой команды, а при загрузке под отладчиком Turbo Debugger и иже с ним, был равен нулю). Вообще-то, закладываться на "предопределенные" значения регистров — дурной тон. Никто не гарантирует, что в следующих версиях Windows что ни будь не изменится, и если такое вдруг произойдет, то защита откажет в работе, обломав не только хакеров, но и легальных пользователей. Впрочем, начальное значение регистра EAX (AX) по жизни равно нулю, и с некоторой натяжной за это можно зацепиться.

Далее защита непонятно зачем увеличивает старшее слово, только что закинутое в стек, на единицу и вызывает абсолютно бесполезные команды XLAT, DAA, ADD, SBB и... загружает регистр флагов в EAX. Уж не пытается ли она этим самым обнаружить флаг трассировки? Затем делает RETN для передачи управления по адресу: (0x42CFAE + 0x10000) + 0x9753 == 0x446701

```
.text:00446701          mov           edi, esi
.text:00446703          mov           esi, ebx
.text:00446705          sub           dword ptr      [esi], 1006Fh
.text:0044670B          lodsw
.text:0044670D          bswap         eax
.text:0044670F          inc           byte  ptr      [esi]
.text:00446711          lodsb
.text:00446712          mov           ah, al
.text:00446714          lodsb
.text:00446715                  bswap   eax
.text:00446717                  mov     ebp, eax
.text:00446719                  movzx   ecx, cl
.text:0044671C                  push    dword ptr [ebp+6Bh]
.text:0044671F                  lea     eax, [esi-8]
.text:00446722                  xchg    eax, fs:[ecx]
.text:00446725                  mov     edx, eax
.text:00446727                  inc     edx
.text:00446728                  jz      short loc_44672D
.text:0044672A                  mov     edx, [eax+4]
.text:0044672D
.text:0044672D loc_44672D:                              ; CODE XREF: .text:00446728j
.text:0044672D                  xchg    eax, [esp]
.text:00446730                  pushf
.text:00446731                  lea     ebx, [eax+21ADFh]
.text:00446737                  jnz     short loc_446745
.text:00446739                  lea     edi, [edi+0ACh]
.text:0044673F                  mov     dword_44CAF8, edi
.text:00446745
.text:00446745 loc_446745:                              ; CODE XREF: .text:00446737j
.text:00446745                  bts     dword ptr [esi-0Ch], 8
.text:0044674A                  jb      short loc_446753
.text:0044674C                  popf
.text:0044674D                  call    $+5
.text:00446752                  retf
```

...отладчик доходит лишь до RETF и после этого сразу же "дохнет". К тому же, остается совершенно непонятным, что же собственно делает этот запутанный и витиеватый код? При желании, конечно, с ним можно разобраться, но... нужно ли? Ведь отладить нашу подопытную мы все равного не сможем, во всяком случае в WDB.

Хорошо, зайдем с другого конца. Предположим, что программа работает с операционной системой не напрямую (через native API), а через подсистему win32 (win32 API). Тогда, установив точку останова на любую API-функцию, вызываемому программой, мы автоматически попадем в гущу "нормального" программного кода, уже распакованного (расшифрованного?) защитой. Весь вопрос в том: какие именно API-функции вызывает программа. Ну, пусть это будет GetVersion, с вызова которой начинается стартовый код практически любой программы. Запускаем soft-ice, нажимаем <Ctrl-D>, даем команду "bpx GetVersion", выходим из отладчика, вызываем unlink.exe и... ничего не происходит!

Отладчик не всплывает! Выходит, исследуемая нами программа не использует GetVersion! Что ж, удаляем предыдущую точку останова и пытаемся "забрейкать" CreateFileA (ну должен же линкер как-то открывать файлы!!!). Так, <Ctrl-D>, bpx CreateFileA<ENTER>, x<ENTER>... Ура! Это срабатывает! Отладчик перехватывает вызов защищенной программы и, после выхода из тела CreateFileA по команде P RET (в CreateFileA для нас действительно нет ничего интересного), мы оказывается в следующем коде:

```
001B:00416DEB  CALL    [USER32!CharToOemBuffA]
001B:00416DF1  PUSH    00000104
001B:00416DF6  LEA     EAX,[ESP+08]
001B:00416DFA  PUSH    EAX
001B:00416DFB  LEA     EDX,[ESP+0C]
001B:00416DFF  PUSH    EDX
001B:00416E00  CALL    [KERNEL32!GetShortPathNameA]
001B:00416E06  TEST    EAX,EAX
001B:00416E08  JZ      00416E2B
001B:00416E0A  LEA     EDX,[ESP+04]
001B:00416E0E  PUSH    00
001B:00416E10  PUSH    27
001B:00416E12  PUSH    03
001B:00416E14  PUSH    00
001B:00416E16  PUSH    01
001B:00416E18  PUSH    80000000
001B:00416E1D  PUSH    EDX
001B:00416E1E  CALL    [KERNEL32!CreateFileA]
001B:00416E24  MOV     EBX,EAX
001B:00416E26  CMP     EBX,-01
001B:00416E29  JNZ     00416E35
001B:00416E2B  CALL    [KERNEL32!GetLastError]
001B:00416E31  MOV     ESI,EAX
001B:00416E33  JMP     00416E5B
```

Обратите внимание: несмотря на отсутствие таблицы импорта, программа каким-то загадочным образом все-таки импортирует из kernell32.dll все, необходимые ей API-функции. Очень хорошо! Секс с native API и прочими извратами программистской хитрости отменяется! И мы остаемся в среде привычной нам подсистемы win32 API. Как именно осуществляется импорт — вот это уже другой вопрос! Кстати, давайте заглянем в одну такую функцию дизассемблером:

```
.text:00416E18            push 80000000h
.text:00416E1D            push edx
.text:00416E1E            call dword_44CC20 ; в отладчике это было KERNEL32!CreateFileA
.text:00416E24            mov  ebx, eax
.text:00416E26            cmp  ebx, 0FFFFFFFFh
.text:00416E29            jnz  short loc_416E35
...
.data:0044CC14 dword_44CC14   dd ?                    ; DATA XREF: sub_416DA0+AD?r
.data:0044CC14                                        ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18   dd ?                    ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C   dd ?                    ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20   dd ?                    ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                                        ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24   dd ?                    ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                                        ; sub_416F3C+128↑r
```

```
.data:0044CC28 dword_44CC28    dd ?                    ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                                         ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C    dd ?                    ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                                         ; sub_41AD20+12E↑r ...
.data:0044CC30 dword_44CC30    dd ?                    ; DATA XREF: .text:004014C4↑r
.data:0044CC34 dword_44CC34    dd ?                    ; DATA XREF: sub_419DD8+31↑r
.data:0044CC34                                         ; .text:0041A3E5↑r ...
.data:0044CC38 dword_44CC38    dd ?                    ; DATA XREF: sub_419DD8+1E↑r
.data:0044CC38                                         ; .text:0041A3A4↑r ...
```

Смотрите! В дисковом файле адресов импортируемых функций просто ***нет*** и таблица импорта судя по всему заполняется защитой динамически. А это значит, что в дизассемблере мы просто не сможем разобраться: какая именно функция в какой точке программы вызывается. Или... все-таки сможем?! Достаточно просто скинуть импорт работающей программы в дамп, а затем просто загрузить его в IDA! Затем, отталкиваясь от адресов экспорта, выданных "dumpbin /EXPORTS kernel32.dll", мы без труда приведем таблицу импорта в нормальный вид. Итак, прокручивая экран дизассемблера вверх, находим где у этой таблицы расположено ее начало или нечто на него похожее (если мы ошибемся — ничего странного не произойдет, просто часть функций останется нераспознанными и когда мы с ними столкнемся лицом к лицу, эту операцию придется повторять вновь). Вот, кажется, мы нашли, что искали, смотрите:

```
.data:0044CC09                                         ; sub_43E6D4+22A↑r ...
.data:0044CC0A                 db ?                    ; unexplored
.data:0044CC0B                 db ?                    ; unexplored
.data:0044CC0C                 db ?                    ; unexplored
.data:0044CC0D                 db ?                    ; unexplored
.data:0044CC0E                 db ?                    ; unexplored
.data:0044CC0F                 db ?                    ; unexplored
.data:0044CC10                 db ?                    ; unexplored
.data:0044CC11                 db ?                    ; unexplored
.data:0044CC12                 db ?                    ; unexplored
.data:0044CC13                 db ?                    ; unexplored
.data:0044CC14 dword_44CC14    dd ?                    ; DATA XREF: sub_416DA0+AD↑r
.data:0044CC14                                         ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18    dd ?                    ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C    dd ?                    ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20    dd ?                    ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                                         ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24    dd ?                    ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                                         ; sub_416F3C+128↑r
.data:0044CC28 dword_44CC28    dd ?                    ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                                         ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C    dd ?                    ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                                         ; sub_41AD20+12E↑r ...
```

Условимся считать адрес 0044CC14h ***началом***. Используя точку останова на CreateFileA, вновь вламываемся в программу и, отключив окно "data" командой wd, скидываем таблицу импорта в хистори: "d 44CC14". Выходим из Айса, запускаем NuMega Symbol Loader и записываем историю команд в файл winice.log (или любой другой по вашему вкусу). И как со всем этим нам теперь работать? Рассмотрим это на примере функции "call dword_44CC78". Прежде всего мы должны выяснить, какое значение находится в загруженной программе по адресу: 0x44CC87. Открываем winice.log по <F3> и смотрим:

```
0010:0044CC78 77E8668C  77E8F51E  77E93992  77E8DBF8      .f.w...w.9.w...w
0010:0044CC88 77E93F05  77E85493  77E87BE4  77E87D16      .?.w.T.w.{.w.}.w
0010:0044CC98 77E8C0A6  77E8AF8E  77E8878A  77E8BDE8      ...w...w...w...w
0010:0044CCA8 77E94911  77E9499C  77E9138C  77E8D019      .I.w.I.w...w...w
```

Теперь, обратившись к таблице экспорта kernel32.dll, определяем: а) базовый адрес ее загрузки (в данном случае: 0x77E80000); б) имя функции, сумма RVA и IMAGE BASE которой совпадает со значением 0x77E8668C. Вычитаем из 0x77E8668C базовый адрес загрузки — 0x77E80000 и получаем: 0x668C. Ищем строку 0x668C простым контекстным поиском и...

```
302   12D 0000668C GetLastError
```

...это оказывается ни кто иной, как GetLastError, что и требовалось доказать. Конечно, восстанавливать весь импорт вручную — крайне скучно и утомительно. Но кто нам сказал, что мы должны это делать именно вручную?! Ведь дизассемблер IDA поддерживает скрипты, что позволяет автоматизировать всю рутинную работу (подробнее о языке скрпитов можно прочитать в книге "Образ мышления — дизассемблер IDA" от Криса Касперски, то есть, собственно, меня).

OK, еще один барьер успешно взять. Воодушевленные успехом и доверху наполненные выпитым во время хака пивом, мы продолжаем! В плане возвращения к нашим баранам, сосредоточим свои усилия на загрузчике таблице импорта, расположенном по всей видимости где-то недалеко от точки входа. Несмотря на то, что soft-ice, по-прежнему, упорно проскакивает Entry Point, обламываясь с загрузкой защищенного файла (впрочем, другие версии soft-ice с этим справляются на ура), мы можем легко обхитрить защиту просто воткнув в точку входа бряк поинт. Поскольку, бряк поиск должен устанавливаться во вполне определенном контексте, используем уже известную нам нычку с CreateFileA. Итак, "bpx CreateFileA", <Ctrl-D>, запускаем unlink и, когда soft-ice "всплывает" даем: "bpx 0x446673" (адрес точки входа), выходим из soft-ice и... запускаем ulink вновь. Отладчик тут же всплывает:

```
001B:00446673  55              PUSH    EBP
001B:00446674  68AECF4200      PUSH    0042CFAE
001B:00446679  8BDC            MOV     EBX,ESP
001B:0044667B  2403            AND     AL,03
001B:0044667D  7203            JB      00446682
001B:0044667F  FE4302          INC     BYTE PTR [EBX+02]
001B:00446682  D7              XLAT
001B:00446683  27              DAA
```

Знакомые места! Трассируем код до тех пор пока на не встретится подозрительный RETF (от RET FAR — далекий возврат), передающий управление по следующему адресу:

```
001B:77F9FB90  8B1C24          MOV     EBX,[ESP]
001B:77F9FB93  51              PUSH    ECX
001B:77F9FB94  53              PUSH    EBX
001B:77F9FB95  E886B3FEFF      CALL    77F8AF20
001B:77F9FB9A  0AC0            OR      AL,AL
001B:77F9FB9C  740C            JZ      77F9FBAA
001B:77F9FB9E  5B              POP     EBX
001B:77F9FB9F  59              POP     ECX
```

Судя по адресу, этот код принадлежит непосредственно самой операционной системе (а точнее — NTDLL.DLL) и представляет собой функцию **KiUserExceptionDispatcher**. Но что это за функция? Ее описание отсутствует в SDK, но поиск по MSDN

обнаруживает пару статей Мета Питтрека, посвященных механизмам функционирования SEH и функции KiUserExceptionDispatcher в частности.

Структурные исключения! Ну конечно же! Какая защита обходится без них! Ладно, разберемся, ворчим мы себе под нос, продолжая трассировку защиты дальше. Увы! В той же точке, где WDB терял над программой контроль, soft-ice просто слетает. Ах, вот значит как!!! Ну, защита, держись!!!

(продолжение следует)

# Примеры реальных взломов (фрагмент книги "Техника и философия хакерских атак 2.000")


## Крис Касперски
## kk@sendmail.ru


В настоящей книге все атаки рассматривались исключительно на примерах, специально написанных для демонстрации того или иного алгоритма программ ‘CrackMe’. При этом многие из них были слишком искусственными и далекими от реальных защитных механизмов. Это было удобно для изложения материала, но не отражало реальных существующих защит.

Поэтому, я решил включить в приложения некоторые примеры реальных взломов. Все эти программы широко распространены и отражают средний уровень защиты условно-бесплатных программ. Заметим, что он существенно ниже чем многие из предложенных в книге реализаций.

Напоминаю, что взлом в той или иной мере конфликтует с российским и международным законодательством. Поэтому, необходимо помнить, что взлом не освобождает от регистрации и может быть использован только в образовательных целях, но никак не для получения какой-либо выгоды. В этом случае конфликтов с законодательсвом не возникнет.

# Компилятор Intel C++ 5.0.1

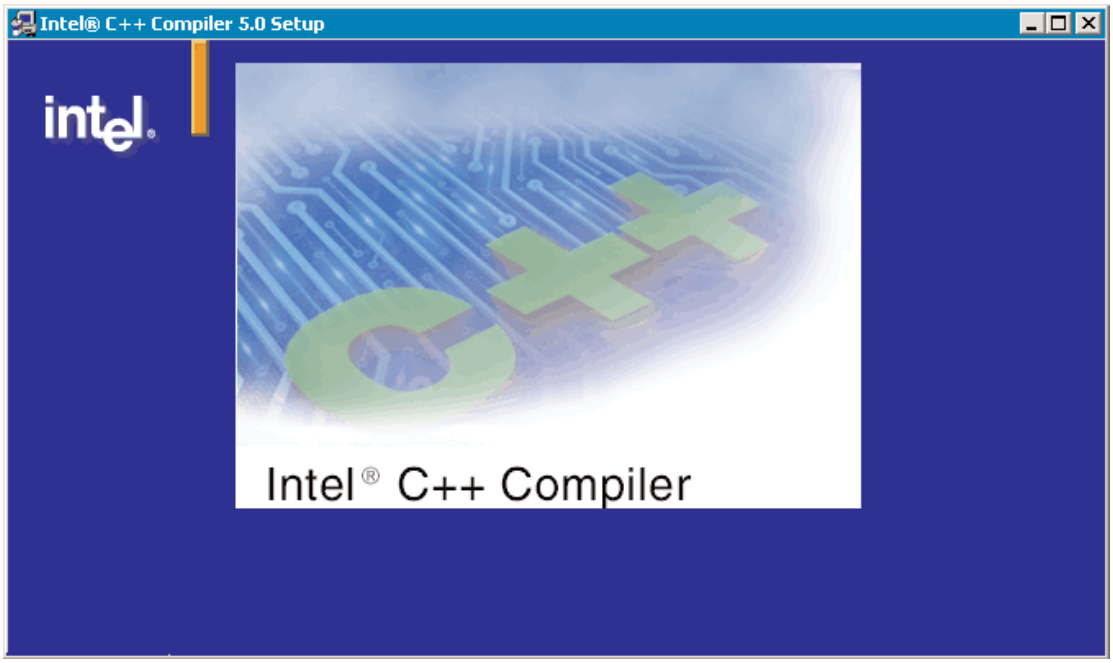


Рисунок 0x001. *Логотип компилятора Intel C++*

Прежде, чем приступать к обсуждению аспектов стойкости защиты компилятора **Intel C++ 5.0.1**, считаю своим долгом заявить, что я глубоко восхищен этим великолепным программным продуктом и ломать его, на мой взгляд, по меньшей мере кощунственно. Впрочем, сегодня только ленивый не найдет в Сети кряк (один только Google по запросу "Intel C++ crack" выдает свыше 12 тысячи ссылок!), так что никакого вреда от данной публикации не будет.

Немного грустных новостей для начала. Приобрести легальную версию данного компилятора для жителей России оказывается чрезвычайно затруднительно. И вопрос упирается даже не в то "сколько он стоит" (а стоит он, если мне не изменяет память что-то в районе тысячи долларов), — компания Intel просто игнорирует данный сегмент рынка. Обращения в российское представительство компании с просьбой предоставить (за деньги!) данный компилятор для его же описания (читай — рекламы и продвижения) в книге "Техника оптимизации программ", положительных результатов не дали. Даже после того, как к этому вопросу подключились прямо-таки скажем не мелкие отечественные издательства BHV и Солон — Р. Ладно, не хотят продавать — ну и не надо! Благо хоть с сервера компании можно свободно утянуть 30-дневный триал. Негусто, конечно, но для сравнительного тестирования — вполне достаточно (а для других целей мне этот компилятор и не нужен!).

Впрочем, все оказалось не так просто! С web-сервера компилятор за просто так не отдался, после заполнения регистрационной формы меня вежливо поблагодарили и сообщили, что сейчас ко мне на "мыло" упадет письмо с триальной лицензией и инструк-

цией по ее установке. Это "сейчас" заняло у севера аж несколько дней (такое впечатление, что анкеты просматриваются вручную). OK! Лицензия получена! Начинаем скачивать файл.... Как это так докачка не поддерживается?! А вот не поддерживается и все! Учитывая, что у меня лишь хлипкий Dial-Up по каналу в 19.200 (да и тот по межгороду), скачать полста мегабайт без единого разрыва просто нереально. К тому же, работа над книгой уже близиться к завершению и вносить в нее еще один компилятор (а значит, переписывать кучу текста заново) мне становится просто в лом. Да и Intel C++ это далеко не самый популярный в кругах российских программистов компилятор и книга без него как ни будь уж переживет (хотя, посмотреть как Intel оптимизирует код под свои процессоры очень хотелось, да и документация по компилятору вдохновляла)[1].

Разозлившись на весь свет (и на парней из Intel в частности), я отправился на ftp-сервер компании, откуда наскоро, всего за каких-то три дня, слил полнофункциональную (хотя и шибко несвежую) версию компилятора, находящуюся по следующему адресу: **ftp://download.intel.com/software/products/downloads/C5.0.1-15.exe.** (приятно, что ftp докачку исправно поддерживал и многократные разрывы никаких проблем не вызывали). Польстившись на размер, я скачал именно пятую версию компилятора, которая была в полтора раза легче шестой (под которую у меня имелась неиспользованная триальная лицензия) и аж в два раза компактнее седьмой — новейшей на момент написания этих строк — версии, ломать которую, из "политических" соображений, я все равно бы не рискнул, так зачем же ее зря качать?

Теперь, собственно, мы и подходим к известному философскому вопросу: этично ли ломать программный продукт уважаемой тобой компании или без этого можно обойтись? Да если бы без этого было возможно обойтись, я бы — честное слово — без тени сожаления выложил за этот замечательный продукт пачку вечнозеленых, но, увы... компания не проявляет ко мне как покупателю никакого интереса и, кроме как ломать, ничего другого просто не остается!

Итак, инсталлируем Intel C++ и, предварительно скопировав в просроченную лицензию от шестой версии в папку \Intel\Licenses, запускаем головной файл программы:

```
...\Program Files\Intel\C501\Compiler50\ia32\bin>icl.exe
Intel(R) C++ Compiler for 32-bit applications, Version 5.0.1   Build 010525Z
Copyright (C) 1985-2001 Intel Corporation.  All rights reserved.

icl: error: could not checkout FLEXlm license
checkout failed: No such feature exists (-5,357)
```

Как и следовало ожидать: "could not checkout **FLEX lm** license" (*"не могу проверить FLEX lm лицензию"*) — компилятор ругается и прекращает свою работу. Ага, стало быть, программа защищена FLEX'ом — достаточно известным в хакерских кругах менеджером лицензий от компании **Globetrotter Inc**, представляющим собой достаточно продвинутую защиту интегрированного типа. Разработчик защищаемого приложения получает в свое распоряжение SDK, содержащее как тривиальные функции проверки валидности ключевого файла (лицензии), так и развитые средства динамической шифровки файла. При грамотном подходе к защите запустить защищенную программу без наличия соответствующей ей лицензии доподлинно невозможно. Если часть про-

[1] Самое смешное, что когда я все-таки скачал компилятор через своих московских знакомых (ну, для Москвы 45 мегабайт это вообще ни что) он наотрез отказался работать, мотивируя свое поведение тем, что срок демонстрационной лицензии уже истек...

граммы зашифрована, то пытаться расшифровать ее без ключа — дохлое дело. Правда не факт, что парни из Intel действительно использовали шифрование, к тому же, зашифрованные фрагменты иногда удается восстановить по косвенным данным. Это смотря, что еще зашифровано!

Разумеется, при наличии триальной лицензии шифровка снимается без труда, но в том-то все и дело, что триальной лицензии у меня не было! Тем не менее надежда меня не покидала и, перекусив для смелости батоном докторской колбасы, сдобренной значительным количеством кетчупа, я запустил свой любимый дизассемблер IDA, и... не знаю у кого как, а у меня вид консольной IDA, распахнутой на весь экран, всегда вызывает чувство благоговения. OK, ну-ка посмотрим, где скрываются те текстовые строки, которые выводятся при отсутствии лицензии на экран. Результат: ни "No such feature exists", ни "could not checkout" в ASCII-строках (т. е. тех строках, что сумел распознать автоматический анализатор IDA) **не найдено**. Хорошо, зайдем с другого конца. Нажимаем <F4> для переключения в hex-режим и давим <ALT-T> для поиска текстовых строк в "сыром" виде. Что ж, на этот раз поиск "could not checkout" увенчался успехом!

```
.data1:0042D9C0  63 6F 75 6C 64 20 6E 6F-74 20 63 68 65 63 6B 6F "could not checko"
.data1:0042D9D0  75 74 20 46 4C 45 58 6C-6D 20 6C 69 63 65 6E 73 "ut FLEXlm licens"
.data1:0042D9E0  65 00 00 00 63 6F 75 6C-64 20 6E 6F 74 20 6C 6F "e...could not lo"
.data1:0042D9F0  63 61 74 65 20 46 4C 45-58 6C 6D 20 72 65 67 69 "cate FLEXlm regi"
.data1:0042DA00  73 74 72 79 20 6B 65 79-00 00 00 00 63 6F 75 6C "stry key....coul"
```

Нажимаем <F4> еще один раз для возврата в режим дизассемблера, подводим курсор к адресу 42D9C0h и нажимаем <A> для преобразования цепочки байт в ASCII-строку. В результате мы получаем:

```
.data1:0042D9C0 aCouldNotChecko db 'could not checkout FLEXlm license',0
```

А как узнать, кто же выводит строку-ругательство на экран? Нет ничего проще! Вновь переключившись в режим дизассемблера, по <F4>, давим <ALT-T> для поиска последовательности "C0 D9 40 00" — адрес строки, представленный в обратном (с учетом порядка следования старших байтов) виде. О-па! Мы видим код, наподобие следующего:

```
.data:00420CE8                  db 50h
.data:00420CE9                  db 0DEh ; ▌
.data:00420CEA                  db  42h ; B
.data:00420CEB                  db    0 ;
.data:00420CEC                  db    1 ;
.data:00420CED                  db    0 ;
.data:00420CEE                  db    0 ;
.data:00420CEF                  db    0 ;
.data:00420CF0                  db 2Ch
.data:00420CF1                  db 0DEh ; ▌
.data:00420CF2                  db  42h ; B
.data:00420CF3                  db    0 ;
.data:00420CF4                  db    2 ;
.data:00420CF5                  db    0 ;
.data:00420CF6                  db    0 ;
.data:00420CF7                  db    0 ;
```

Косвенный вызов строки! Ну, собственного, этого и следовало ожидать (иначе с чего бы это, автоматический анализатор IDA их не распознал?). Хорошо, преобразуем

двойные слова в смещения, руководствуясь при этом тем, что число "42h" должно выпадать на младший байт старшего слова (иначе адрес ссылки уйдет за диапазон предельно допустимых значений) и получаем:

```
.data:00420DE8 dd offset aCouldNotLoca_0 ; "could not locate FLEXlm registry direct"
.data:00420DEC dd 21h
.data:00420DF0 dd offset aCouldNotLocate ; "could not locate FLEXlm registry key"
.data:00420DF4 dd 22h
.data:00420DF8 dd offset aCouldNotChecko ; "could not checkout FLEXlm license"
.data:00420DFC dd 23h
```

Попробуем теперь найти ту су..., в общем тот код, что обращается к указателю (на ругательную строку) расположенному по адресу 420CE8h? Не надо спешить! По виду полученной таблицы смещений можно с уверенностью заключить, что прямого обращения к ее элементам не будет. Можно предположить, что числа, стоящие возле ссылок на строки — коды ошибок, а сами строки — соответствующие тексты сообщений. Если так, то с вероятностью близкой к единице разработчиками программы использовалась относительная адресация, т. е. для вычисления эффективного адреса элемента, ее смещение в таблицы суммируются с базовым адресом таблицы, — единственным адресом, который загружается явно.

Прокручивая экран дизассемблера вверх, мы внезапно натыкаемся на длинную последовательность нулей, интерпретируемую нами как начало таблицы:

```
.data:00420CDE                db    0 ;
.data:00420CDF                db    0 ;
.data:00420CE0 off_420CE0     dd offset unk_42DE80      ; DATA XREF: sub_403370+5E↑r
.data:00420CE4 dword_420CE4   dd 0                      ; DATA XREF: sub_403370+19↑r
.data:00420CE4                                          ; sub_403370+39↑r
.data:00420CE8                dd offset aCouldNotFindDi ; "could not find directory"
.data:00420CEC                dd 1
```

Ага! Есть две перекрестных ссылки! Это хорошо! Теперь поднимемся по ним вверх, прямиком к вызывающему их коду? Можно, конечно, поступить и так, но есть и более универсальное решение: запустив Soft-Ice, мы устанавливаем точку останова на чтение ячейки 420DE8h (если вы еще не забыли, это адрес элемента таблицы, ссылающийся на искомую ругательную строку). Теперь, кто бы к ней не обращался, Soft-Ice обязательно всплывет, и ведь действительно он всплывает! Пару раз отдав команду "P RET", поднимающую нас из дебрей глубоко вложенных процедур поближе к свету. Наконец, мы взбирается на вершину стека, и очередной "P RET" приводит к завершению программы. OK, повторяем все заново, делая на этот раз на один "P RET" меньше. Записываем любой из близлежащих адресов (пусть это будет для определенности адрес 4031C4h) и натравливаем на него IDA.

```
.text:004031C4       call    lc_checkout
.text:004031C9       test    eax, eax
.text:004031CB       jz      short loc_403215
.text:004031CD       cmp     eax, 0FFFFFFF6h
.text:004031D0       jz      loc_41B000
.text:004031D6       cmp     eax, 0FFFFFFB7h
.text:004031D9       jz      loc_41B01A
.text:004031DF
.text:004031DF loc_4031DF:                           ; CODE XREF: .text1:0041B015↓j
.text:004031DF                                       ; .text1:0041B026↓j
```

```
.text:004031DF       mov   [esp+240h+var_240], 23h
.text:004031E6       call  sub_405B00
.text:004031EB       mov   eax, dword_424C9C
.text:004031F0       mov   [esp+240h+var_240], eax
.text:004031F3       mov   [esp+240h+var_23C], offset aCheckoutFailed ; "checkout failed"
.text:004031FB       call  lc_perror
.text:00403200       mov   eax, dword_424C9C
.text:00403205       mov   [esp+240h+var_240], eax
.text:00403208       call  lc_get_errno
.text:0040320D       mov   [esp+240h+var_240], eax
.text:00403210       call  sub_405BA0
.text:00403215
.text:00403215 loc_403215:                                ; CODE XREF: sub_403000+1CB↑j
.text:00403215       mov   eax, dword_424C9C
.text:0040321A       mov   edx, dword_421E3C
.text:00403220       mov   [esp+240h+var_240], eax
.text:00403223       mov   [esp+240h+var_23C], edx
.text:00403227       call  lc_auth_data
.text:0040322C       mov   edx, eax
.text:0040322E       mov   eax, dword_424C9C
.text:00403233       call  sub_40A6F8
.text:00403238
```

Вот это да! — восклицаем мы, пришибленно уставившись на экран. Многое мы ожидали от IDA, но вот чтобы она, так запросто, представила символьные имена защитных функций, все говорящие за себя: lc_chekout, lc_perror, lc_auth_data... Черт, возьми, как? Вдохновленные смутной надеждой, мы неуверенно подгоняем курсор к lc_chekout и нажимаем на <ENTER>.

```
.idata:0041D12C ; Imports from LMGR327A.dll
.idata:0041D12C ;
.idata:0041D12C extrn __imp_lc_init:dword          ; DATA XREF: lc_init↑r
.idata:0041D130 extrn __imp_lc_expire_days:dword
.idata:0041D130                                    ; DATA XREF: lc_expire_days↑r
.idata:0041D134 extrn __imp_lc_free_job:dword      ; DATA XREF: lc_free_job↑r
.idata:0041D138 extrn __imp_lc_checkin:dword       ; DATA XREF: lc_checkin↑r
.idata:0041D13C extrn __imp_lc_auth_data:dword     ; DATA XREF: lc_auth_data↑r
.idata:0041D140 extrn __imp_lc_get_errno:dword     ; DATA XREF: lc_get_errno↑r
.idata:0041D144 extrn __imp_lc_perror:dword        ; DATA XREF: lc_perror↑r
.idata:0041D148 extrn __imp_lc_checkout:dword      ; DATA XREF: lc_checkout↑r
.idata:0041D14C extrn __imp_lc_set_attr:dword      ; DATA XREF: lc_set_attr↑r
.idata:0041D150
```

Святой Кондратий! И ***это*** они еще называют защитой?! Все защитные функции вынесены в отдельную динамическую библиотеку (наверное, чтобы взломщику разбираться было легче?) — LMGR327A.DLL, в названии которой угадывается "Library MaNaGeR", причем, это штатные функции FLEX lm, описание которых можно найти в его же SDK (хоть SDK на FLEX ln с компилятором и не поставляется, найти его в Сети — плевое дело).

Отыскав в текущем каталоге этот самый LMGR327A.DLL, мы открываем его HIEW'ов на предмет полного переписывания функции lc_checkout. Ну, насчет "переписывания" автор, ясное дело, загнул. Всего-то и требуется, — заставить lc_checkout всегда возвращать нуль, для чего первые две команды ее тела должны выглядеть приблизительно так: "XOR EAX, EAX / RETN". Записываемся, и с дрожью в сердце, запускам

icl.exe на выполнение. Критическая ошибка приложения? А чего мы хотели?! Ведь теперь функция lc_auth_data получает неверные данные и гробит все к черту. Впрочем, не будем спешить. Беглое исследование процедуры sub_40A6F8, как будто, не выявляет никаких следов шифрования и поэтому ее можно смело удалить, не забыв тоже самое, "на всякий пожарный" случай, проделать и с lc_auth_data (самое простое — впихнуть в ее начало RETN). Сохраняемся, запускам icl.exe и... компилятор работает! Все! Больше тут нечего ломать!

Самое забавное, что размер защитного механизма (413 Кб) в **два с половиной** раза превышает размер защищенной с его помощью программы (176 Кб)! Как говорится — no comment.


Рисунок 0x002. *Логотип Intel Fortran Compiler*

Ситуация с этим компилятором, в кратце, такова. В процессе работы над третьим томом "Образа мышления IDA" я исследовал большое количество компиляторов на предмет особенностей их кодогенерации и вытекающих отсюда трудностей восстановления исходного кода. Не избежал этой участи и "Intel Fortran Compiler", обнаруженный на диске "Научись сам программировать на FORTRAN". Краткая аннотация на буклете гласила "*Intel FORTRAN Compiler 4.5 — новейшая версия знаменитого компилятора. Для регистрации программы смотрите поддиректорию CRACK*". Ну, на счет "новейшего" составители диска явно приврали, т .к. на тот момент уже вышла седьмая версия, да и CRACK оказался некорректным. Вместо того, чтобы ломать защиту, он ломал сам компилятор, необратимо его гробя. К счастью, оригинальный ifl.exe на диске все-таки имелся и это давало возможность заставить работать компилятор мне самому. В конце концов, использовать в коммерческих целях этот, бесспорно, замечательный программный продукт я все равно не собирался, а для серии тестовых прогонов не то, что месяца (положенного мне по праву) даже нескольких дней было вполне предостаточно, поэтому, с этической точки зрения, ничего кощунственного я не совершал (просто мне очень уж не хотелось тянуть ~160 метров из Интернета, с моим междугородним Интернетом это действительно проблематично).

Итак, запускаем оригинальный файл компилятора на выполнение и лицезрим, как он спускает на нас Полкана (ругается в смысле):

```
KPNC$C:\Program Files\Intel\compiler45\bin>ifl1.exe >1
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error: The evaluation period has expired.

    The evaluation period for this trial version of the
    Intel(R) Fortran Compiler has expired.  For product ordering
    information, please refer to the product release notes or visit the
    Intel Developer web site at the following URL:

                http://developer.intel.com/vtune
```

Ни слова о FLEX lm! (см. "Компилятор Intel C++ 5.0.1") и файл LMGxxx.DLL отсутствует. Странно! Похоже, что Fortran Compiler защищен иначе, что, собственно, и не удивительно, поскольку их делали разные группы.

Что ж, запускаем IDA и натравливаем на нее исполняемый файл, который, кстати, занимает всего 176,128 Кб, что с точностью до байта соответствует размеру Intel C++ 5.1 Compiler. Странно! Но, как бы там ни было, ASCII-строки "The evaluation period has expired" автоматический анализатор IDA в тексте дизассемблируемого файла так и не нашел. Что ж, тогда мы сделаем это сами. <F4>, <ALT-T>, "The evaluation period" и...

```
.data1:0042A220  54 68 65 20 65 76 61 6C-75 61 74 69 6F 6E 20 70 "The evaluation p"
.data1:0042A230  65 72 69 6F 64 20 68 61-73 20 65 78 70 69 72 65 "eriod has expire"
.data1:0042A240  64 2E 0A 0A 20 20 20 20-54 68 65 20 65 76 61 6C "d.◙◙    The eval"
.data1:0042A250  75 61 74 69 6F 6E 20 70-65 72 69 6F 64 20 66 6F "uation period fo"
.data1:0042A260  72 20 74 68 69 73 20 74-72 69 61 6C 20 76 65 72 "r this trial ver"
.data1:0042A270  73 69 6F 6E 20 6F 66 20-74 68 65 0A 20 20 20 20 "sion of the◙    "
```

Теперь, вновь нажимаем <ALT-T> для поиска последовательности "20 A2 42 00" — адрес начала строки, заданный в обратном виде. Результат не заставляет себя долго ждать:

```
.data:00419390  60 A3 42 00 4F 00 00 00-20 A2 42 00 50 00 00 00 "`гB.O... вB.P..."
.data:004193A0  00 A2 42 00 51 00 00 00-E0 A1 42 00 52 00 00 00 ".вB.Q...рбB.R..."
.data:004193B0  C0 A1 42 00 53 00 00 00-A0 A1 42 00 54 00 00 00 "LбB.S...абB.T..."
.data:004193C0  60 A1 42 00 55 00 00 00-40 A1 42 00 56 00 00 00 "`бB.U...@бB.V..."
.data:004193D0  20 A1 42 00 57 00 00 00-00 A1 42 00 58 00 00 00 " бB.W....бB.X..."
```

Переключаемся обратно в дизассемблер, трижды жмем <D> для преобразования цепочки байт в двойное слово, затем <O> для перевода его в смещение и... в результате таких манипуляций получаем приблизительно такую же таблицу, как и в нашем предыдущем случае с Intel C++

```
.data:00419390 dd offset aSNoteTheEvalua ; "%s: NOTE: The evaluation period for thi"
.data:00419394 dd 4Fh
.data:00419398 dd offset aTheEvaluationP ; "The evaluation period has expired.\n\n    "
.data:0041939C dd 50h
.data:004193A0 dd offset aCommandLineErr ; "Command line error"
.data:004193A4 dd 51h
.data:004193A8 dd offset aCommandLineWar ; "Command line warning"
.data:004193AC dd 52h
```

А посему и действовать мы будем точно так же: поставим бряк на адрес 0419390h и дождемся пока отладчик не получит управления. Кстати, на счет отладчика. В момент написания этих строк у автора как раз закачивалась седьмая версия компилятора Intel C++ и от использования soft-ice пришлось воздержаться (в момент своей активации soft-ice полностью "замораживает" операционную систему, что пагубно влияет на Интернет, а точнее на установленные TCP/IP соединения). И вместо soft-ice автор решил для разнообразия использовать **Microsoft WDB**, который кстати, справился со своей задачей ничуть не хуже.

Запускам WDB на выполнение, нажимаем <Ctrl-E>, указываем имя загружаемого файла, переходим в окно команд ("Command Window") и устанавливаем точку остановка на адрес 0419398h для чего отдаем команду "BA r4 0x0419398" (что расшифровывается как: "Break on Access of Read 4 bytes long"). Затем для продолжения выполнения программы пишем "G" и с полсекунды ждем...

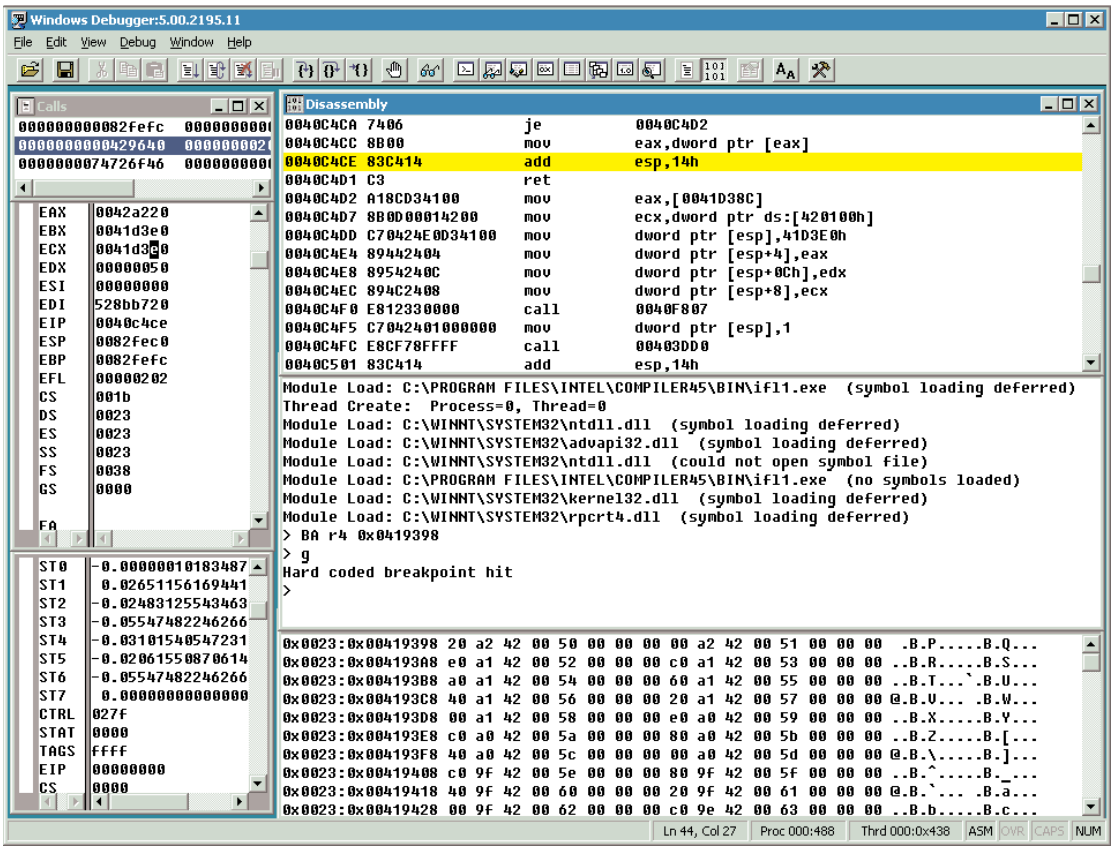


Рисунок 0x003. *Внешний вид отладчика MS WBD в процессе ломания программы*

Ага, отладчик говорит "Hard coded breakpoint hit" ("*сработала аппаратная точка останова*") и приостанавливает выполнение отлаживаемой программы. Сама же отлаживаемая программа к этому моменту уже успела вывести на экран:

```
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error:
```

Обратите внимание на строки, выделенные жирным шрифтом! Очевидно, они свидетельствуют о том, что мы попали не в самое начало защитной процедуры, а где-то в ее середину. Кстати, а что у нас там лежит на стеке? Смотрим (~View Stack, см. рис. 0x003). Всего три адреса, — довольно неглубокий уровень вложения, не так ли? Причем (обратив свой взор к окну дизассемблера), сейчас уровень вложения еще понизится, т. к. следующей командой мы выходим из этой процедуры:

```
0040C4CC 8B00            mov         eax,dword ptr [eax]
0040C4CE 83C414          add         esp,14
0040C4D1 C3              ret
```

Теперь неспешно трассируем код, попеременно поглядывая то на дизассемблированный листинг, то на консоль отлаживаемой программы. Следующая трассируемая функция (внутрь которой мы не заходим, а "заглатываем" ее одним нажатием <F10>) выводит на экран "The evolution period has expired", но не завершает программу, а продолжает ее выполнение. Что ж! Тогда и мы продолжим (трассировку)! Вызов функции

040F5FEh проходит без каких либо внешних проявлений. Так и не поняв, зачем она собственно нужна, мы поднимается на еще один уровень вверх, куда нас забрасывает завершающий функцию RET.

```
00403C7C E833880000       call        0040C4B4
; отсюда ^^^^^^^^^^^^^^^^^^^^ мы только что вышли

00403C81 89442404         mov         dword ptr [esp+4],eax
00403C85 891C24           mov         dword ptr [esp],ebx
00403C88 896C2408         mov         dword ptr [esp+8],ebp
00403C8C E8A8BB0000       call        0040F839
; эта процедура выводит "The evaluation period has expired."

00403C91 C70424C04C4200   mov         dword ptr [esp],424CC0h
00403C98 895C2404         mov         dword ptr [esp+4],ebx
00403C9C E85DB90000       call        0040F5FE
; эта процедура ничего не делает

00403CA1 83C414           add         esp,14h
00403CA4 5B               pop         ebx
00403CA5 5D               pop         ebp
00403CA6 C3               ret
```

...и таким Макаром мы трассируем код до тех пор, пока не наткнемся на следующую конструкцию:

```
0040105A E8E1800000       call        00409140 ; отсюда мы только что вышли по RETN
0040105F 0FB6C0           movzx       eax,al
00401062 85C0             test        eax,eax
00401064 0F84C4000000     je          0040112E
```

Что в ней необычного? А то, что это первая, встретившаяся нам материнская процедура, которая анализирует код возврата дочерней функции. В нашем случае регистр EAX содержит значение "ноль" и, стало быть, следующий условный переход выполняется. Но не тот ли это переход который нам нужен? Что ж, сейчас мы это узнаем — нажимаем клавишу <F10> еще несколько раз... Оп-ля! Наш условный переход перебрасывает нас на ту ветку программы, которая спустя несколько команд скоропостижно сдыхает, захлопывая окно программы. А что произойдет, если команду "JE" в строке 401064h заменить на противоположную (или, как вариант, просто удалить этот условный переход)? Пробуем...

Компилятор по прежнему смачно ругается на "evaluation expired", но... он работает! Работает!! Работает!!! По соображениям экономии экранного места (в самом деле, ругательство занимает чуть ли не половину экрана и смотрится крайне некрасиво) мы забиваем вызов процедуры 0409140h командами NOP. Проверяем — сработало ли? Ну... это смотря как посмотреть. Трехэтажный мат действительно исчез, но вот лаконичная строка "Evaluation Copy" так и осталась. Найдем что за код ее выводит? Зачем? — лучше найти саму эту строку и тем же HIEW'ом ее переписать во что ни будь более привычное, например: "hacked by mother-fucker guy". Переписываем, и... пользуемся компилятором в свое удовольствие, не забывая, однако о том, что по истечении 30-дневного срока вы будете должны его стереть, в противном случае вы поступите очень и очень нехорошо, да и незаконно.

# Intel C++ 7.0 Compiler

...компилятор Intel C++ 7.0 докачался глубокой ночью, часу где-то в пятом утра. Спать хотелось неимоверно, но и любопытство: была ли усилена защита или нет, тоже раздирало. Решив, что до тех пор пока не разберусь с защитой, я все равно не усну, я, открыв новую консоль, и переустановив системные переменные TEMP и TMP на каталог C:\TEMP, наскоро набил неприлично длинное имя инсталлятора W_CC_P_7.0.073.exe в командной строке (необходимость в установке переменных TEMP и TMP объясняется тем, что в Windows 2000 они по умолчанию указывают на очень глубоко вложенный каталог, а инсталлятор Intel C++ — да и не только он — не поддерживает путей такого огромного размера).

Сразу же выяснилось, что политика защиты была кардинально пересмотрена и теперь наличие лицензии проверялось уже на стадии установки программы (в версии 5.x установка осуществлялось без проблем). OK, даем команду dir и смотрим на содержимое того, с чем нам сейчас предстоит воевать:

```
>dir
Содержимое папки C:\TMP\IntelC++Compiler70
17.03.2003  05:10        <DIR>           html
17.03.2003  05:11        <DIR>           x86
17.03.2003  05:11        <DIR>           Itanium
17.03.2003  05:11        <DIR>           notes
05.06.2002  10:35                45 056 AutoRun.exe
10.07.2001  12:56                    27 autorun.inf
29.10.2002  11:25                 2 831 ccompindex.htm
24.10.2002  08:12               126 976 ChkLic.dll
18.10.2002  22:37               552 960 chklic.exe
17.10.2002  16:29                28 663 CLicense.rtf
17.10.2002  16:35                   386 credist.txt
16.10.2002  17:02                34 136 Crelnotes.htm
19.03.2002  14:28                 4 635 PLSuite.htm
21.02.2002  12:39                 2 478 register.htm
02.10.2002  14:51                40 960 Setup.exe
02.10.2002  10:40                   151 Setup.ini
10.07.2001  12:56                   184 setup.mwg
              19 файлов       2 519 238 байт
               6 папок      886 571 008 байт свободно
```

Ага! Программа установки setup.exe занимает всего сорок с хвостиком килобайт. Очень хорошо! В такой объем серьезную защиту навряд ли спрячешь, а если даже так этот крохотный файл ничего не стоит проанализировать целиком до последнего байта дизассемблерного листинга. Впрочем, не факт, что защитный код расположен именно в setup.exe, он может находится и в другом месте, вот например... ChkLic.dll/ChkLic.exe, занимающими в совокупности немногим менее семисот килобайт. Постой, какой такой ChkLic? Это сокращение от Check License, что ли?! Гм, у ребят из Intel, очевидно, серьезные проблемы с чувством юмора. Уж лучше бы они назвали этот файл "Hack Me", честное слово! Ладно, судя по объему, ChkLic — это тот самый FLEX lm и есть, а с ним мы уже сталкивались (см. "Intel C++ 5.0 Compiler") и приблизительно представляем как его ломать.

Даем команду "dumpbin /EXPORTS ChkLic.dll" для исследования экспортируемых функций и... крепко держимся за Клаву, чтобы не упасть со стула:

```
Dump of file ChkLic.dll

File Type: DLL

  Section contains the following exports for ChkLic.dll

           0 characteristics
    3DB438B4 time date stamp Mon Oct 21 21:26:12 2002
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 000010A0 _CheckValidLicense
```

Черт побери! Защита экспортирует всего одну-единственную функцию с замечательным именем **CheckValidLicense**. "Замечательным" потому, что назначение функции становится понятным ее названия и появляется возможность избежать кропотливого анализа дизассемблерного кода. Ну вот, отбили весь интерес... Уж лучше бы они ее по ординалу экспортировали что ли, или, по крайней мере, окрестили ее каким ни будь отпугивающим именем типа DES Decrypt.

...размечтались! Ладно, вернемся к нашим баранам. Давайте рассуждать логически: если весь защитный код сосредоточен непосредственно в ChkLic.dll (а, судя по "навесному" характеру защиты, это действительно так), то вся "защита" сводится к вызову CheckValidLicense из Setup.exe и проверке возвращенного ею результата. Поэтому для "взлома" достаточно лишь пропадчить ChkLic.dll, заставляя функцию ChekValidLicense всегда возвращать... Да, кстати, что она должна возвращать? Точнее: какое именно возвращаемое значение соответствует успешной проверке лицензии? Нет, не торопитесь дизассемблировать setup.exe для определения ведь возможных вариантов не так уже и много: либо FALSE, либо TRUE. Вы делаете ставку на TRUE? Что ж, в каком-то смысле это логично, но с другой стороны — а почему мы, собственно, решили, что функция CheckValidLicense возвращает именно флаг успешности операции, а не код ошибки? Ведь должна же она как-то мотивировать причины отказа устанавливать компилятор: файл с лицензией не найден, файл поврежден, лицензия просрочена и так далее? Хорошо, попробуем возвратить ноль, а если это не прокатит, возвратим единицу.

ОК, пристегивайтесь, поехали! Запускаем HIEW, открываем файл ChkLic.dll (если же он не открывается — трижды помянув сусликов, временно скопируем его в корневую или любую другую директорию, не содержащую в своем имени спецсимволов, которые так не нравятся hiew'у). Затем, обратившись еще раз к таблице экспорта, полученной с помощью dumpbin, определяем адрес функции CheckValidLicense (в данном случае 010A0h) и через <F5>, "10A0" переходим в ее начало. Теперь режем по "живому", перезаписывая поверх старого кода `"XOR EAX, EAX/RETN 4"`. Почему именно `"REN 4"`, а не просто "RET"? Да потому, что функция поддерживает соглашение stdcall, о чем можно узнать, взглянув в HIEW'е на ее эпилог (просто пролистывайте экран дизассемблера вниз до тех пор, пока не встретите RET).

Проверяем... Это работает!!! Несмотря на отсутствие лицензии, инсталлятор, не задавая лишних вопросов, начинает установку! Стало быть, защита пала. Ой, не верится нам, что все так просто и чтобы не сидеть, тупо уставившись в монитор в ожидании завершения процесса инсталляции программы, мы натравливаем на setup.exe свой любимый дизассемблер IDA. Первое, что бросается в глаза — отсутствие CheckValidLicense в списке импортируемых функций. Может быть, она файл ChkLic.exe как-то запускает? Пробуем найти соответствующую ссылку среди автоматически распознанных строк: "~View Names", "ChkLic"... Ага, строки "Chklic.exe" здесь вообще нет, но зато обнаруживается "Chklic.dll". Понятно, значит, библиотека ChkLic загружается явной компоновкой через LoadLibrary. И переход по перекрестной ссылке подтверждает это:

```
.text:0040175D                  push    offset aChklic_dll ; lpLibFileName
.text:00401762                  call    ds:LoadLibraryA
.text:00401762 ; загружаем ChkLic.dll ^^^^^^^^^^^^^^^^
.text:00401762 ;
.text:00401768                  mov     esi, eax
.text:0040176A                  push    offset a_checkvalidlic ; lpProcName
.text:0040176F                  push    esi             ; hModule
.text:00401770                  call    ds:GetProcAddress
.text:00401770 ; получаем адрес функции CheckValidLicense
.text:00401770 ;
.text:00401776                  cmp     esi, ebx
.text:00401778                  jz      loc_40192E
.text:00401778 ; если такой библиотеки нет, то выходим из программы установки
.text:00401778 ;
.text:0040177E                  cmp     eax, ebx
.text:00401780                  jz      loc_40192E
.text:00401780 ; если такой функции в библиотеке нет, то выходим из установки
.text:00401780 ;
.text:00401786                  push    ebx
.text:00401787                  call    eax
.text:00401787 ; вызываем функцию ChekValidLicense
.text:00401787 ;
.text:00401789                  test    eax, eax
.text:0040178B                  jnz     loc_4019A3
.text:0040178  ; если функция возвратила не ноль, то выходим из программы установки
```

Невероятно, но эта до ужаса примитивная защита построена именно так! Причем, полуметровый файл ChkLic.exe вообще не нужен! И чего ради стоило тащить его из Интернета? Кстати, если вы надумаете сохранять дистрибьютив компилятора (внимание: я не говорил "распространять"!), то для экономии дискового места ChkLic.* можно стереть — либо пропадчив setup.exe, навсегда отучив его к ним обращаться, либо же просто создав свою собственную ChkLic.dll, экспортирующую stdcall функцию CheckValidLicence вида: `int CheckValidLicence(int some_flag) { return 0;}`

Так-с, пока мы все это обсуждали, инсталлятор закончил установку компилятора и благополучно завершил свою работу. Интересно запустится ли компилятор или все самое интересное только начинается? Лихорадочно спускаемся вниз по разветвленной иерархии вложенных папок, находим icl.exe, который как и следовало ожидать, находится в каталоге bin, нажимаем <ENTER> и... Компилятор, естественно, не запускается, ссылаясь на то, что "icl: error: could not checkout FLEX lm license", без которой он не может продолжить свою работу.

Выходит, что Intel применила многоуровневую защиту и первый уровень оказался грубой защитой от дураков. Что ж! Мы принимаем этот вызов и, опираясь на свой предыдущий опыт, машинально ищем файл LMGR*.DLL в каталоге компилятора. Бесполезно! На этот раз такого файла здесь не оказывается, зато выясняется, что icl.exe сильно прибавил в весе, перевалив за отметку шестиста килобайт... Стоп! А не прилинковали ли разработчики компилятора, этот самый FLEX lm статической компоновкой? Смотрим: в Intel C++ 5.0 сумма размеров lmgr327.dll и icl.exe составляла 598 Кб, а сейчас одни лишь icl.exe занимает 684 Кб. С учетом поправки на естественное старческое "ожирение", цифры очень хорошо сходятся. Значит, все-таки FLEX lm! Ой-ой! А ведь теперь, — без символических имен функций ломать защиту будет намного труднее... Впрочем, не будем раньше времени паниковать! Давайте думать, только спокойно! Навряд ли команда разработчиков полностью переписала весь код, взаимодействующей с этой "конвертной" защитой. Скорее всего, ее "усовершенствование" одной лишь сменой типа компоновки и закончилось. А раз так, то шансы взломать программу по прежнему велики!

Памятуя о том, что в прошлый раз защитный код находится в функции main, мы, определив ее адрес, просто устанавливаем точку останова и, дождавшись всплытия отладчика, тупо трассируем код, попеременно поглядывая то на отладчик, то на окно вывода программы: не появилась ли там ругательное сообщение? При этом, все встретившиеся нам условные переходы, мы отмечаем на отдельном листке бумаги (или откладываем в своей собственной памяти, если вы так хотите), не забыв указать выполнялся ли каждый условный переход или нет... Стоп! Что-то заболтались мы с вами, а ведь ругательное сообщение уже выскочило! OK, хорошо! Посмотрим, какой условный переход ему соответствовал. Наши записи показывают, что последним, встретившимся переходом, был условный переход JNZ, расположенный по адресу 0401075h и "реагирующий" на результат, возвращенной процедурой sub_404C0E:

```
.text:0040106E                  call    sub_404C0E
.text:00401073                  test    eax, eax
.text:00401075                  jnz     short loc_40107F
.text:00401077                  mov     al, 1
.text:00401079                  mov     byte ptr [esp+40h+var_18], al
.text:0040107D                  jmp     short loc_4010BA
.text:0040107F ; ---------------------------------
.text:0040107F
.text:0040107F loc_40107F:                              ; CODE XREF: _main+75↑j
.text:0040107F                  mov     eax, offset aFfrps ; "FFrps"
.text:00401084                  mov     edx, 21h
.text:00401089                  call    sub_404C0E
.text:0040108E                  test    eax, eax
.text:00401090                  jnz     short loc_40109A
```

Очевидно, что sub_404C0E и есть та самая защитная процедура, которая осуществляет проверку лицензии на ее наличие. Как ее обхитрить? Ну, тут много вариантов... Во-первых, можно, вдумчиво и скрупулезно проанализировать содержимое sub_404C0E на предмет выяснения: что именно и как именно она проверяет. Во-вторых, можно просто заменить `JNZ short loc_40107F` на `JZ short loc_40107F` или даже NOP, NOP. В-третьих, команду проверки результата возврата `TEST EAX, EAX` можно превратить в команду установки нуля: `XOR EAX, EAX`. В-четвертых, можно пропадчить

саму sub_404C0E, чтобы она всегда возвращала ноль. Не знаю, как вы, но мне больше всех приглянулся способ номер три. Меняем два байта и запускаем компилятор. Если никаких других проверок его "лицензионности" в защите нет, то программа заработает и, соответственно, наоборот. (Как мы помним, в пятой версии таких проверок было две). Поразительно, но компилятор больше не ругается и работает!!! Действительно, как и следовало ожидать — его разработчики ничуть не усилили защиту, а, напротив, даже ослабили ее!

# Record Now

...и угораздило же меня приобрести "писец" (то бишь CD-RW) в OEM-поставке! И ведь спрашивал продавца: а где, позвольте, тут пишущий софт или, по крайней мере, драйвера? На что продавец, удивленно так пожимая плечами, ответил: какие драйвера? Втыкаете — работает. А пишущие программы подходят любые, вот купите в соседнем магазине диск с Neuro CD. Мне — обладателю retail-"писца" от PHILPS, еще тогда это показалось странным, поскольку, я хорошо помнил, что диск с драйверами в коробке PHILPS'а был, а Easy CD Creator — непосредственно сам пишущий софт, — располагался совсем на другом диске. Но ведь как-то же справляются с OEM-продукцией другие люди, подумал я, и... купил.

Наскоро воткнув новехонький 40-скоростной NEC в свой компьютер, я был немало удивлен, когда Neuro CD наотрез отказался признать его "писцом". Не помог тут и Easy CD Creator, взятый с Филечкиного CD. Провозившись битый час и, ничего ровным счетом так и не выяснив, я, зверски разозленный на продавца, решил сделать ход конем, установив NEC на компьютер с "девяносто восьмой" Windows, вернув PHILPS'а себе. Никаких изменений! Собравшись было отдавать привод назад продавцу, я неожиданно вспомнил, что в одном из последних номеров Компьютер Пресс был обзор пишущих программ, причем демонстрационные версии всех этих программ содержались на прилагаемом к журналу компакт-диске. Из всех программ NEC'овый писец опознала лишь одна: **Record NOW**, которая, к счастью не имела никаких функциональных ограничений, исключая, правда 30-днейвый триальный период. Причем, программа оказалась такой уютной и удобной, что расставаться с ней мне не захотелось, но и расставаться со своими деньгами мне не хотелось тоже.

Как выглядит защита? При каждом запуске программа выводит противный nag-screen, напоминающий сколько дней ей еще "жить" осталось, и тем самым страшно нервирующий. Хорошо, ищем фразу "Number of days remaining in evaluation" во всех файлах программы и, если наша искалка поддерживает юникод, быстро выясняется, что данный текст содержится в файле lockers.dll, открыв который любым редактором ресурсов, мы обнаруживаем в нем тот самый заветный диалог! Остается выяснить: кто же выводит этот диалог на экран? Ищем строку "lockres.dll" во всех файлах программы. OK, это lockout.dll. Да... и эти разработчики не в ладах с юмором. Запускам dumpbin и смотрим список экспортируемых функций:

```
Dump of file lockout.dll

File Type: DLL

  Section contains the following exports for lockout.dll

           0 characteristics
    3C855E8D time date stamp Wed Mar 06 03:10:53 2002
        0.00 version
           1 ordinal base
          23 number of functions
          23 number of names
```

```
ordinal hint RVA      name

      3    0 0000CFF0 ?DESDecrypt@@YAKPBDPAD0@Z
      4    1 0000CC40 ?DESEncrypt@@YAKPBDPAD0@Z
      1    2 00003520 EvalModeTest
      2    3 00003930 EvalModeTestVB
      6    4 0000B230 _ezLICENSE_Check_Delphi@16
      7    5 0000B1A0 _ezLICENSE_Check_VB@16
      9    6 0000BC20 _ezLICENSE_ChkExpire_Delphi@16
     10    7 0000BB90 _ezLICENSE_ChkExpire_VB@16
     12    8 00009DB0 _ezLICENSE_ChkFileCRC_Delphi@8
     13    9 00009D40 _ezLICENSE_ChkFileCRC_VB@8
     15    A 0000BA30 _ezLICENSE_Clear_Delphi@12
     16    B 0000B9B0 _ezLICENSE_Clear_VB@12
     18    C 0000A320 _ezLICENSE_GetRestNumber_Delphi@16
     19    D 0000A290 _ezLICENSE_GetRestNumber_VB@16
     22    E 0000A6C0 _ezLICENSE_Upgrade_Delphi@20
     23    F 0000A610 _ezLICENSE_Upgrade_VB@20
      5   10 0000B2C0 ezLICENSE_Check
      8   11 0000BCA0 ezLICENSE_ChkExpire
     11   12 00009E20 ezLICENSE_ChkFileCRC
     14   13 0000BAA0 ezLICENSE_Clear
     17   14 0000A3B0 ezLICENSE_GetRestNumber
     20   15 00009C30 ezLICENSE_GetVersion
     21   16 0000A770 ezLICENSE_Upgrade
```

Сурово! Во-первых, обращает на себя внимание пара функция DES Encrypt/DES Decrypt, что-то (как и следует из ее названия) зашифровывающая/расшифровывающая. Во-вторых, тройственный подход к наименованию функций наводит на мысль, что мы имеем дело с "конвертной" защитой, разработанной независимо от защищенной с ее помощью программы и поддерживающий все основные языки программирования: Си/Си++, Дельфи и конечно же, Visual Basic, узнаваемый по суффиксу VB. В-третьих, такое обилие всевозможных проверочных функций предвещает, что исследование защиты и защищенной программы окажется делом отнюдь не легким! Причем, в те три сотни килобайт, которые занимает файл lockout.dll можно много всяких ловушек и хитростей понапхать, так что на скорый успех нам рассчитывать не приходится. Но... глаза страшатся, а руки делают. Начнем с того, что посмотрим какие именно функции защитной библиотеки использует программа.

Вот тебе и раз! Защищенная-то программа состряпана на Визуальном Бейсике, о чем красноречиво свидетельствует единственная, явно загружаемая ею библиотека MSVBVM60.DLL! Ах, так?! Хорошо, пойдем напролом. Просто удаляем lockout.dll из каталога программы и подсовываем ей любую другую DLL, предварительно переименованную в данную. Запускаем программу. На экране незамедлительно появляется сообщение об ошибке: среда Visual Basic'а ругается, что не может найти функцию EvalModeTestVB. Что ж, это уже кое-что! Загружаем lockout.dll в дизассемблер, находим в нем эту самую "Eval", быстро выясняем что она является "переходником" к EvalModeTest, которая... которая... Ой-ой-ой, которая занимает до черта килобайт и содержит в себе крайне запутанный с большим количеством глубоко вложенных друг в друга процедур программный код. Да чтобы проанализировать все это и месяца не хватит! А кто сказал, что этот код вообще следует анализировать?! Достаточно просто подсунуть

нужный код возврата и все! Весь вопрос в том какой именно код нужный. Беглый просмотр содержимого функции показал, что существуют как минимум три различных кода возврата: "0", "2" и "3". Если это так, то скорее всего одному из них соответствует состояние "программа не зарегистрирована, но лицензия еще не истекала", "программа не зарегистрирована и лицензия уже истекла", и, наконец, "программа зарегистрирована". Что ж, на перебор трех вариантов не уйдет много времени! Взяв в руки HIEW, переписываем код защитной функции "с нуля": `XOR EAX, EAX/RETN`.

Возвращаем lockout.dll на ее прежнее место, запускаем Record NOW и... не можем поверить свои глазам — программа исправно работает! "Исправно" — в том смысле, что nag-screen уже не выводится и по истечении положенных тридцати дней писец по-прежнему живет, а не умирает.

Хорошо, а если бы разработчик защищенного приложения, не поленился бы воткнуть проверку на успешность загрузки функции EvalModeTestVB и при ее отсутствии немотивированно прекращал свою работу? Смогли бы мы тогда узнать: какие функции библиотеки lockout используется, а какие нет? Уговорили! Взломаем программу другим путем! Подгоняем курсор к MyCDPro.exe и, нажав на <F3>, пытаемся найти lockout.dll прямым контекстным поиском. Вот, пожалуйста:

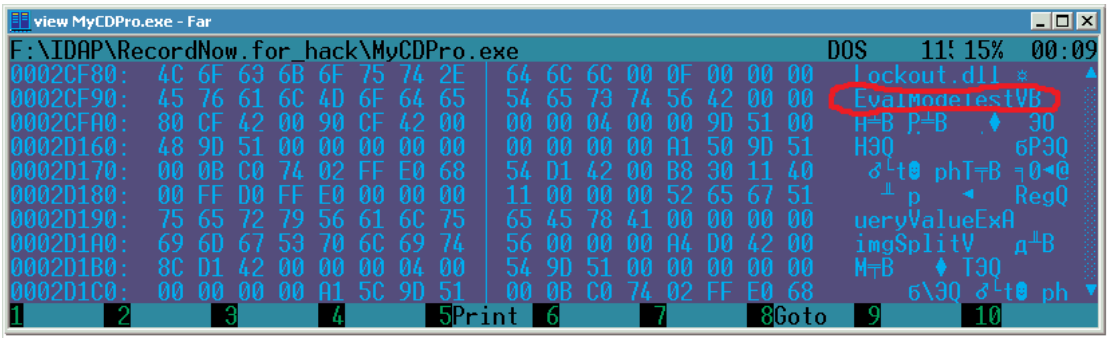


Рисунок 0x004. *Поиск ссылки на lockout.dll в защищенной программе*

Прямым текстом: "lockout.dll" и рядышком с ней EvalModeTestVB. Имена остальных защитных функций в исследуемой программе отсутствуют. Самое забавное, что в модуле lockout.dll присутствует огромное количество строк типа: "'*User has turned back their clock, so calculating days based on last and init*", "*The CRC file is valid*", "*Failed to update the Last Accessed time*", — т. е. защита составлена довольно грамотно и в состоянии как следует за себя постоять. Если, конечно, разработчик защищаемого приложения использовал все, предоставленные ей возможности, сполна. Увы, этого не произошло и на этот раз...

# UniLink v1.03 от Юрия Харона

Баста! Надоело! Все эти уродские защиты... (см. описания четырех предыдущих взломов) только портят настроение и еще, чего доброго, вызывают у читателей смутное сомнение: а не специально ли автор подобрал такие простые программы? Может быть, он, автор, вообще не умеет ничего серьезного ломать?! Уметь-то он (вы уж поверьте) умеет, но публично описывать взлом "серьезных" программ — боязно, а в "несерьезных" хороших защит мне как-то и не попадалось. Хотя, стоп! Ведь есть же такой программный продукт как UniLink, созданный опытнейшим системщиком Юрием Хароном (хорошо известным всем членам тусовки FIDO7.SU.C-CPP; если же вы никогда не заглядывали туда ранее, не поленитесь, сходите на Google, поднимите архив конференции и почитайте. Уверяю вас, вы не пожалеете). Достаточно сказать, что один лишь bag-list на UniLink — настоящий кладезь информации, перечисляющий больше количество ошибок операционной системы и ее окружения.

Наша цель — отучить UniLink ругаться на trial expired при запуске (из уважения к Харону необходимо отметить, что взлом проводится исключительно из спортивного интереса и природного любопытства. Какие либо корыстные цели тут не причем — линкер абсолютно бесплатен и может быть свободно скачан по следующему адресу: **ftp://ftp.styx.cabel.net/pub/UniLink/ulnbXXXX.zip**, где XXXX — номер версии). Цитирую со слов Харона "*Любая бета через полтора месяца начнёт "ругаться", что мол она expired :). Сделано это, просто как напоминание. в силу заинтересованности в том, что бы тестировались последнии билды*". Так что, ломая линкер помните, что взлом еще не освобождает от beta-тестирования ;-).

Несмотря на бесплатность линкера, Харон очень неплохо его защитил. Во всяком случае у меня на полный анализ защиты (включая развернутое описания взлома и отвлечения на повседневную текучку) ушла добрая неделя! Сейчас, когда пишутся эти строки, даже жалко, что защита так быстро сломалась и то интересное, во что еще можно вонзить свои зубы, закончилось. Впрочем, лучше отложим всю эту ностальгию до лучших времен в сторону, и вспомним, как эта неделя "эротических развлечений с защитой" собственно и начиналась...

...Привычным движением руки загружаем исполняемый файл линкера в свою любимую IDA 4.1.7. и... IDA грязно ругается по поводу того, что... "*can't find translation for virtual address 00000000, continue?*". Хм, ну что нам еще остается делать, — покорно жмем "Yes", чтобы сделать "continue". Увы! Наш фокус не увенчался успехом — на экране возникает еще одно ругательство "*File read error at 0004C7AC (may be bad PE structure), continue?*". Обречено жмем "Yes" и... ...IDA просто исчезает. Да-да! Именно **исчезает**, даже не успев перед смертью выдать сообщение о критической ошибке!!![2]

Интересный формат файла, однако! Пытаясь выяснить что же в нем содержится такое нехорошее, что так не понравилось IDA, мы решаем натравить на него утилиту dumpbin. Щас! Разбежались — при попытке вывести таблицу импорта, dumpbin выдает сообщение о внутренней ошибке "*DUMPBIN: error: Internal error during DumpImports*",

[2] В последующих версиях IDA это было исправлено.

и только что успев скинуть контекст, аварийно прекращает свою работу. Вот, значит, как?! Ну, защита, держись! Сейчас мы заглянем внутрь тебя "вручную" каким ни будь низкоуровневым инструментом. Ну, например, HIEW'ом...

Облом-с! При попытке сделать "prepare import data" HIEW скручивает дулю и, выдав нам на прощание трогательно красное окошко с надписью "Import name No free memory" банально виснет. Конкурирующий с ним QVIEW умирает и вовсе без каких либо пояснений. Утилита "PEDUMP" от Мэта Питтрека (известнейшего исследователя недр Windows) хоть и не виснет, но выдает сообщение о критической ошибке приложения и автоматически прибивается операционной системой. Так, чем еще можно исследовать внутренний формат PE-файла? На ум приходит **efd** (***E**xecutable **F**ile **D**umper*) от Ильфака, но даже эта утилита не справляется. — Выдав сообщение "*Can't find translation for 000002F6 (758.)*", она просто прекращает свою работу. **Dump PE** от Clive Turvey поступает аналогично. Дизассемблер **De Win** от Милюкова — виснет. **Win DASM** не виснет, но и не дизассемблирует. Даже знаменитый **PROCDUMP** распаковывать этот файл отказывается, правда позволяет сделать rebuild PE-заголовка, однако, после такой операции полученный файл становится неработоспособным. В общем, этот список можно продолжать бесконечно.

Кошмар! Защиты, срывающие крышу отладчику, — это я еще понимаю, но вот чтобы так агрессивно сопротивляться дизассемблеру! Причем, не какому-то одному, конкретно взятому дизассемблеру, а всем дизассемблерам сразу. И в это же самое время защита ухитряется работать в любой, Windows-совестимой операционной системе, включая NT и w2k, а, значит, никаких грязных хаков не использует. Харон по определению гений!

Вот мы и столкнулись с тем самым случаем, когда приходится дизассемблировать не готовым дизассемблером, а своими собственными руками и головой![3] Тяпнув для храбрости пивка, запускаем Иду и загружаем нашего подопытного в ***бинарном режиме***, то есть без анализа заголовков файла. Файл, естественно, успешно загружается. Теперь, открываем свой MSDN на странице "**Microsoft Portable Executable and Common Object File Format Specification**" и вдумчиво читаем все, что там написано. Без четкого представления о структуре и порядке загрузки PE-файлов, Харонову защиту нам ни за что не сломать. Если чтение фирменных спецификаций вызывает проблемы, попробуйте обратится к сторонним источникам. В том же MSDN содержится масса статей, посвященных исследованию PE-формата, в частности: "**The Portable Executable File Format from Top to Bottom**" by Randy Kath, русский перевод которой ("Исследование переносимого формата исполнимых файлов сверху вниз") легко найти в Сети. На худой конец можно обойтись и одним лишь заголовочным файлом WINNT.H, входящим в штатный комплект поставки любого windows-компилятора (но разобраться с "голым" WINNT.H сумеет лишь гений!)

Наша задача состоит в том, чтобы вручную проанализировать все заголовки, все секции и все поля исследуемого файла, пытаясь определить: что же такого необычного есть в каждом из них. Спрашиваете: "*необычное*" — это вообще как? Навскидку

[3] Вообще-то, анализировать PE-заголовок руками я ринулся чисто с перепугу. Тот же EXEVIEW от Randy Kath пусть и не совсем корректно обрабатывает защищенный файл, но по крайней мере не виснет и не завершает свою работу. К тому же он распространяется вместе с исходниками (см. MSDN) и у нас есть возможность оперативно исправить баг.

можно предположить по крайней мере три варианта: а) защита использует документированные, но малоизвестные возможности PE-файлов, не поддерживаемые распространенными дизассемблерами; б) защита использует недокументированные особенности (и/или поля) PE-файлов, не поддерживаемые дизассемблерами, но корректно обрабатываемые операционной системой; в) разночтения спецификаций PE-формата привели к тому, что разработчики ОС трактовали отдельные поля заголовков по-своему, а разработчики дизассемблеров — по-своему, в результате чего появилась возможность создать такой извращенный файл, корректно загрузить который сумеет одна лишь система, а все остальные исследовательские программы конкретно обломаются на его анализе.

Из пункта "а" со всей очевидностью следует, что для анализа защищенного файла одной лишь документации явно недостаточно, ведь нам требуется не только убедиться в соответствии всех полей исследуемого файла фирменной спецификации, но и выяснить насколько эти поля вообще типичны. Другими словами нам необходим практический опыт работы с PE-файлами, а если его нет, — что ж, возьмите несколько заведомо неизвращенных PE-файлов и основательно проштудируйте их от пола до потолка.

С пунктом "б" справится сложнее. Допустим, в фирменной спецификации такое-то поле помечено как неиспользуемое, а в защищенном файле здесь прописано некоторое значение. Как быть? (Дизассемблировать загрузчик операционной системы не предлагать). Да очень просто! Берем hiew старой версии — той, которая ничего не знает о PE и никак его не анализирует, и перебиваем "неиспользуемое" поле нулями или любым другим значением, пришедшимся нам по вкусу. Если это не нарушит работоспособности защищенного файла, — по всей видимости это поле действительно не используется и, соответственно, наоборот.

Пункт "в" еще более сложен. Никакие прямолинейные решения тут не действуют и все, что нам остается — вдумчиво читать каждую букву исходной спецификации и... нет! не стремиться "понять" ее, а пытаться представить себе: как она вообще должна быть понята, чтобы загрузчик операционной системы работал, а дизассемблер — нет. Дайте волю своему воображению, напрягите интуицию — всех многих тонкостей PE-форматов составители документации просто не описали. С другой стороны, сами разработчики ОС данный формат не с потолка брали и по тем же самым спецификациям его и реализовывали. Задумайтесь, а как бы вы реализовали загрузку PE-файла в память? Какие бы комбинации свойств PE-файла вы могли бы использовать для его защиты?

Первое, что нам приходит в голову — инициализация некоторых критических ячеек памяти посредством добавления их адреса в таблицу перемещаемых элементов. А что, это мысль! Особенно привлекательной в этом плане выглядит таблица перемещаемых элементов из old exe — заглушки, расположенной перед PE-файлом и большинством дизассемблеров просто игнорируемой. Но обращает ли системный загрузчик внимание на эти элементы или нет, — вот ведь в чем вопрос! Хорошо, давайте посмотрим на восстановленный old exe заголовок, извлеченный нами из защищенного файла.

```
seg000:00000000 ; OLD EXE HEADER
seg000:00000000   cc                    db 'MZ'
seg000:00000002   e_cblp                dw 405
seg000:00000004   e_cp                  dw 1
seg000:00000006   e_crlc                dw 0
seg000:00000008   e_cparhdr             dw 4
```

```
seg000:0000000A   e_minalloc              dw 33
seg000:0000000C   e_maxalloc              dw 33
seg000:0000000E   e_ss                    dw 16h
seg000:00000010   ccaaa                   dw 512
seg000:00000012   e_csum                  dw 0
seg000:00000014   e_ip                    dw 106
seg000:00000016   e_cs                    dw 0
seg000:00000018   e_lfarlc                dw offset RelocationTable
seg000:0000001A   e_ovno                  dw 0
seg000:0000001C   ae_res                  db 'UniLink!'
seg000:00000024   e_OEMid                 dw 0
seg000:00000026   e_OEMinfo               dw 1
seg000:00000028   e_res2                  db 14h dup(0)
seg000:0000003C   e_lfanew                dd offset IMAGE_NT_SIGNATURE_PE ; "PE"
```

Баста карапузики! Нас обломали! Никаких перемещаемых элементов в DOS-заглушке нет, о чем поле e_ovno красноречиво и свидетельствует (в дизассемблерном листинге оно выделено жирным шрифтом). Да и во всех остальных отношениях, old exe заголовок выглядит вполне корректным и приличным. Ладно, лиха беда начало! Отталкиваясь от значения поля e_lfanew, переходим по содержащемуся в нем смещению на заголовок PE-файла.

```
seg000:00000198 ; NEW EXE HEADER
seg000:00000198 IMAGE_NT_SIGNATURE_PE db 'PE',0,0       ; DATA XREF: seg000:0000003C
seg000:0000019C Machine               dw 14Ch           ; IMAGE_FILE_MACHINE_I386
seg000:0000019E NumberOfSection       dw 3              ; три секции
seg000:000001A0 TimeDateStamp         dd 3D4EE158h      ; временная метка
seg000:000001A4 PointerToSymbolTable  dd 0              ; указатель на таблицу символов
seg000:000001A8 NumberOfSymbols       dd 0              ; кол-во символов ноль, т.е. нет
seg000:000001AC SizeOfOptionalHeader  dw 0C0h           ; размер опционального заголовка
seg000:000001AC ;            а вот это  ^^^^  уже интересно: зная, за концом опциональ-
seg000:000001AC ; ного заголовка сразу же следуют заголовки сегментов, пытаемся проверить
seg000:000001AC ; корректность этого поля "на глаз": складываем 0x1B0 (начало опциональ-
seg000:000001AC ; ного заголовка) с 0xC0 (указанный размер заголовка) и получаем 0x270.
seg000:000001AC ; смотрим - по этому смещению в файле расположено слово ".text", значит,
seg000:000001AC ; размер заголовка указан правильно. Но… в то же самое время 0xC0 - это
seg000:000001AC ; крайне нетипичный размер для опционального заголовка и все исследуемые
seg000:000001AC ; мной файлы, содержали совсем другое значение, - а именно 0xE0. за счет
seg000:000001AC ; чего же "наш" заголовок оказался меньше? очевидно, защищенный файл со-
seg000:000001AC ; держит урезанный массив data directory, что теоретически должно воспри-
seg000:000001AC ; ниматься всеми дизассемблерами нормально, но вот полной увечности у нас
seg000:000001AC ; в этом нет. Как быть? Представляется логичным найти (или создать) PE-
seg000:000001AC ; файл с урезанной data directory и натравить на него дизассемблер (ту
seg000:000001AC ; же IDA) - интересно зависнет он или нет? А вот как создать такой файл,
seg000:000001AC ; не имея под руками соответствующего линкера? Просто пропадчить заголо-
seg000:000001AC ; вок в готовом PE-файле нельзя, т. к. за концом data directory загрузчик
seg000:000001AC ; ожидает увидеть каталог сегментов, а при "искусственном" уменьшении
seg000:000001AC ; размера заголовка там окажется "хвост" от data directory, что приведет
seg000:000001AC ; дизассемблер в сильное замешательство. "вырезать" кусочек data directo-
seg000:000001AC ; ry из файла так же невозможно, ведь при этом посыплются все смещения,
seg000:000001AC ; что так же приведет к непредсказуемой реакции дизассемблера при попыт-
seg000:000001AC ; ке анализа такого файла. А если… Постойте-ка! ведь можно просто сдви-
seg000:000001AC ; нуть каталог сегментов на место "освободившихся" после усечения заго-
seg000:000001AC ; ловка элементов data directory?! а знаете, это должно сработать! OK,
seg000:000001AC ; вооружившись hiew'ом усекаем размер заголовка любого заведомо нормаль-
```

```
seg000:000001AC ; ного файла до 0xC0 и перемещаем каталог сегментов на 0x20 байт "вверх".
seg000:000001AC ; Запускаем сам файл. Работает? Работает! Загружаем файл в дизассемблер…
seg000:000001AC ; Работает!!! OK, значит, размер заголовка в 0xC0 действительно допустим
seg000:000001AC ; продолжаем анализ….
seg000:000001AE Characteristics       dw 30Fh   ; IMAGE_FILE_RELOCS_STRIPPED|
seg000:000001AE                                 ; IMAGE_FILE_EXECUTABLE_IMAGE|
seg000:000001AE                                 ; IMAGE_FILE_LINE_NUMS_STRIPPED|
seg000:000001AE                                 ; IMAGE_FILE_32BIT_MACHINE |
seg000:000001AE                                 ; IMAGE_FILE_DEBUG_STRIPPED
seg000:000001AE ; атрибуты файла несколько нетипичны. обычно встречается 0x10F, а не 0x30F
seg000:000001AE ; (т.е. в нормальных файлах отсутствует флаг  IMAGE_FILE_DEBUG_STRIPPED
seg000:000001AE ; даже когда они не содержат никакой отладочной инфы), но с другой сто-
seg000:000001AE ; роны, так даже и правильнее. Эксперименты показывают, что исправление
seg000:000001AE ; 0x10F на 0x30F в остальных файлах (ес-но без дебужной инфы) проходит
seg000:000001AE ; безболезненно, значит, собака зарыта не здесь
```

Вот мы и выяснили, что PE-заголовок защищенного файла не содержит абсолютно ничего интересно, и если кто и завешивает HIEW и срывает IDA крышу, то уж точно не он. Что ж, сделав короткий перерыв (для "пивка"), продолжим наше утомительное исследование формата PE-файла, на сей раз взявшись за так называемый ***опциональный заголовок*** (***optional header***), следующий за концом PE-заголовка.

```
seg000:000001B0 ; ОПЦИОНАЛ ХИДЕР
seg000:000001B0 ; ==============
seg000:000001B0 Magic                   dw 10Bh         ; NORMAL EXE (все OK)
seg000:000001B2 MajorLinkerVersion      db 1            ; версия линкера
seg000:000001B3 MinorLinkerVersion      db 3            ; версия линкера
seg000:000001B4 SizeOfCode              dd 49817h       ; размер кода
seg000:000001B4                                         ; выглядит вполне нормально.
seg000:000001B4                                         ; т. е. при длине exe-файла в
seg000:000001B4                                         ; 0x4C7AA байт, потребности в
seg000:000001B4                                         ; 0x49817 байт вполне
seg000:000001B4                                         ; удовлетворяются
seg000:000001B4
seg000:000001B8 SizeOfInitializedData   dd 3008h        ; размер секции
seg000:000001B8                                         ; инициализированных данных
seg000:000001B8                                         ; выглядит вполне нормально
seg000:000001B8
seg000:000001BC SizeOfUninitializedData dd 0            ; нет секции
seg000:000001BC                                         ; неинициализированных данных
seg000:000001C0 AddressOfEntryPoint     dd 46673h       ; адрес точки входа
seg000:000001C4 BaseOfCode              dd 1000h        ; базовый адрес сегмента кода,
seg000:000001C4                                         ; забегая вперед, отметим,
seg000:000001C4                                         ; что этот адрес в точности равен
seg000:000001C4                                         ; адресу сегмента .text, так что
seg000:000001C4                                         ; тут все законно
seg000:000001C4
seg000:000001C8 BaseOfData              dd 4B000h       ; базовый адрес сегмента данных,
seg000:000001C8                                         ; проверка подтверждает его
seg000:000001C8                                         ; корректность
seg000:000001C8
seg000:000001CC ImageBase               dd 400000h      ; image base абсолютно нормальный
seg000:000001D0 SectionAlignment        dd 1000h        ; выравнивание секций по границе
seg000:000001D0                                         ; в 4Кб, что OK
seg000:000001D0
```

```
seg000:000001D4 FileAlignment           dd 200h         ; выравнивание файла по границе
seg000:000001D4                                         ; в 512 байт, что OK
seg000:000001D8 MajorSysVersion         dw 4            ; версия требуемой системы, OK
seg000:000001DA MinorSysVersion         dw 0            ; OK
seg000:000001DC MajorImageVersion       dw 1            ; версия приложения, OK
seg000:000001DE MinorImageVersion       dw 0            ; OK
seg000:000001E0 MajorSubsystemVersion   dw 4            ; версия подсистемы, OK
seg000:000001E2 MinorSubsystemVersion   dw 0            ; OK
seg000:000001E4 Win32VersionValue       dd 0            ; OK
seg000:000001E8 SizeOfImage             dd 52000h       ; размер образа файла в памяти
seg000:000001E8                                         ; выглядит вполне достоверно
seg000:000001E8
seg000:000001EC SizeOfHeaders           dd 400h         ; размер всех заголовков, OK
seg000:000001F0 CheckSum                dd 0            ; нет контрольной суммы, OK
seg000:000001F4 Subsystem               dd 3            ; кол-во секций, OK
seg000:000001F4                                         ; (дальше мы их все найдем)
seg000:000001F4
seg000:000001F8 SizeOfStackReserve      dd 100000h      ; кол-во резервируемой памяти
seg000:000001F8                                         : под стек, OK
seg000:000001F8
seg000:000001FC SizeOfStackCommit       dd 2000h        ; кол-во выделенной под стек
seg000:000001FC                                         ; памяти, OK
seg000:000001FC
seg000:00000200 SizeOfHeapReserve       dd 100000h      ; кол-во резервируемой под кучу
seg000:00000200                                         ; памяти, OK
seg000:00000200
seg000:00000204 SizeOfHeapCommit        dd 1000h        ; кол-во выделенной под кучу
seg000:00000204                                         ; памяти, OK
seg000:00000204
seg000:00000208 LoaderFlags             dd 0            ; не используется, OK
seg000:0000020C NumberOfRvaAndSizes     dd 0Ch          ; кол-во элементов в
seg000:0000020C                                         ; IMAGE_DATA_DIRECTORY
```

...и опциональный заголовок не содержит ничего интересного, но вот IMAGE DATA DIRECTORY, расположенная за ним следом, — дело другое и буквально с третий по счету строки мы выходим на след защиты:

```
seg000:00000210 IMAGE_DATA_DIRECTORY    dd 0            ; EXPORT dir
seg000:00000214                         dd 0
seg000:00000218
seg000:00000218 Import Table
seg000:00000218                         dd offset IMPORT_TABLE ;
```

Вот она — ссылка на таблицу импорта, ту самую таблицу, которая приводит к буйному замешательству огромное количество дизассемблеров и срывает крышу всем PE-утилитам вместе взятым. Посмотрим на нее?

```
seg000:0004B000 IMPORT_TABLE            dd 94010F0Eh    ; DATA XREF: seg000:00000218-o
seg000:0004B000                                         ; flags
seg000:0004B004                         dd 4000696h     ; date start
seg000:0004B008                         dd 54414C46h    ; foward index
seg000:0004B00C                         dd offset unk_39A39
seg000:0004B010                         dd 8965410h     ; import addres
seg000:0004B014
```

Пошла вода в хату! Оказывается, в таблице импорта вместо нормальных полей содержится какой-то голимый "мусор", который кое-что проясняет. С такой таблицей импорта дизассемблеры работать просто не могут и... если проверка корректности содержимого таблицы импорта отсутствует, они — виснут, в противном же случае, — аварийно прерывают свою работу с сообщением об ошибке.

Но это совершенно не объясняет как с такой защитой ухитряется работать загрузчик операционной системы? Уж не имеем ли мы дело с некоторыми недокументированными особенностями? Или, быть может, по этим "мусорным" адресам в оперативной памяти расположено что-то особенное? Последнее навряд ли! Поскольку защита успешно функционирует во всех windows-подобных системах, представляется сомнительным, что содержимое данных адресов всегда и везде одно и то же (кстати, беглая проверка отладчиком, это допущение с треском опровергает). Недокументированные возможности? Хм, непохоже... да если так — где прикажите искать реально импортируемые адреса?! Ладно, двигаемся дальше, может быть нам и повезет...

```
seg000:00000268 ; Bound Import
seg000:00000268                         dd offset bound_import_table
seg000:0000026C                         dd 1Ch
```

Ага! Держи Тигру за хвост! Защита использует документированное, но малоизвестное поле **bound import**, — представляющее собой альтернативный механизм импорта функций из DLL. Смотрим, что у нас там...

```
seg000:000002E8 ; bound import table
seg000:000002E8 TimeDateStamp          dd 0FFFFFFFFh         ; DATA XREF: seg000:0000268
seg000:000002EC OffsetModuleName       dw 0Eh                ; относительное смещение
seg000:000002EC                                              ; строки, содержащей имя
seg000:000002EC                                              ; импортируемой DLL
seg000:000002EC                                              ; 0x2E8 + 0xE == 0x2F6
seg000:000002EC                                              ; где мы обнаруживаем
seg000:000002EC                                              ; "kernel32.dll", что
seg000:000002EC                                              ; очевидно, уже не мусор!
seg000:000002EC
seg000:000002EE NumberOfModuleForward  dw 0                  ; ничего не импортируем?!
seg000:000002F0 Reserverd              dw 0
seg000:000002F2                        dd 0
seg000:000002F6 aKernel32_dll          db 'kernel32.dll',0   ; DATA XREF: seg000:049E0C
```

Вот **это** уже явно не мусор, а вполне удобоваримая таблица импорта, загружающая динамическую библиотеку kernel32.dll, и импортирующая.... Как это так — никаких функций?! Странно... Но ведь защита все-таки работает (пусть час от часу становится все менее и менее понятно ***как***). Хорошо, давайте рассуждать логически. Программ, не импортирующих никаких функций, под Windows NT существовать в принципе не может. Даже если защита использует native API (т. е. обращается к системным функциям напрямую через прерывание 2Eh), операционный загрузчик окажется не в состоянии загрузить такое приложение, поскольку ему необходимо, чтобы на адресное пространство загружаемого процесса была спроецирована библиотека kernel32.dll. Это в Windows 9x, где системные библиотеки автоматически отображаются на адресные пространства процессов, "голые" файлы работают безо всяких проблем, а в NT, отображающий только явно загруженные библиотеки, такой фокус уже не проходит. А, знаете, это многое объясняет! Теперь становится понятно в частности почему таблица импорта

не содержит в себе ни одной функции — они просто не нужны! Ссылка на kernel32.dll присутствует лишь затем, чтобы спроецировать эту библиотеку на адресное пространство процесса, как этого требует системный загрузчик. Хорошо, но как быть с "мусором" в стандартной таблице импорта? Как ни крути, а такие извращения системный загрузчик скорее удавится, чем обработает... Увы, нам нечего ответить на этот вопрос и, скрепя сердце, его вновь приходится откладывать, надеясь, что последующий анализ отделит свет от тьмы и все расставит по своим местам...

```
seg000:00000270 ; НАЧАЛО СЕГМЕНТОВ
seg000:00000270 a_text              db '.text',0,0,0
seg000:00000278 vir_size_text       dd 49817h           ; размер секции text в памяти
seg000:0000027C virt_addr_text      dd 1000h            ; адрес проекции на память
seg000:00000280 szRawData_text      dd 49810h           ; размер в файле
seg000:00000284 pRawData_text       dd 400h             ; смещение начала секции в файле
seg000:00000288 pReloc_text         dd 0
seg000:0000028C pLineNum_text       dd 0
seg000:00000290 nReloc_text         dw 0
seg000:00000292 nLineNum_text       dw 0
seg000:00000294 FLAG_TEXT           dd 60000020h        ; code | executable | readable
```

Вот мы и добрались до каталога сегментов! IMAGE HEADER секции ".text" выглядит вполне типично и никаких подозрений у нас не вызывает, но вот следующая за ним секция ".data" очень многое прояснеет...

```
seg000:00000298 a_data              db '.data',0,0,0
seg000:000002A0 vir_size_data       dd 3008h            ; размер секции .data в памяти
seg000:000002A4 vir_addr_data       dd 4B000h           ; адрес проекции на память
seg000:000002A8 szRawData_data      dd 14h              ; размер в файле
seg000:000002AC pRawData_data       dd 49E00h           ; смещение в файле
seg000:000002B0 pReloc_data         dd 0
seg000:000002B4 pLineNum_data       dd 0
seg000:000002B8 nReloc_data         dw 0
seg000:000002BA nLineNum_data       dw 0
seg000:000002BC FLAG_DATA           dd 0C0000040h       ; readable | writeable
```

"Ну и что здесь интересного?" — спросит иной читатель. А вот что — присмотритесь повнимательнее ***куда именно*** грузится содержимое данной секции. Если верить выделенной жирным шрифтом строке, — то по адресу IMAGE_BASE + 0x4B000. Ничего не напоминает? Во-первых, адрес 0x4B000 "волшебным" образом совпадает с адресом "мусорной" таблицы импорта (те, кто поимел сект с защитой этот адрес надолго запомнят, кстати, Харону не мешало бы его немножко замаскировать, чтобы он не так бросался в глаза). Во-вторых, изобразив процесс проецирования секций графически (см. рис. 0x005), мы с удивлением обнаружим, что секция .data расположена не следом за секцией .text (как это обычно и бывает), а находится ***внутри*** нее. Действительно, давайте подсчитаем: виртуальный адрес секции .text равен 0x1000, а ее размер — 0x49817, и последний байт секции приходится на адрес 0x59817, что превышает виртуальный адрес секции .data, равный 0x4B000.

Так вот оно что! Поскольку, секции отображаются на память в порядке их перечисления в каталоге (недокументированно, но факт!), то содержимое секции .data затирает область адресов 0x4B000 — 0x4E008! А что там у нас расположено?! ТАБЛИЦА ИМПОРТА!!! В дисковом файле по смещению 0x4B000 действительно расположен чи-

стейшей воды мусор (и это косвенно подтверждается тем, что изменения первых 0x14 байт работу программы не нарушают), а истинная таблица импорта расположена непосредственно в секции .data, которой соответствует смещение 0x49E00 дискового файла. Заглянем; что у нас там?!

```
seg000:00049E00 RealImportTable     dd offset IAT             ; OriginalFirstThunk
seg000:00049E04 TimeDateStamp1      dd 1
seg000:00049E08 ForwarderChain      dd 0FFFFFFFFh             ; no forward
seg000:00049E0C Name                dd offset aKernel32_dll   ; "kernel32.dll"
seg000:00049E10 FirstThunk          dd offset IAT
```

Вот, это действительно похожее на таблицу импорта со ссылкой на IAT. Кстати, не мешает посмотреть, что за функции импортирует IAT. Подгоняем курсор к "IAT" и, нажав, на <ENTER> смотрим:

```
seg000:0004B014 IAT                 dd 47440600h        ; DATA XREF: seg000:00049E00-o
seg000:0004B014                                         ; seg000:00049E10↑o
seg000:0004B018                     dd 50554F52h
seg000:0004B01C                     dd 69A8Bh
seg000:0004B020                     dd 0FF03FF11h
seg000:0004B024                     db    2             ;
seg000:0004B025                     db  4Ch             ; L
```

Мать родная! Ну почему ты не родишь меня обратно?! Опять вместо символических имен или на худой конец — ординалов, нам попадается этот проклятый мусор! Хотя, — подождите минуточку, — давайте попробуем определить что будет расположено по данному адресу после загрузки программы. Возвращаясь к описанию секции .data, мы обнаруживаем, что упустили один очень важный момент. Виртуальный размер секции .data (0x3008 байт) намного больше ее физического размера (0x14 байт) и потому, регион 0x4B014 — 49E008 будет заполнен нулями, а ведь "мусорная" IAT как раз и расположена по адресу 0x4B014! Следовательно, после загрузки ее содержимое окажется заполнено одними нулями, что соответствует пустой таблице импорта функций. Фу-х! Невероятно, но мы действительно в этом разобрались!!! Кстати, подобный прием и широко используется авторами упаковщиков исполняемых файлов.

```
seg000:000002C0 seg000:000002C0 b_rsrc  db '.rsrc',0,0,0
seg000:000002C8 vir_size_rsc            dd 27ACh          ; размер секции rsrc в памяти
seg000:000002CC vir_addr_rsc            dd 4F000h         ; адрес проекции на память
seg000:000002D0 szRawData_rsc           dd 27ACh          ; размер в файле
seg000:000002D4 pRawData_rsc            dd 4A000h         ; смещение секции в файле
seg000:000002D8 pReloc_rsc              dd 0
seg000:000002DC pLineMun_rsc            dd 0
seg000:000002E0 nReloc_rsc              dw 0
seg000:000002E2 nLineNum_rsc            dw 0
seg000:000002E4 FLAG_RSC                dd 50000040h      ; initalized data |
seg000:000002E4                                           ; shareable |readable
```

Аналогичным образом поступает и секция .rsrc, внедрясь в середину секции .text (но секцию .data она не перекрывает), причем, для ослепления некоторых дизассемблеров тут используется еще один хитрый примем: указанный "физический" размер секции .rsrc "вылетает" за пределы дискового файла. Системному загрузчику — хоть бы что, а вот некоторые исследовательские утилиты от этого и крышей поехать могут.

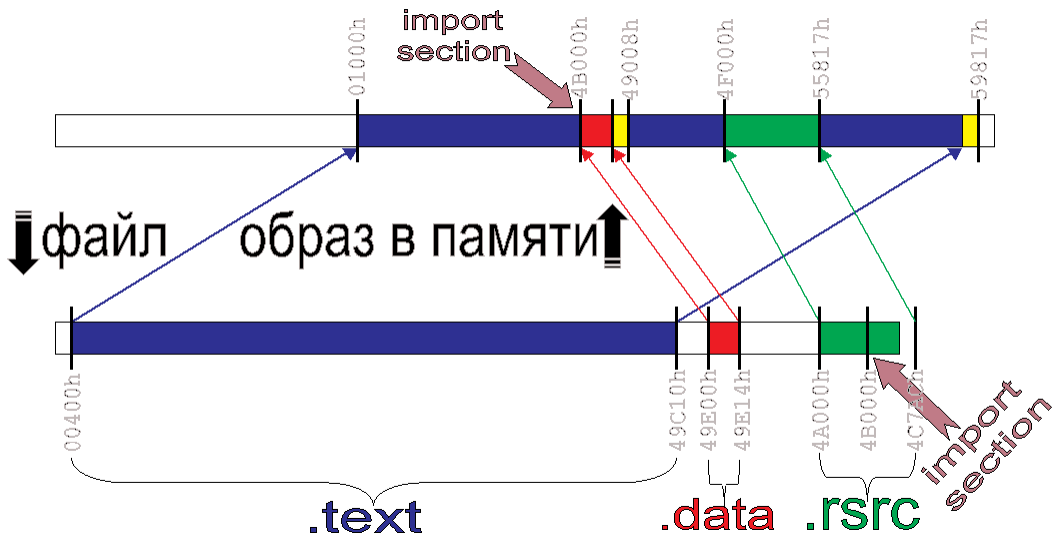


Рисунок 0x005. *Динамическое замещение таблицы импорта в процессе загрузки PE-файла*

Настало время проверить наши предположения на практике. Давайте загрузим эту извращенную программу отладчиком и посмотрим что содержится в памяти по адресу IMAGE_BASE + 0x4B000 = 0x44B000: мусор или нормальная таблица импорта? Отладчик soft-ice (как это и следовало ожидать) обламывается с отладкой этого извращенного файла, просто проскакивая точку входа, а вот WDB сполна оправдывая репутацию фирмы Microsoft (это не ирония!), пусть и не без ругательств, но все-таки загружает наш подопытный файл и послушно останавливается в точке входа.

```
Module Load: F:\IDAP\HARON\ulink.exe  (symbol loading deferred)
Thread Create:  Process=0, Thread=0
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (could not open symbol file)
Module Load: F:\IDAP\HARON\ulink.exe  (could not open symbol file)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (could not open symbol file)
Stopped at program entry point
```

Обратите внимание на выделенную жирным шрифтом строку. Отладчику показалось, что отлаживаемая программа импортирует некоторые функции... из самой себя! Но мы-то, излазившие защищенный файл вдоль и поперек, хорошо знаем, что за исключением kernel32.dll, никаких других экспортируемых и/или импортируемых библиотек здесь нет и такое поведение отладчика, судя по всему, объясняется все тем же самым "мусором". OK, переключаем свое внимание на окно с дампом памяти, заставляя ее отобразить содержимое таблицы импорта:

```
0x0023:0x0044B000 14 b0 04 00 01 00 00 00 ff ff ff ff f6 02 00 00 ................
0x0023:0x0044B010 14 b0 04 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0023:0x0044B020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Ура! Открываем на радостях пиво! Содержимое памяти доказательно подтверждает, что загрузка файла действительно происходит именно так, как мы и предполагали! Хорошо, но что же нам теперь делать? То бишь, найти-то причину помешательства дизассемблеров мы нашли, но вот как ее нейтрализовать? Ну, это не вопрос! Достаточно

лишь скопировать 0x14 байт памяти с адреса 0x49E00 по адресу 0x4B000 и скорректировать ссылку на IAT, направив ее на любое, заполненное нулями, место.

...HIEW теперь заглатывает защищенную программу и даже не пикает! А IDA... а IDA по прежнему отказываться обрабатывать этот файл и с завидным упорством слетает. В чем же причина? Вы, конечно, будете смеяться, но истинный виновник есть ни кто иной как Microsoft! Если бы не ее жутко прогрессивная платформа NET... А, впрочем, чего это я разворчался? Сами смотрите:

```
(•) Microsoft.Net assembly [pe.ldw]
( ) Portable executable for IBM PC (PE) [pe.ldw]
( ) MS-DOS executable (EXE) [dos.ldw]
( ) Binary file
```

Вот это да! С роду такого не было! Чтобы IDA да не правильно опознала формат файла!!! Перемещаем радио-кнопку на одну позицию вниз (ведь мы имеем дело отнюдь не с Microsoft Net assembly, а с PE!) и... IDA успешно открывает файл. Причем, с восстановлением таблицы импорта можно было и не возиться, — IDA просто ругнулась на мусор и все! Но кто ж знал?! Задним умом все мы крепки...

Короче, возвращаясь к нашим баранам (в данном случае — к терпеливо ожидающему нас отладчику) в точке входа, дизассемблерный текст выглядит так:

```
00446673 55              push    ebp
00446674 68AECF4200      push    42CFAEh
00446679 8BDC            mov     ebx,esp
0044667B 2403            and     al,3
0044667D 7203            jb      00446682
0044667F FE4302          inc     byte ptr [ebx+2]
00446682 D7              xlat    byte ptr [ebx]
00446683 27              daa
00446684 81042453970000  add     dword ptr [esp],9753h
0044668B 1AC9            sbb     cl,cl
0044668D 9F              lahf
0044668E FF33            push     dword ptr [ebx]
00446690 FC              cld
00446691 C3              ret
```

Не очень-то это похоже на осмысленный код программы! Может быть, это снова мусор? Маловероятно, — ведь отладчик использует штатный системный загрузчик PE-файлов и потому показывает образ файла таким, какой он в действительности есть, ну... если, конечно, защита тем или иным образом не противостоит отладке. Ладно, отставив разговорчики в строю, начинам трассировать код и... с первых же строк впадаем в некоторое замешательство. Защита опрашивает начальное значение регистра EAX, которое (если верить отладчику!) как будто бы равно нулю, но полной уверенности в этом у нас нет. Еще со времен старушки MS-DOS многие отладчики славились тем, что самостоятельно инициализировали регистры после загрузки, чем и выдавали себя (в частности, при нормальной загрузке файла регистр SI содержал в себе адрес первой исполняемой команды, а при загрузке под отладчиком Turbo Debugger и иже с ним, был равен нулю). Вообще-то, закладываться на "предопределенные" значения регистров — дурной тон. Никто не гарантирует, что в следующих версиях Windows что ни будь не изменится, и если такое вдруг произойдет, то защита откажет в работе, обломав не только хакеров, но и легальных пользователей. Впрочем, начальное значение регистра EAX (AX) по жизни равно нулю, и с некоторой натяжной за это можно зацепиться.

Далее защита непонятно зачем увеличивает старшее слово, только что закинутое в стек, на единицу и вызывает абсолютно бесполезные команды XLAT, DAA, ADD, SBB и... загружает регистр флагов в EAX. Уж не пытается ли она этим самым обнаружить флаг трассировки? Затем делает RETN для передачи управления по адресу: (0x42CFAE + 0x10000) + 0x9753 == 0x446701

```
.text:00446701          mov           edi, esi
.text:00446703          mov           esi, ebx
.text:00446705          sub           dword ptr      [esi], 1006Fh
.text:0044670B          lodsw
.text:0044670D          bswap         eax
.text:0044670F          inc           byte  ptr      [esi]
.text:00446711          lodsb
.text:00446712          mov           ah, al
.text:00446714          lodsb
.text:00446715                  bswap   eax
.text:00446717                  mov     ebp, eax
.text:00446719                  movzx   ecx, cl
.text:0044671C                  push    dword ptr [ebp+6Bh]
.text:0044671F                  lea     eax, [esi-8]
.text:00446722                  xchg    eax, fs:[ecx]
.text:00446725                  mov     edx, eax
.text:00446727                  inc     edx
.text:00446728                  jz      short loc_44672D
.text:0044672A                  mov     edx, [eax+4]
.text:0044672D
.text:0044672D loc_44672D:                              ; CODE XREF: .text:00446728j
.text:0044672D                  xchg    eax, [esp]
.text:00446730                  pushf
.text:00446731                  lea     ebx, [eax+21ADFh]
.text:00446737                  jnz     short loc_446745
.text:00446739                  lea     edi, [edi+0ACh]
.text:0044673F                  mov     dword_44CAF8, edi
.text:00446745
.text:00446745 loc_446745:                              ; CODE XREF: .text:00446737j
.text:00446745                  bts     dword ptr [esi-0Ch], 8
.text:0044674A                  jb      short loc_446753
.text:0044674C                  popf
.text:0044674D                  call    $+5
.text:00446752                  retf
```

...отладчик доходит лишь до RETF и после этого сразу же "дохнет". К тому же, остается совершенно непонятным, что же собственно делает этот запутанный и витиеватый код? При желании, конечно, с ним можно разобраться, но... нужно ли? Ведь отладить нашу подопытную мы все равного не сможем, во всяком случае в WDB.

Хорошо, зайдем с другого конца. Предположим, что программа работает с операционной системой не напрямую (через native API), а через подсистему win32 (win32 API). Тогда, установив точку останова на любую API-функцию, вызываемому программой, мы автоматически попадем в гущу "нормального" программного кода, уже распакованного (расшифрованного?) защитой. Весь вопрос в том: какие именно API-функции вызывает программа. Ну, пусть это будет GetVersion, с вызова которой начинается стартовый код практически любой программы. Запускаем soft-ice, нажимаем <Ctrl-D>, даем команду "bpx GetVersion", выходим из отладчика, вызываем unlink.exe и... ничего не происходит!

Отладчик не всплывает! Выходит, исследуемая нами программа не использует GetVersion! Что ж, удаляем предыдущую точку останова и пытаемся "забрейкать" CreateFileA (ну должен же линкер как-то открывать файлы!!!). Так, <Ctrl-D>, bpx CreateFileA<ENTER>, x<ENTER>... Ура! Это срабатывает! Отладчик перехватывает вызов защищенной программы и, после выхода из тела CreateFileA по команде P RET (в CreateFileA для нас действительно нет ничего интересного), мы оказывается в следующем коде:

```
001B:00416DEB  CALL    [USER32!CharToOemBuffA]
001B:00416DF1  PUSH    00000104
001B:00416DF6  LEA     EAX,[ESP+08]
001B:00416DFA  PUSH    EAX
001B:00416DFB  LEA     EDX,[ESP+0C]
001B:00416DFF  PUSH    EDX
001B:00416E00  CALL    [KERNEL32!GetShortPathNameA]
001B:00416E06  TEST    EAX,EAX
001B:00416E08  JZ      00416E2B
001B:00416E0A  LEA     EDX,[ESP+04]
001B:00416E0E  PUSH    00
001B:00416E10  PUSH    27
001B:00416E12  PUSH    03
001B:00416E14  PUSH    00
001B:00416E16  PUSH    01
001B:00416E18  PUSH    80000000
001B:00416E1D  PUSH    EDX
001B:00416E1E  CALL    [KERNEL32!CreateFileA]
001B:00416E24  MOV     EBX,EAX
001B:00416E26  CMP     EBX,-01
001B:00416E29  JNZ     00416E35
001B:00416E2B  CALL    [KERNEL32!GetLastError]
001B:00416E31  MOV     ESI,EAX
001B:00416E33  JMP     00416E5B
```

Обратите внимание: несмотря на отсутствие таблицы импорта, программа каким-то загадочным образом все-таки импортирует из kernell32.dll все, необходимые ей API-функции. Очень хорошо! Секс с native API и прочими извратами программистской хитрости отменяется! И мы остаемся в среде привычной нам подсистемы win32 API. Как именно осуществляется импорт — вот это уже другой вопрос! Кстати, давайте заглянем в одну такую функцию дизассемблером:

```
.text:00416E18            push 80000000h
.text:00416E1D            push edx
.text:00416E1E            call dword_44CC20 ; в отладчике это было KERNEL32!CreateFileA
.text:00416E24            mov  ebx, eax
.text:00416E26            cmp  ebx, 0FFFFFFFFh
.text:00416E29            jnz  short loc_416E35
...
.data:0044CC14 dword_44CC14   dd ?                    ; DATA XREF: sub_416DA0+AD?r
.data:0044CC14                                        ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18   dd ?                    ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C   dd ?                    ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20   dd ?                    ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                                        ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24   dd ?                    ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                                        ; sub_416F3C+128↑r
```

```
.data:0044CC28 dword_44CC28    dd ?                    ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                                         ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C    dd ?                    ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                                         ; sub_41AD20+12E↑r ...
.data:0044CC30 dword_44CC30    dd ?                    ; DATA XREF: .text:004014C4↑r
.data:0044CC34 dword_44CC34    dd ?                    ; DATA XREF: sub_419DD8+31↑r
.data:0044CC34                                         ; .text:0041A3E5↑r ...
.data:0044CC38 dword_44CC38    dd ?                    ; DATA XREF: sub_419DD8+1E↑r
.data:0044CC38                                         ; .text:0041A3A4↑r ...
```

Смотрите! В дисковом файле адресов импортируемых функций просто ***нет*** и таблица импорта судя по всему заполняется защитой динамически. А это значит, что в дизассемблере мы просто не сможем разобраться: какая именно функция в какой точке программы вызывается. Или... все-таки сможем?! Достаточно просто скинуть импорт работающей программы в дамп, а затем просто загрузить его в IDA! Затем, отталкиваясь от адресов экспорта, выданных "dumpbin /EXPORTS kernel32.dll", мы без труда приведем таблицу импорта в нормальный вид. Итак, прокручивая экран дизассемблера вверх, находим где у этой таблицы расположено ее начало или нечто на него похожее (если мы ошибемся — ничего странного не произойдет, просто часть функций останется нераспознанными и когда мы с ними столкнемся лицом к лицу, эту операцию придется повторять вновь). Вот, кажется, мы нашли, что искали, смотрите:

```
.data:0044CC09                                         ; sub_43E6D4+22A↑r ...
.data:0044CC0A                 db ?                    ; unexplored
.data:0044CC0B                 db ?                    ; unexplored
.data:0044CC0C                 db ?                    ; unexplored
.data:0044CC0D                 db ?                    ; unexplored
.data:0044CC0E                 db ?                    ; unexplored
.data:0044CC0F                 db ?                    ; unexplored
.data:0044CC10                 db ?                    ; unexplored
.data:0044CC11                 db ?                    ; unexplored
.data:0044CC12                 db ?                    ; unexplored
.data:0044CC13                 db ?                    ; unexplored
.data:0044CC14 dword_44CC14    dd ?                    ; DATA XREF: sub_416DA0+AD↑r
.data:0044CC14                                         ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18    dd ?                    ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C    dd ?                    ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20    dd ?                    ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                                         ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24    dd ?                    ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                                         ; sub_416F3C+128↑r
.data:0044CC28 dword_44CC28    dd ?                    ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                                         ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C    dd ?                    ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                                         ; sub_41AD20+12E↑r ...
```

Условимся считать адрес 0044CC14h ***началом***. Используя точку останова на CreateFileA, вновь вламываемся в программу и, отключив окно "data" командой wd, скидываем таблицу импорта в хистори: "d 44CC14". Выходим из Айса, запускаем NuMega Symbol Loader и записываем историю команд в файл winice.log (или любой другой по вашему вкусу). И как со всем этим нам теперь работать? Рассмотрим это на примере функции "call dword_44CC78". Прежде всего мы должны выяснить, какое значение находится в загруженной программе по адресу: 0x44CC87. Открываем winice.log по <F3> и смотрим:

```
0010:0044CC78 77E8668C  77E8F51E  77E93992  77E8DBF8      .f.w...w.9.w...w
0010:0044CC88 77E93F05  77E85493  77E87BE4  77E87D16      .?.w.T.w.{.w.}.w
0010:0044CC98 77E8C0A6  77E8AF8E  77E8878A  77E8BDE8      ...w...w...w...w
0010:0044CCA8 77E94911  77E9499C  77E9138C  77E8D019      .I.w.I.w...w...w
```

Теперь, обратившись к таблице экспорта kernel32.dll, определяем: а) базовый адрес ее загрузки (в данном случае: 0x77E80000); б) имя функции, сумма RVA и IMAGE BASE которой совпадает со значением 0x77E8668C. Вычитаем из 0x77E8668C базовый адрес загрузки — 0x77E80000 и получаем: 0x668C. Ищем строку 0x668C простым контекстным поиском и...

```
302   12D 0000668C GetLastError
```

...это оказывается ни кто иной, как GetLastError, что и требовалось доказать. Конечно, восстанавливать весь импорт вручную — крайне скучно и утомительно. Но кто нам сказал, что мы должны это делать именно вручную?! Ведь дизассемблер IDA поддерживает скрипты, что позволяет автоматизировать всю рутинную работу (подробнее о языке скрпитов можно прочитать в книге "Образ мышления — дизассемблер IDA" от Криса Касперски, то есть, собственно, меня).

OK, еще один барьер успешно взять. Воодушевленные успехом и доверху наполненные выпитым во время хака пивом, мы продолжаем! В плане возвращения к нашим баранам, сосредоточим свои усилия на загрузчике таблице импорта, расположенном по всей видимости где-то недалеко от точки входа. Несмотря на то, что soft-ice, по-прежнему, упорно проскакивает Entry Point, обламываясь с загрузкой защищенного файла (впрочем, другие версии soft-ice с этим справляются на ура), мы можем легко обхитрить защиту просто воткнув в точку входа бряк поинт. Поскольку, бряк поиск должен устанавливаться во вполне определенном контексте, используем уже известную нам нычку с CreateFileA. Итак, "bpx CreateFileA", <Ctrl-D>, запускаем unlink и, когда soft-ice "всплывает" даем: "bpx 0x446673" (адрес точки входа), выходим из soft-ice и... запускаем ulink вновь. Отладчик тут же всплывает:

```
001B:00446673  55              PUSH    EBP
001B:00446674  68AECF4200      PUSH    0042CFAE
001B:00446679  8BDC            MOV     EBX,ESP
001B:0044667B  2403            AND     AL,03
001B:0044667D  7203            JB      00446682
001B:0044667F  FE4302          INC     BYTE PTR [EBX+02]
001B:00446682  D7              XLAT
001B:00446683  27              DAA
```

Знакомые места! Трассируем код до тех пор пока на не встретится подозрительный RETF (от RET FAR — далекий возврат), передающий управление по следующему адресу:

```
001B:77F9FB90  8B1C24          MOV     EBX,[ESP]
001B:77F9FB93  51              PUSH    ECX
001B:77F9FB94  53              PUSH    EBX
001B:77F9FB95  E886B3FEFF      CALL    77F8AF20
001B:77F9FB9A  0AC0            OR      AL,AL
001B:77F9FB9C  740C            JZ      77F9FBAA
001B:77F9FB9E  5B              POP     EBX
001B:77F9FB9F  59              POP     ECX
```

Судя по адресу, этот код принадлежит непосредственно самой операционной системе (а точнее — NTDLL.DLL) и представляет собой функцию **KiUserExceptionDispatcher**. Но что это за функция? Ее описание отсутствует в SDK, но поиск по MSDN

обнаруживает пару статей Мета Питтрека, посвященных механизмам функционирования SEH и функции KiUserExceptionDispatcher в частности.

Структурные исключения! Ну конечно же! Какая защита обходится без них! Ладно, разберемся, ворчим мы себе под нос, продолжая трассировку защиты дальше. Увы! В той же точке, где WDB терял над программой контроль, soft-ice просто слетает. Ах, вот значит как!!! Ну, защита, держись!!!

(продолжение следует)