obj файлы на топчане или линковка дизассемблерных файлов

крис касперски ака мыщъх, no-email

в прошлой статье этого цикла мы подошли к тому, что ассемблировали дизассемблерный листинг, поборов все ошибки транслятора и получив в итоге… неработоспособный obj файл, вызывающий у ликера хроническое недержание error'ов. сегодня мы продолжим заниматься извращенным сексом (кама-сутра отдыхает), обогащаясь новыми знаниями и пополняя свой словарный запас матерных слов

введение

Освежая в памяти события давно минувших дней (уже листья успели облететь за это время), напомним, что исправив кучу багофичей IDA Pro (перечисление которых заняло бы слишком много места) мы дошли до файла demo_3.asm, который нас удалось ассемблировать MASM'ом, со следующими ключам:

```
ML.EXE /coff /I. /c /Cp /Zp1 /Zm demo_3.asm
```

Листинг 1 ключи для транслирования ассемблирования дизассемблерного листинга, сгенерированного IDA Pro

Здесь: **/coff** – создавать obj-файл в формате coff (иные форматы ms link не поддерживает, а искать другие линкеры нам в лом), **/I.** – искать включаемые файлы в текущей директории, **/c** – только ассемблировать, не линковать (линковать мы будем вручную, с помощью мата, кувалды и, конечно, хвоста), **/Cp** – учитывать регистр символов, **/Zp1** – выравнивание для структур, **/Zm** – режим совместимости с MASM 5.10, в формате которого IDA Pro и создает листинги.

битва за API

Транслятор MASM (входящий, в частности, в состав NTDDK) не выдает не единой ошибки (ну да! еще бы!!! чуть хвост не треснул, пока мы их вылавливали) и генерирует obj-файл. Наступает волнующее время линковки:

```
$link.exe demo_3.obj
Microsoft (R) Incremental Linker Version 5.12.8181
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

LINK:fatal error LNK1221: a subsystem can't be inferred and must be defined
```

Листинг 2 попытка линковки только что ассемблированного файла

Линкер материться, что подсистема не задана и ни хвоста линковать не хочет. Ну это даже не вопрос! Подсистема задается через ключ **/SUBSYSTEM** за которым следует одно из следующих ключевых слов: NATIVE — для драйверов, WINDOWS — для GUI приложений, CONSOLE — для консольных приложений, WINDOWSCE — для платформы Windows CE, POSIX – э… ну… это эдакая пародия на UNIX, но все равно ни хрена не работающая.

Фактически выбирать приходится между WINDOWS и CONSOLE. Чем они отличаются? С точки зрения PE-формата, одним битом в заголовке, указывающему системному загрузчику создавать или не создавать консоль при запуске файла. Попытка линковки консольного файла как GUI заканчивается фатально (консоль не создается и весь ввод/вывод обламывается). Обратное не столь плачевно, но пустое консольное окно на фоне GUI выглядит как-то странно. Но мы-то знаем, что наше приложение — консольного типа, следовательно пишем:

```
$link /SUBSYSTEM:CONSOLE demo_3.obj
Microsoft (R) Incremental Linker Version 5.12.8181
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

demo_3.obj:error LNK2001: unresolved external symbol _WriteFile
demo_3.obj:error LNK2001: unresolved external symbol _GetVersion
```

```
…
demo_3.obj:error LNK2001: unresolved external symbol _SetStdHandle
demo_3.obj:error LNK2001: unresolved external symbol _CloseHandle
demo_3.exe:fatal error LNK1120: 40 unresolved externals
```

Листинг 3 подсистема определена, но линкер не может найти API-функции

Хорошая новость — линкер заглатывает наживку и пытается переварить файл. Плохая новость — это у него не получается. А не получается потому, что он не распознает имена API функций, которых в нашем демонстрационном примере аж целых 40 штук! В переводе с английского ругательство "error LNK2001: unresolved external symbol _WriteFile" звучит как "Егор: LNK2001: неразрешимый внешний символ _WriteFile".

Сразу же возникает вопрос: откуда взялся знак прочерка и почему это WriteFile вдруг стала неразрешимым символом?! Смотрим в ассемблерный листинг. Контекстный поиск по "_WriteFile" ничего не дает! API-функция там объявлена _без_ знака прочерка:

```
; Segment type:        Externs
; BOOL __stdcall WriteFile(HANDLE hFile, LPCVOID lpBuffer,
; DWORD nBytesToWrite, LPDWORD lpNumberOfBytesWritten,LPOVERLAPPED lpOverlapped);
        extrn WriteFile:dword
```

Листинг 4 объявления API-функций в ассемблерном листинге

А теперь открываем demo_3.obj в любом hex-редакторе (например, в FAR'e по <F3> или в HIEW'e) и повторяем процедуру поиска еще раз:

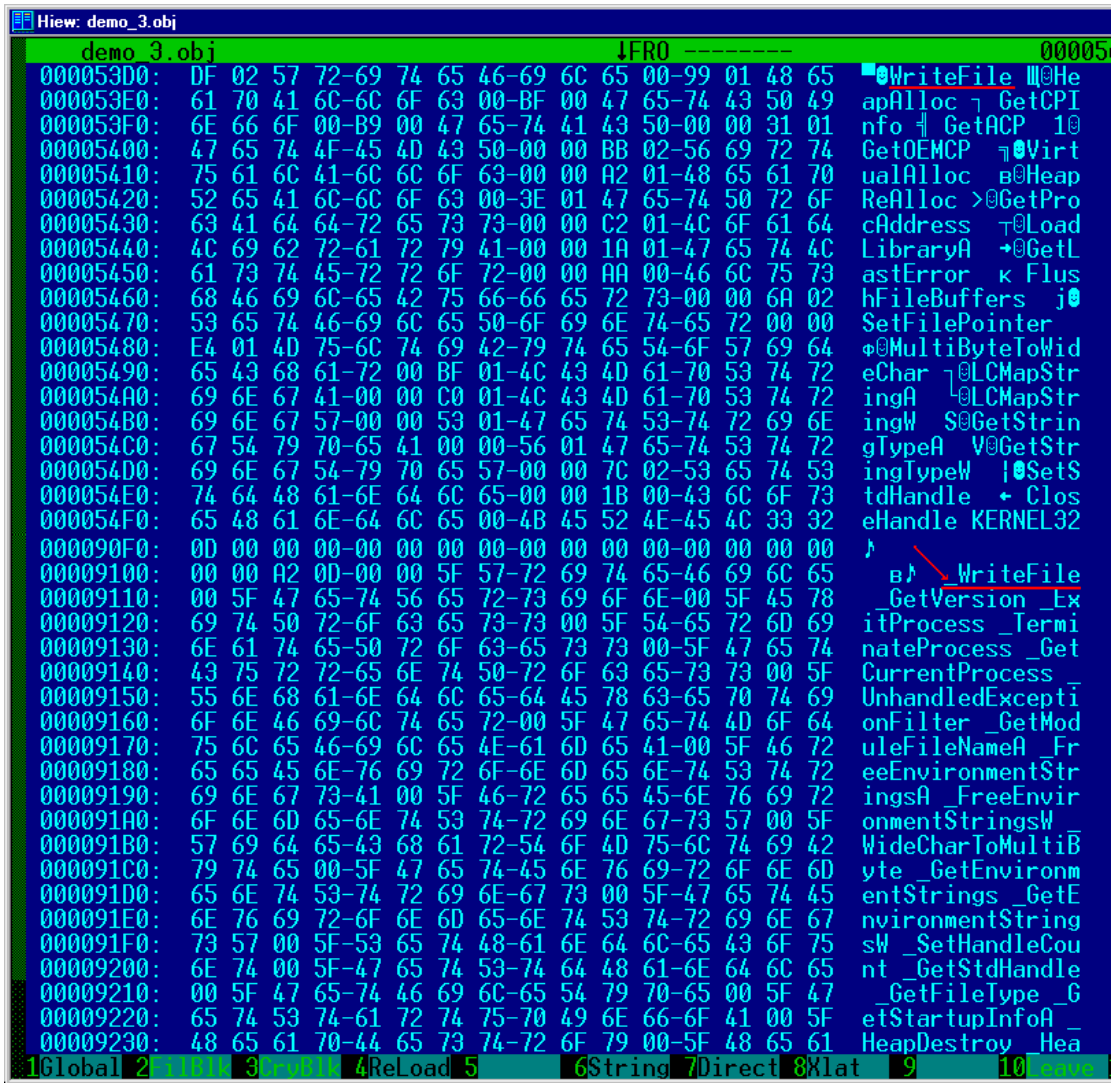

Рисунок 1 просмотр obj-файла в hex-редакторе

Строка "WriteFile" встречается дважды (см. рис. 1) — один раз со знаком прочерка, другой — без. Вот этот самый прочерк линкеру и не нравится. Откуда же он берется?! А оттуда! Курим листинг 4 и убеждаемся, насколько IDA Pro коварна и хитра. Тип API-функции (stdcall) задан только в комментарии! Транслятор же комментариев не читает, и берет тип по умолчанию, которым в данном случае является Си (cdecl), предписывающий перед всеми символьными именами ставить знак прочерка, что, собственно говоря, и происходит.

Кстати говоря, комментарий неправильный. Потому что тип вызова никак не stdcall, согласно которому, транслятор должен превратить "WriteFile" в "_WriteFile@20", где 20 — размер аргументов в байтах, заданный в десятеричной нотации. Это вообще не сама функция, а двойное слово, в которое операционная система заносит эффективный адрес WriteFile при загрузке PE-файла в память. В библиотеке KERNEL32.LIB (входящей, в частности, в состав SDK) ему соответствует имя "__imp__WriteFile@20". Именно такое "титул" должен носить прототип API-функции, если мы хотим успешно слинковать obj-файл и именно это имя мы используем при вызове API-функции при программировании на "голом" ассемблере (без включаемых файлов). Вот только IDA Pro во все эти подробности не вникает, перекладывая их на наши плечи и хвост.

Если во всем ассемблерном листинге заменить "WriteFile" на "_imp__WriteFile@20", то линкер переварит его вполне нормально и даже не отрыгнет. Нет, это не опечатка. Именно "_imp__WriteFile@20", а не "__imp__WriteFile@20". Почему?! Да потому, что второй символ прочерка транслятор добавит самостоятельно. Если же сразу указать два символа прочерка, то на выходе их образуется целых три, а это уже передоз.

Копируем "demo_3.asm" в "demo_3_test.asm", загружаем его в FAR по <F4>, давим <CTRL-F7> (replace) и заменяем "WriteFile" на "_imp__WriteFile@20", ассемблируем как и раньше, после чего повторяем попытку линковки с явным указанием имени библиотеки KERNEL32.LIB:

```
$link /SUBSYSTEM:CONSOLE demo_3_test.obj KERNEL32.LIB
Microsoft (R) Incremental Linker Version 5.12.8181
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

demo_3_test.obj:error LNK2001: unresolved external symbol _GetVersion
…
demo_3_test.obj:error LNK2001: unresolved external symbol _SetStdHandle
demo_3_test.obj:error LNK2001: unresolved external symbol _CloseHandle
demo_3_test.exe:fatal error LNK1120: 39 unresolved externals
```

Листинг 5 результат линковки после замены WriteFile на _imp_WriteFile@20

Оторвать мыщъх'у хвост! Это работает! Количество ошибок уменьшилось на единицу и первое неразрешенное имя теперь не _WriteFile, а _GetVersion! Переименовав оставшиеся API-функции, мы добьемся нормальной линковки программы, но… это же сколько труда предстоит! И в каждом новом ассемблерном файле, эту тупую работу придется повторять заново, вращая хвостом вплоть до полного его залегания. А ведь мы, мыщъх'и, не хотим, чтобы наш хвост залегал, верно?

Настоящие мыщъх'и (хакеры в смысле) идут другим путем — воспользовавшись директивами "**externdef**" и "**equ**" они создают для каждой API-функции свой алисас (alias), заставляющий транслятор трактовать функцию "func" как "__imp__func@XX". В частности, для WriteFile это будет выглядеть так:

```
externdef imp__WriteFile@20:PTR pr5
WriteFile equ <_imp__WriteFile@20>
```

Листинг 6 создание алиаса для API-функции WriteFile

Эту работу необязательно выполнять вручную и за вечер-другой можно написать утилиту, захватывающую DLL и выдающую готовый набор алиасов на выходе. Другой вариант — воспользоваться макросредствами FAR'а или редактора TSE-Pro (бывший QEDIT), позволяющих делать все что угодно и даже больше.

Самое главное, что "коллекцию" алиасов можно разместить в отдельном файле, подключаемом к ассемблерному листингу директивой "**include**". Создав все необходимые включаемые файлы один-единственный раз, мы можем пользоваться ими сколько угодно, причем не только для ассемблирования дизассемблерных листингов, полученных IDA Pro, но и в своих собственных ассемблерных программах.

Параметр "prN", идущий после "PTR", показывает сколько аргументов принимает функция и численно равен их размеру (число после символа "@") деленному на размер двойного слова, составляющему, как известно, 4 байта. То есть, в случае с WriteFile мы получаем: 20/4 = 5. Так же обратите внимание на символы прочерка. В первой строке "imp__func@XX" пишется вообще без знаков прочерка, во второй — с одним прочерком. Любые другие варианты не работают. Так что не надо косячить!

Рисунок 2 никакой хак не обходится без черной магии

колдование макроса

В нашем случае, создать включаемый файл для 40х API-функцией будет быстрее, чем писать и отлаживать полностью автоматизированную утилиту. С макросами на FAR'е вся работа не займет и пятнадцати минут. Главное — правильную стратегию иметь!!!

Перенаправив вывод линкера в файл "demo_3.err" (см. листинг 3), открываем его в редакторе по <F4>, подгоняем курсор к строке с первой ошибкой, затем по <CTRL-TAB> возвращаемся назад в панели, открывая по <F4> файл KERNEL32.LIB из SDK и тут же нажимаем <CTRL-L> для запрета редактирования (чтобы случайно его не испортить). Вновь возвращаемся в панели по <CTRL-TAB> и, нажав <SHIFT-F4>, создаем новый файл "demo_API.inc".

На этом подготовительные работы можно считать законченными и самое время приступать к созданию макроса. При всей свой внешней простоте в макросах заключена огромная сила, но пользоваться ею могут только маги (черные), мыщъх'и (серые, пещерные) и хомяки (всех пород). Я бы еще добавил к этому списку траву и грибы, да ведь только редакция,

стремающая наркоконтроля, ни за что это не пропустит, хотя… трава — она и под забором растет. Я же не конкретизировал! Короче, ситуация напоминает бородатый (как мыщъх) анекдот: идет еврей по послевоенной Москве и причитает: сколько бед и все от одного человека. Его вяжут парни из ГБ и начинают пытать: скажите, а какого человека вы имели ввиду? Еврей — Гитлера, конечно! Гэбисты — Хм, ну тогда идите. Еврей — простите, а вы кого имели ввиду?

Рисунок 3 мыщъх серый, небритый, лохматый, пещерный. совершенно запущенный тип

Короче, как ни крути, а без травы не обойтись, потому что, совершить следующий ритуал можно только с похмелья или находясь в состоянии расширенного сознания. Но он работает! И это главное! Значит так, находясь в "demo_API.inc", нажимаем <CTRL-.>, переводя FAR в режим записи макроса (при этом в левом верхнем углу злобно загорается красная буквица R, что означает Record). Погружаемся в состояние медитации и…

1. вызываем меню "Screen" по <F12>, в котором окна перечислены в порядке их открытия;
2. нажимаем <1> для перехода в "demo_3.err", которое мы открыли первым;
3. нажимаем <END> для перехода в конец строки;
4. нажимаем <CTRL-LEFT> для перемещения курсора в начало имени функции;
5. нажимаем <LEFT> для перехода, через символ прочерка;
6. нажимаем <SHIFT-END> для выделения имени API-функции;
7. нажимаем <CTRL-INS> для копирования его в буфер обмена;
8. нажимаем <HOME>, <DOWN> для перехода к следующей строке;
9. нажимаем <F12> для вызова меню "Screen" и давим <2> для открытия KERNEL32.LIB;
10. нажимаем <F7> (search) и вставляем имя функции по <SHIF-INS>, затем <ENTER>;
11. нажимаем <SHIFT-CTRL-RIGHT> для выделения имени функции со знаком "@XX";
12. копируем его в буфер обмена по <CTRL-INS>;
13. нажимаем <HOME>, чтобы следующий поиск начинался сначала файла;
14. нажимаем <F12> и по нажатию <3> переходим в demo_API.inc;
15. пишем "externdef imp__" и нажимаем <SHIFT-INS> для вставки имени из буфера;
16. дописываем к нему ":PTR pr0" и нажимаем <ENTER> для перехода к след. строке;
17. нажимаем <SHIFT-INS> еще раз, вставляя имя типа "WriteFile@20";
18. нажимаем "пробел" и вставляем имя еще раз;
19. нажимаем <HOME> для перехода в начало строки;
20. нажимаем <F7> и затем "@", <ENTER> для поиска символа "@";
21. нажимаем <SHIFT-CTRL-LEFT> для выделения "@NN";
22. нажимаем <SHIFT-DEL> для удаления "@NN" в буфер обмена;

23. пишем " equ <_imp__" (с ведущим пробелом в начале);
24. нажимаем <DEL> для удаления символа проблема под курсором;
25. нажимаем <END> для перехода в конец строки;
26. пишем ">";
27. нажимаем <ENTER> для перехода на следующую строку;

Все! Создание макроса завершено! Нажимаем <CTRL-.> и вешаем макрос на любую незанятую комбинацию горячих клавиш (например, на <CTRL-~>), после чего нам остается только уронить кирпич на <CTRL-~>, созерцая как трудолюбивый макрос выполняет всю рутинную работу за нас. Ну… или почти всю. Количество аргументов в параметре "pr0" необходимо вычислить самостоятельно, но это уже мелочи, почти не отнимающие времени.

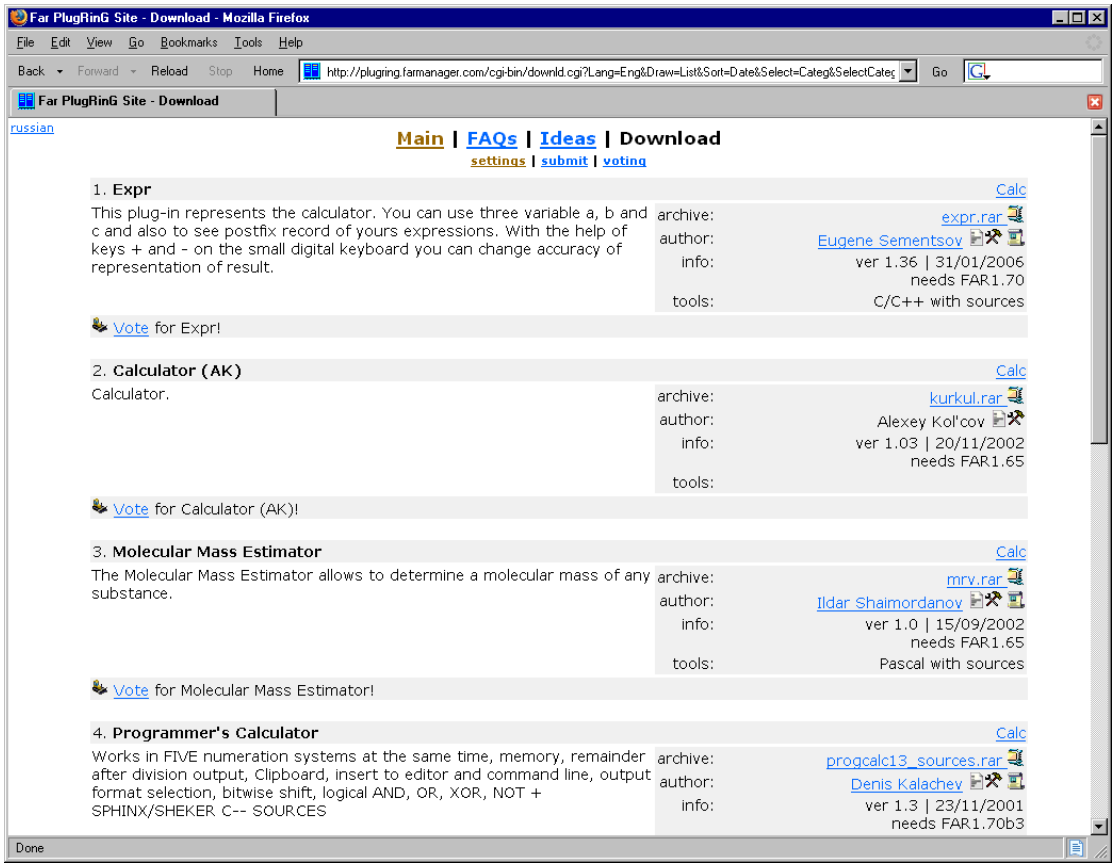

Рисунок 4 плагины калькуляторов к FAR'у

Тем не менее, при желании можно сотворить полностью автоматизированный макрос. Для этого нам потребуется скачать с http://plugring.farmanager.com/index_e.html один из многих валяющихся там калькуляторов (см. рис. 4), после чего, дойдя до шага 23, слегка изменить свою тактическую стратегию, представленную ниже (чтобы не перебивать макрос заново, имеем смысл обзавестись редактором макросов, так же представляющим собой плагин):

1. вызываем калькулятор, используя свойственный ему метод вызова;
2. нажимаем <SHIFT-INS> вставляя "@NN" из буфера обмена;
3. нажимаем <HOME> для перехода в начало строки;
4. нажимаем <DEL> для удаления символа "@";
5. нажимаем <END> для перехода в конец строки;
6. пишем "/4" и нажимаем <ENTER> для расчета значения;
7. копируем вычисленное значение в буфер обмена;
8. *** продолжаем выполнение прежней макропоследовательности до шага 27 ***
9. нажимаем <UP> для перехода на строку вверх;
10. нажимаем <END> для перехода в конец строки (на "pr0");
11. нажимаем <BASKSPACE> для удаления "0" и вставляет результат вычислений;
12. нажимаем <DOWN>, <END> для перехода в конец следующей строки;

13. *** продолжаем выполнение прежней макропоследовательности с шага 27 ***

В результате у нас должен образоваться включаемый файл следующего вида (смотрите, сколько времени у нас заняло составление макроса и сколько бы отняла разработка программы на любом другом языке программирования!):

```
externdef imp__WriteFile@20:PTR pr5
WriteFile equ <_imp__WriteFile@20>

externdef imp__GetVersion@0:PTR pr0
GetVersion equ <_imp__GetVersion@0>

externdef imp__ExitProcess@4:PTR pr1
ExitProcess equ <_imp__ExitProcess@4>
```

Листинг 7 фрагмент включаемого файла demo_API.inc

непонятки с ___imp_RtlUnwind

Пока мыщъх'и оттягивались на конопляном поле в полный рос, находясь в блаженном кайфе, магический макрос споткнулся на функции "___imp_RtlUnwind" (он попросту не нашел ее в KERNEL32.LIB), все пошло кувырком и мыщъх'ей сразу попустило.

Что же это за противная функция такая?! Кстати, в KERNEL32.LIB ее действительно нет. Так что макрос тут не причем. Смотрим в demo_3.asm (не забывая убрать один из символов прочерка, вставленный транслятором). Контекстный поиск тут же находит RtlUnwind, представляющую собой классический переходник (thunk) к одноименной функции из KERNEL32.DLL:

```
RtlUnwind       proc near
                jmp     ds:__imp_RtlUnwind
RtlUnwind       endp
```

Листинг 8 переходник (thunk) к API-функции RtlUnwind из KERNEL32.DLL

Только вот по не совсем понятной причине, IDA Pro не нашла этой функции в KERNEL32.DLL (не знала о ней? или не захотела искать?!), но тот же HIEW отобразил thunk совершенно правильно!

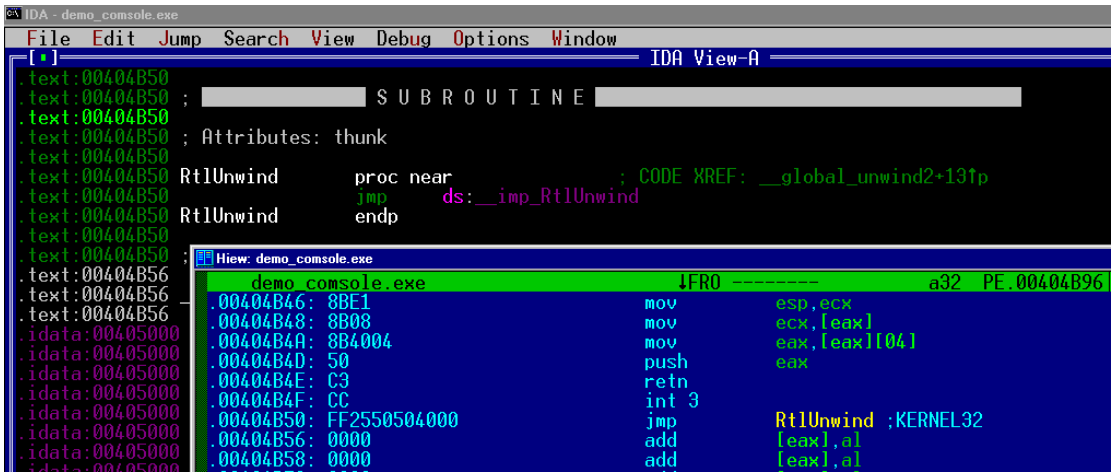

Рисунок 5 HIEW правильно отображает имя API-функции RtlUnwind, а IDA Pro – нет

Собственно, багофича заключается в том, что IDA Pro дает API-функции неправильное имя. Ну какой же это "__imp_RtlUnwind"?! Правильный вариант включает в себя _два_ символа прочерка между "imp" и "RtlUnwind". Естественно, наш магический (но слегка туповатый) макрос не ожидал такой внезапной подлости! Приходится брать бразды правления на себя и либо править ассемблерный листинг, добавляя еще один символ прочерка, либо алиасить функцию как есть. Последний вариант более предпочтителен, поскольку он не требует вмешательства в исходный код. Включаемый файл нужно писать, так, чтобы он работал, а не исходить из того, что правильно/неправильно и не пытаться оправдаться "это же не наш баг?

почему мы должны его учитывать?!". Положитесь на мой хвост парни! Мы должны! И правильный алисасинг выглядит так:

```
externdef imp__RtlUnwind@16:PTR pr4
__imp_RtlUnwind equ < _imp__RtlUnwind@16>
```

Листинг 9 алиасинг API-функции __imp_RtlUnwind

линковка

Копируем файл demo_3.asm в demo_4.asm, добавляем в его начало директиву "include .\demo_api.inc" (см. рис. X), подключающую включаемый файл, и повторяем весь цикл трансляции вновь. Ассемблируем: "ML.EXE /coff /I. /c /Cp /Zp1 /Zm demo_4.asm". Убеждаемся в отсутствии ошибок и линкуем: "link /SUBSYSTEM:CONSOLE demo_3_test.obj KERNEL32.LIB".

О чудо! Линкер совсем без матюгов и почти без перекуров создает demo_4.exe, приближая нас к конечной цели еще на один шаг!

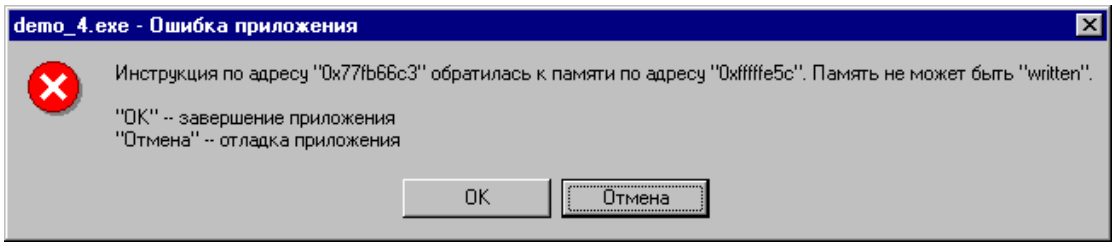

Рисунок 6 результат запуска demo_4.exe

запуск файла или отложенное заключение

Полученный таким трудом exe-файл при попытке его запуска выдает сообщение о критической ошибке и отваливает. Значит, еще не все косяки исправлены и IDA Pro где-то конкретно напортачила с дизассемблированием. Чтобы узнать где — необходимо взять в лапы отладчик и проанализировать причины падения программы, устраняя ошибки одну за другой (а их там — целый легион!). Но этим мы займемся в следующий раз, а пока покурим и попутно подведем итоги.

Мы проделали объемную работу и сделали больше дело: разобрались с алиасами, научились готовить включаемые файлы, позволяющие транслировать дизассемблерные листинги без их доработки напильником, и — самое главное — испытали на себе действие черной магии макросов FAR'a, а эти чувства не забываются!

>>> врезка "не ассемблируется aam 16"

После выхода первой статьи этого цикла, на хакерском форуме читатель с ником realstudent задал предметный вопрос из которого выяснилось, что во-первых, у него не ассемблируется инструкция "aam 16" (но это мелочи, сейчас докурим и ее обломаем), и, во-вторых, "секреты ассемблирования дизассемблерных листингов" (в девичестве) превратились в "сношение с идой". Вот такая трава растет на широте Москвы!

А само сообщение (и ответ на него) выглядели так:

> есть древнее приложение - программер микрухи через LPT
> (очень напрягает он своей работой) но без сырцов,
> и твоя статья пришлась очень в тему. а тема такая:
> решил восстановить его и дописать если возможно.
> пользовался IDA 5.x (лицензионная, ясное дело!) и
> MASM 9.0 (тоже лицензионный, с митино).
> все ошибки убил кроме одной и в чем ее смысл
> никак не могу понять (в асме я нормально разбираюсь,
> смотрел другие исходникик на koders.com - все у людей также,
> был на microsoft.com - но так и не понял к чему эта ошибка здесь).
> не ассемблируется строка "aam 16"
> - error A2008: syntax error : integer (блин это последняя ошибка!)

> по мануалам от Intel'a команда поддерживает аргумент
> (в смысле команда правильная), а вверху листинга у меня торчит:
>..686p
> ;.mmx
> ..model large, C
без измен, мужик! только без измен! это она по спецификациям ассемблируется, но только разработчики ассемблера спецификации читают по диагонали, и у них на этот счет имеется свое, особое мнение, которое умом не понять, а в свете того, что Microsoft озаботилась разработкой собственного процессора, сдается мне, что x86 коллектив разработчиков не осилил, что совсем не удивительно, если вернутся на десяток лет назад и вспомнить, что Microsoft не могла разобраться в разработанных ею же спецификациях на расширенную память и драйвера забивали косяки только так.
но это была лирика. что же касается сути проблемы, то она обходится методом "ну и не ассемблируется... ну и хрен с ней...". вставляем директиву DB и записываем инструкцию непосредственно в машинных кодах.
в данном случае это выглядит так: "DB 0DBh, 10h", где DBh – опкод команды AAM, а 10h — непосредственный операнд. та же самая история наблюдается и с командой AAD (да и не только ей), опкод которой D5h и в машинной форме она вызывается так: "DB D5h, XXh".