ассемблирование без секретов

крис касперски аргентинский болотный бобер nezumi el raton ака жирный нутряк ибн мыщъх

решил хакер блинов напечь. первый блин у него вышел, как водится, комом. зато - экзешником.

народное

в сети лежит множество ассемблерных листингов, но большинство из них находится в сильно разобранном состоянии и… не транслируется! как "причесать" листинг, внедрить его в свою программу, выбрать правильный транслятор и ключи командной строки — поведает мыщъх в этой статье.

введение или много лет тому назад

Свою программистскую карьеру мыщъх начинал с микрокомпьютера "Правец-8D", оснащенного довольно экзотической версией Бейсика и нехилым руководством с кучей конкретных примеров (правда, на болгарском языке). Процесс освоения буржуинской техники происходил приблизительно так. Набрал программу. Запустил. Помедитировал над листингом. Попробовал что-нибудь изменить. Запустил. Посмотрел на реакцию. Осмыслил. Что-то еще изменил. И вот так, шаг за шагом мыщъх разобрался во всех операторах языка и научился писать эффективные программы, в которых нет ничего лишнего.

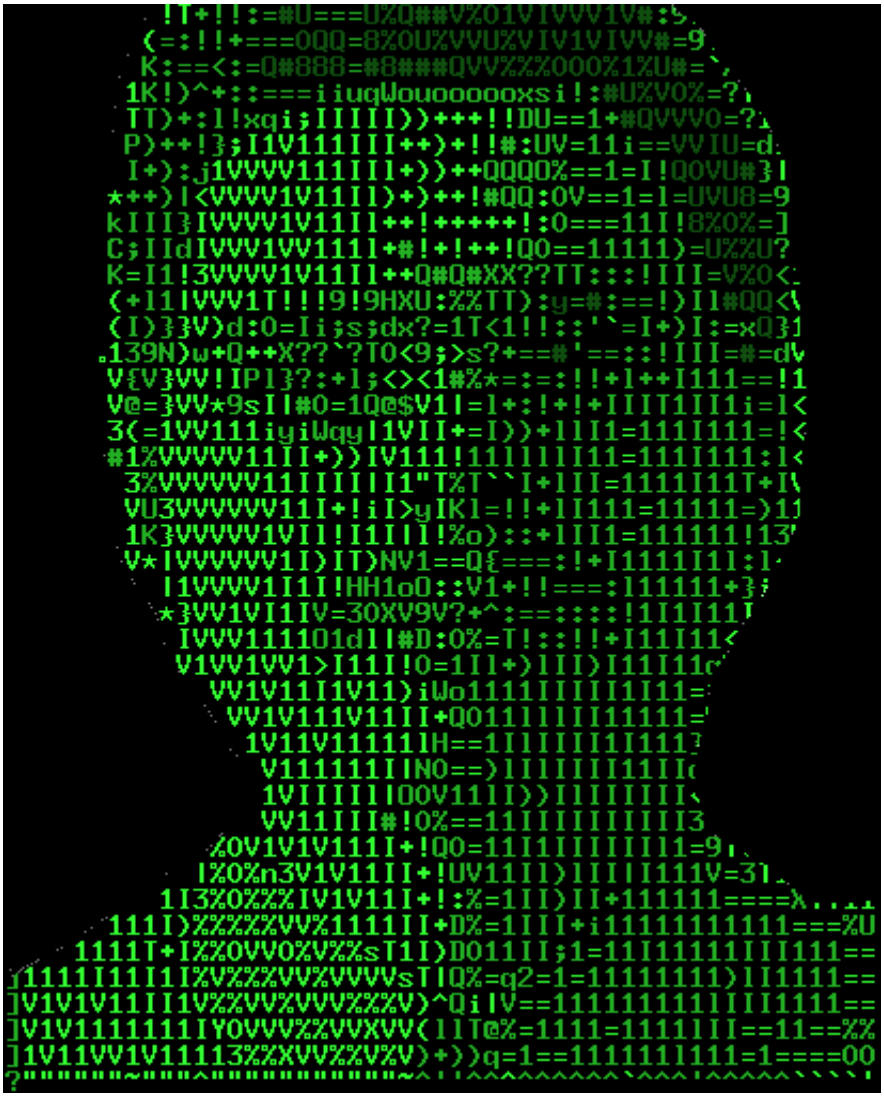

Рисунок 1 лицо асматика в ASCII

Ассемблер на этот трюк не поддался — исходные тексты программ, нарытые в различных журналах и книгах, не транслировались, а те, что транслировались, — не работали. Много позже я узнал, что то были не программы, а всего лишь их фрагменты. Отсутствие законченного "фундамента", на который было бы можно опереться, отбрасывало мыщъх'а далеко назад и к ассемблеру он вернулся только с приобретением компьютеров Электроника БК (ака PDP-11) и Агат (ака Apple][). Большое количество разнообразных программ существенно упрощало их изучение, а знакомство с монитором повергло мыщъха в настоящий культурный шок — ничего подобного на Правеце не было. (К слову сказать, "монитор" — это не "телевизор", это программа такая — прообраз отладчика).

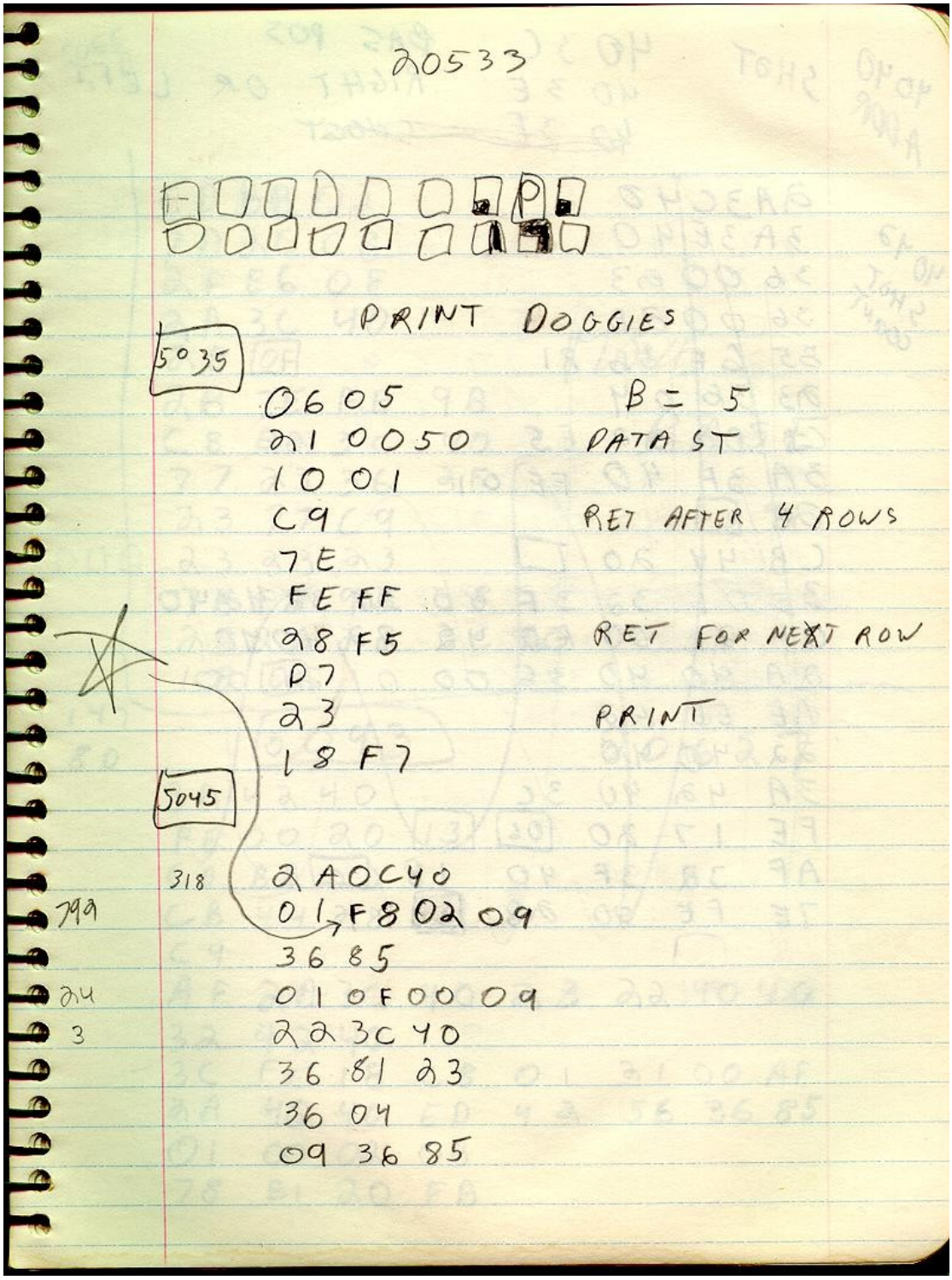

Рисунок 2 памятный блокнот с записями начинающего асматика

Переход на IBM PC оказался довольно болезненным, а x86-ассемблер не похожим ни на что изученное до него. Хоть общие параллели и имелись, освоить его тихим сапом у мыщъх'а не получилась. Пришлось брать штурмом. В debug.com. Дизассемблирование (командой u) чужих программ давало мнемоники инструкций, которые вводились в ассемблерном режиме (команда a), а затем изучалось воздействие на флаги, регистры и память. Этот увлекательный процесс растянулся на многие месяцы, но даже когда мыщъх вполне уверенно сочинял ассемблерные программы в тысячу строк, трансляция чужих ассемблерных программы по-прежнему представляла проблему, которую мыщъх испытал на своей шкуре и хвосте, поэтому будет писать как для себя самого, то есть предельно доступно и без лишнего выпендрежа.

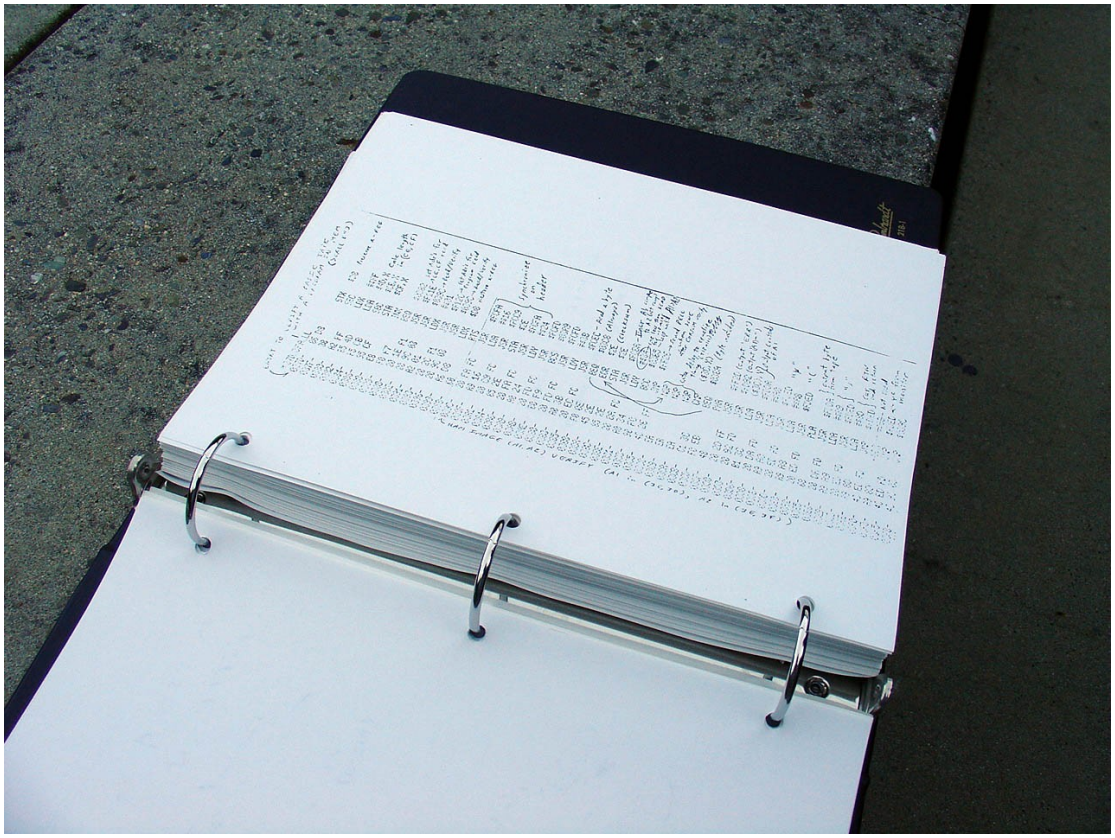

Рисунок 3 рабочий блокнот хронического асматика

зоопарк ассемблеров

Ассемблер — это не только язык, но еще и транслятор. То, что "язык" PDP-11 не пригоден для x86 — это понятно, но вот несовместимость ассемблерных трансляторов друг с другом — для многих становится новостью. Что составляет фундамент ассемблера как языка? Мнемоники машинных команд (mov, nop, cmp) – это раз и средства самого языка (метки, директивы, макросы) — это два! Формально, за мнемоники отвечают Intel и AMD. Именно они дают символические имена машинным командам, регистрам, флагам и иже. Большинство x86-ассемблеров придерживаются этой нотации (хоть она никем и не стандартизирована), однако, "большинство" это еще не все. В мире UNIX широко распространен AT&T синтаксис, отличающийся не только синтаксисом, но и порядком операндов! Здесь операнд-приемник расположен не ***слева***, как у Intel, а ***справа***!!! Регистры начинаются со знака процента, константы — со знака доллара, а инструкции имеют суффикс, соответствующий типу обрабатываемых данных. На языке Intel пересылка в регистр eax значения 666h выглядит так: "`mov eax,666h`", а на AT&T так: "`movl $666h,%eax`". Как говориться — почувствуете задницу! (И всю плачевность ситуации тоже).

Программа, предназначенная для одного типа ассемблеров, не может быть откомпилирована на другом без радикальной переделки или автоматической конвертации! Но даже среди ассемблеров "своего" типа наблюдается разброд, разнобой и множество различий: в

ключевых словах, в правилах оформления листинга, в поставляемых библиотеках и заголовочных файлов и т. д. Если только совместимость не заявлена явно, транслировать программу нужно тем и только тем ассемблером для которого она предназначена. В противном случае — готовьтесь к переделкам (то есть, к адоптации). Отличия зачастую проявляются в самых неожиданных местах. Некоторые ассемблеры понимают, что "`mov eax, x`" это тоже самое, что и "`mov eax,[x]`", некоторые — нет. Они спотыкаются и выдают ошибку. Но еще ничего! Гораздо хуже, когда транслятор молчаливо трактует эту конструкцию как "`mov eax, offset x`", что совсем не одно и тоже! Так что при переносе программы приходится быть очень и очень осторожным.

Совместимость операционных систем — вообще песня. Программы, ориентированные на MS-DOS, без мата не только не транспортабельны, но и непереносимы. Для них характерно прямое взаимодействие с оборудованием, доступное в NT только с ядерного уровня, не говоря уже о том, что 16-разрядный код вызывается из 32-разрядных приложений только через DPMI, да и то не без ухищрений.

Таким образом, ***прежде чем транслировать ассемблерную программу, необходимо отождествить для какого транслятора и операционной системы она предназначена!*** С ассемблерными фрагментами, выхваченными из "родного" контекста, приходится еще хуже. Допустим, в некоторой статье описывается интересный антиотладочный прием и приводится ассемблерный код, но как встроить его в свою программу — не говорится. Знакомая ситуация, не правда ли? Непосредственная трансляция невозможна — транслятор дико материться, но ничего не говорит.

Вот обо всем этом мы и будем говорить, а пока пойдем на балкон и покурим!

Рисунок 4 процесс ассемблерной трансляции в UNIX

определение целевой платформы

Проще всего определить разрядность — если в листинге преобладают 16-разрядные регистры типа AX/BX/CX, то скорее всего она предназначается для MS-DOS. Если встречаются прямые вызовы прерываний INT 21h, INT 13h, INT 16h, INT 10h — это точно MS-DOS от попыток трансляции программы под NT лучше сразу воздержаться. Тоже самое относится и портам ввода/вывода (инструкции IN/OUT). И хотя NT позволяет открывать к ним доступ и с прикладного уровня, это не выход и такую программу проще взять переписать.

32-режим характеризуется регистрами EAX/EBX/ECX. Это может быть как программа для Windows, так и для DOS/DPMI. Windows распознается по своим API-функциям, DOS/DPMI – по прерыванию INT 31h. Прерывание INT 2Fh – свидетельствует о принадлежности к 9x, INT 2Fh/SYSENETR — NT/XP. Через эти прерывания осуществляется доступ к низкоуровневому API операционной системы, что делает такие программы непереносимыми. Привилегированные инструкции защищенного режима или вызовы функций, экспортируемых ядром (например, IoGetCurrentIrpStackLocation) указывают на драйвер (или его

фрагмент), совершенно не приспособленный к работе на прикладном режиме. Если листинг не вызывает никаких API-функций, не дергает прерываниями, не содержит привилегированных инструкций и не обращается к памяти по абсолютным адресам (типа mov eax,fs:[20h]), то он может работать в любой 32-разрядной операционной системе.

64-режим x86-64 распознается по регистрам RAX/RDX/RCX и с 32-разрядными трансляторами, естественно, не совместим.

UNIX распознается своим характерным AT&T синтаксисом. Программы, опирающиеся на библиотеку libc (а таких — большинство) легко переносятся в Windows, поскольку libc – это стандартная Си-библиотека, однако, некоторые UNIX-функции в ее Windows-версии не реализованы. В частности, отсутствует вызов fork, расщепляющий процесс на два и в комбинации fork/exec делающий тоже самое, что и CreateProcess. То есть, в ряде случаев перенос все-таки возможен! Особенно это относится к математическим функциями, абстрагированным от системного мира. Программы, работающие в обход libc через интерфейс INT 80h/call 0007h:00000000h (ими обычно являются вирусы и черви) практически непереносимы.

Определить транслятор несколько сложнее. Признаком TASM'а являются директивы "jumps", "locals" в начале файла. FASM обычно определяется по директиве "format", например, "format PE GUI 4.0", (что, впрочем, не совсем надежно) и хроническому отсутствую ключевого слова offset. На долю MASM'а приходится все остальное.

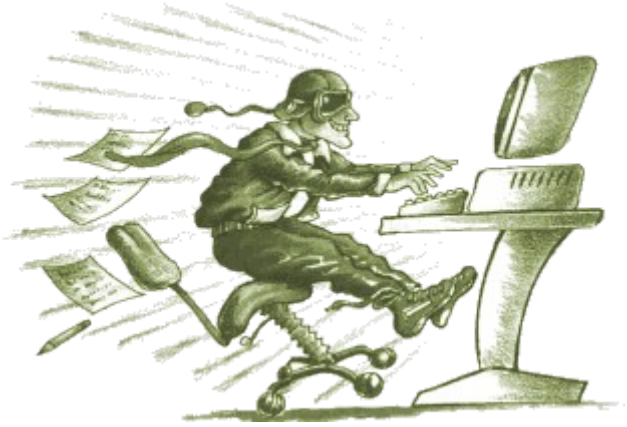

Рисунок 5 ассемблер — полный улет

метод ассемблерных вставок

В качестве "боевого" примера рассмотрим классический антиотладочный код, встречающийся во многих статьях и книгах:

```
        push offset my_she     ; назначаем свой обработчик структурных исключений
        push dword ptr fs:[0]  ; сохраняем старый обработчик в цепочке
        mov fs:[0],esp         ; регистрируем новый обработчик

        pushf                  ; толкаем в стек флаги
        or dword ptr[esp],100h ; взводим трассировочный бит
        popf                   ; выталкиваем обновленный бит в регистр флагов,
                               ; заставляя ЦП возбуждать исключение на каждой команде

        xor eax,eax            ; без отладчика после xor возбуждается исключение и
                               ; управление получает my_seh, а в eax будет не нуль
                               ; под отладчиком исключение молчаливо "съедается"
my_seh:
        test    eax,eax        ; если отладчика нет, eax != 0
        jnz     debugger_is_present
```

Листинг 1 классический пример антиотладочного кода, основанного на поглощении трассировочного прерывания отладчиком

Как нам откомпилировать? И главное во что? На самостоятельную программу оно как-то не тянет… А давайте попробуем заточить ассемблерную вставку? В большинстве Си компиляторов она оформляется как: "__asm{…ассемблерный код…}", однако,

непосредственно осуществить этот замысел не получится! Ведь наш "подопечный" ассемблерный листинг, как и большинство других демонстрационных прогарам, содранных с наглядно-агитационных пособий, представляет собой нечто промежуточное между псевдокодом и рабочей программой.

Во-первых, метка debugger_is_present не определена и вставлена чисто для наглядности, во-вторых, установить обработчик структурных исключений мы установили, флаг трассировки — взвели, а вот о том, что после завершения проверки на отладчик все необходимо вернуть обратно — забыли! Поэтому, перед употреблением, листинг необходимо слегка доработать, например, так:

```
main()
{
        int a;                          // через эту переменную мы возвратим значение eax

__asm{                                  // ассемблерная вставка — начало
        push offset my_seh
        push dword ptr fs:[0]
        mov fs:[0],esp

        pushf
        or dword ptr[esp],100h ;set trap flag
        popf

        xor eax,eax
my_seh:
        pop dword ptr fs:[0]            ; восстанавливаем старый обработчик
        add esp,4                       ; восстанавливаем стек

        mov a,eax                       ; возвращаем результат в переменной a
        }                               ; ассемблерная вставка - конец

        // проверка переменной a на равенство нулю
        printf("%s\n",a?"no debugger":"under debugger");
}
```

Листинг 2 законченная программа anti-debug.c

Не самый лучший вариант конечно. При входе в обработчик структурных исключений мы должны выйти из него через недокументированную API-функцию Continue, хотя… будет работать и так. Поэтому, не будем отвлекаться на мелочи технической реализации, а сосредоточимся на оформлении ассемблерной вставки.

```
$cl anti-debug.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

anti-debug.c
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:anti-debug.exe
anti-debug.obj

$anti-debug.exe
no debugger
```

Рисунок 6 anti-debug.exe без отладчика

Мы удаляем "jnz debugger_is_present", а вместо этого возвращаем значение через предварительно объявленную переменную "a". Компилируем программу как обычно ("`cl.exe anti-debug.c`") и пытаем, то есть делаем попытку запуститься. При прогоне под soft-ice, ollydbg или любым другим не эмулирующим отладчиком на экране покажется, "under debugger" и "no debugger" в противном случае. Значит, наша программа работает правильно и трансляция удалась! Отрываем мыщъху хвост на радость, тем более что с крышей у него проблемы (ну, с головой). Прохудилась и течет зараза, а времени на ремонт нет. Так что мыщъх пишет эти строки сидя на ноотропах в состоянии измененного сознания и не вполне вменяем, а потому временами не совсем адекватен. Ладно, это мои личные проблемы, так что не высаживайтесь.

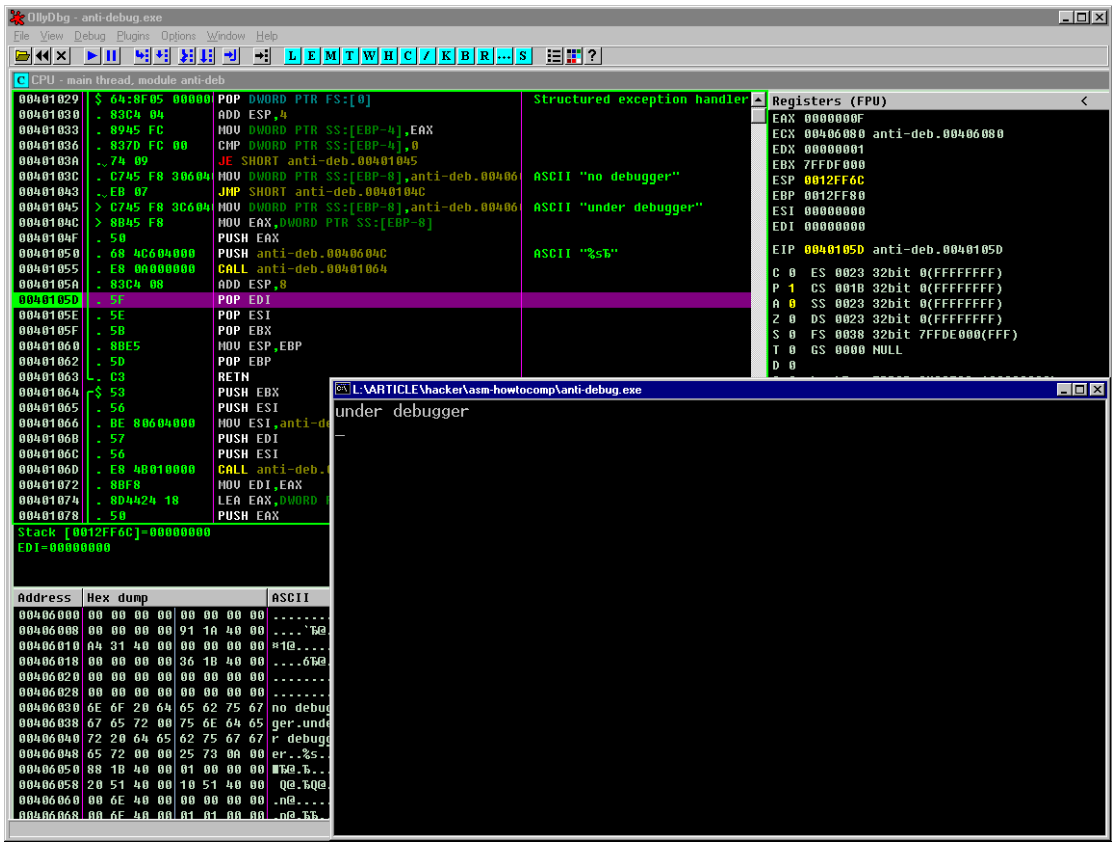

Рисунок 7 anti-debug.exe под отладчиком

А вот другой классический пример:

```
.code                          ; секция кода
start:                         ; точка входа
        push 0                 ; uType
        push 0                 ; lpCamtion
        push offset s0         ; lpText
        push 0                 ; hWnd
        call MessageBoxA       ; зовем функцию
        ret                    ; после нажатия на ок выходим отсюда на хер

.data                          ; секция данных

s0 db "hello,wordl",0Dh,0Ah,0 ; строка, которую мы будем выводить

end start                      ;
```

Листинг 3 ассемблерный фрагмент, приветствующий мир через экран, гуевый мир, в котором все вызывает отвращение к чему ни прикоснись

Попытка загнать текст программы в ассемблерную вставку ни к чему хорошему не приводит. Компилятор кроет нас матом, но не компилит. Редиска! Приходится действовать стратегически. То есть свирепо и радикально. Убираем директивы .code и .data вместе с ненужной инструкций "ret" (в оригинале она завершает программу, пользуясь тем фактом, что при запуске PE-файла на вершине стека лежит адрес на термирующую процедуру, однако, в стековой фрейме нашей ассемблерной вставки ничего подобного нет!).

Метку start можно, в принципе, и не убирать, но на фиг она будет торчать?! А вот к метке s0 подобный диалектический подход уже не приемлем и с ней надо кончать. В смысле — избавляться от этой твари любой ценой. Ну не поддерживает встроенный ассемблер директивы "db", что тут поделать?! Приходится объявлять строковую константу средствами самого Си. Она может быть размещена как в стеке (т. е. объявлена как локальная переменная), так и в секции данных (т. е. объявлена как глобальная переменная). В действительности, строки всегда размещаются в секции данных, а стек заносится лишь их копия, а копия — это оверхид, то есть накладные расходы и прочий маст-дай (***примечание**: некоторые оптимизирующие*

компиляторы, в том числе и ms vc, могут "загонять" строковые переменные в стек при помощи инструкций PUSH XXYYZZ, и тогда в секции данных их уже не оказывается).

Если переменная объявлена как глобальная, то ключевое слово offset сохраняет свою силу и компилятор не матерится. С локальными переменными все сложнее. Конструкция "`push offset s0`" в этом случае разворачивается компилятором в "`push offset [ebp+x]`", что с точки зрения синтаксиса совершенно бессмысленно! Но убирать "offset" нельзя, поскольку "`push [ebp+x]`" заталкивает в стек отнюдь не указатель на s0, а… значения первых четырех байт, то есть ведет себя как "`*((DWORD*)s0)`". Правильный вариант выглядит так: "`lea eax,s0/push eax`" (разумеется, вместо eax можно использовтаь любой другой регистр общего назначения).

Еще один нюанс — конструкция "`call MessageBoxA`" выполняется совсем не так, как задумывалось, поскольку вместо MessageBoxA коварный компилятор подставляет отнюдь не адрес самой MessageBoxA, а указатель на двойное слово, хранящее адрес MessageBoxA! Следовательно, чтобы программа не развалилась и не умерла, необходимо использовать префикс `ds` и тогда вызывающий код будет выглядеть так: "`call ds:MessageBoxA`"

Обобщив все вышесказанное, мы получаем следующую программу. Даже две! С объявлением строки как глобальной и локальной переменной:

```
#include <windows.h>

// глобальная переменная выводимая на экран
char s0[]="hello,wordl\n";

main()
{
__asm
{
        push 0
        push 0
        push offset s0
        push 0
        call ds:MessageBoxA    ; добавляем ds:
        }
}
```

Листинг 4 программа hello_global.c, с "глобализацией" ассемблерных перемнных

Компилируем ("`cl.exe hello_global.c USER32.lib`"), где USER32.lib – имя библиотеки для MessageBoxA, и запускам на выполнение. Получаем симпатичное диалоговое окно.

Рисунок 8 окно, созданное ассемблерной вставкой

Вариант с локальной переменной компилируется и запускается точно так же, как и предыдущий:

```
#include <windows.h>
main()
{
char s0[]="hello,wordl\n";
__asm
{
        push 0
        push 0
        lea eax, s0
        push eax
        push 0
        call ds:MessageBoxA
        }
}
```

Листинг 5 программа hello_local.c с "локализацией" ассемблерных переменных

Ассемблерные вставки, бесспорно, удобная вещь, но все-таки не свободная от ограничений. В частности, встроенный ассемблер не поддерживает никаких макросредств и если в транслируемой программе присутствует множество макросов, без помощи MASM'a (или его конкурентов) здесь уже не обойтись.

MASM, TASM и FASM

Будем считать, что мы достаточно созрели для ассемблирования всей программы целиком. Казалось бы, чего же тут сложного? Бери и транслируй. Ан нет! Вот еще один классический пример, выловленный на просторах Интернета и по замыслу своего создателя выводящий "hello,world":

```
.386
.model flat

extern ExitProcess:PROC
extern MessageBoxA:PROC

.data
s0 db 'hello, world',0

.code
start:
        push 0
        push 0
        push offset s0
        push 0
        call MessageBoxA

        push 0
        call ExitProcess
end start
```

Листинг 6 пример простейшей программы hello.c которую мы собираемся ассемблировать всю целиком

Транслируем программу MASM'ом, последнюю версию которого можно позаимствовать из NTDDK: "`ml /c /coff hello.asm`", где "/c" – ключ, означающий "только ассемблировать, не линковать" (ликованием мы займемся самостоятельно, только позже), "/coff" – транслировать в coff-файл (по умолчанию создается omf с которым мало кто из линкеров умеет работать). Ну а "hello.asm" – имя нашего файла. MASM ругается: "warning A4022: with /coff switch, leading underscore required for start address: start", но вроде бы ассемблирует. Постойте! Но ведь у нас уже есть метка start, заданная в качестве стартового адреса! Что же транслятору еще надо?! Ебанный Microsoft! MASM хочет иметь "_start" (с подчеркиваем), а у нас подчеркивания и нету! Выход: заменить start на _start или в модели паияти указать тип вызовов "stdcall".

Теперь программа ассемблируется без проблем и наступает черед ее линковать. Это делается так: "`link /SUBSYSTEM:WINDOWS hello.obj KERNEL32.LIB USER32.lib`", где "SUBSYSTEM" – ключ, отвечающий за выбор подсистемы (в данном случае WINDOWS, еще есть CONSOLE для консольных программ и NATIVE – для драйверов), "hello.obj" – имя линкуемого файла, KERNEL32.LIB и USER32.LIB — имена необходимых библиотек, поставляемых вместе с Platform SDK. Если же SDK нет, линкер ms link может сгенерировать их самостоятельно, стоит только указать ему ключ "`/IMPLIB:KERNEL32.DLL`". Откуда мы знаем, какие библиотеки нужно подключать? Ответ дают вызываемые API-функции: ExitProcrss, экспортируемая KERNEL32.DLL и MessageBoxA, экспортируемая USER32.DLL, о чем написано в SDK. Если же SDK нет — смотрите экспорты всех системных библиотек любой подходящей утилитой типа dumpbin.

Только все равно ни фига у нас не линкуется!

```
hello2.obj : error LNK2001: unresolved external symbol _ExitProcess
hello2.obj : error LNK2001: unresolved external symbol _MessageBoxA
hello2.exe : fatal error LNK1120: 2 unresolved externals
```

Листинг 7 линковка не удалась

Вот так номер! Линкер не может найти функции! Почему это так? Заглянув в USER32.lib hex-редактором, мы увидим, что MessageBoxA там объявлена как "`_MessageBoxA@16`", где "_" признак stdcall-вызова, а "@16" – размер всех аргументов функции в байтах. Соответственно, ExitProress зовется как "`_ExitProcess@4`", поскольку принимает всего один аргумент, а в 32-разрядном режиме все они двухсловные.

```
00004940:  40 31 32 00-5F 45 6E 75-6D 50 72 6F-70 73 45 78  @12 _EnumPropsEx
00004950:  57 40 31 32-00 5F 5F 69-6D 70 5F 5F-45 6E 75 6D  W@12 __imp__Enum
00004960:  50 72 6F 70-73 45 78 57-40 31 32 00-5F 4D 65 73  PropsExW@12 _Mes
00004970:  73 61 67 65-42 6F 78 41-40 31 36 00-5F 5F 69 6D  sageBoxA@16__im
00004980:  70 5F 5F 4D-65 73 73 61-67 65 42 6F-78 41 40 31  p__MessageBoxA@1
00004990:  36 00 5F 4D-65 73 73 61-67 65 42 6F-78 57 40 31  6 _MessageBoxW@1
```

Рисунок 9 hex-редактор показывает истинное лицо MessageBoxA

Все равно ни хрена не понятно! Мы же уже сказали в модели памяти "stdcall" и транслятор послушно добавил ко всем функциям знак прочерка, но "забыл" дописать аргументы. А как бы он их дописал? Ведь прототип функции объявлен как "PROC"! Вот ассемблер и постеснялся разводить самодеятельность!

В комплекте с полной версией MASM'a идут inc-файлы, в которых все прототипы объявлены правильно, однако, в DDK ничего подобного нет и потому эту работу нам приходится выполнять самостоятельно и писать так: "`extern MessageBoxA@16:near`" или так: "`extern _imp__MessageBoxA@16:dword`" (в последнем случае функция будет вызвана через "переходник"). Если слово "stdcall" в модели памяти не указано и, следовательно, знак прочерка транслятором не добавляется, обе конструкции будет выглядеть так: "`extern _MessageBoxA@16:near`" и "`extern __imp__MessageBoxA@16:dword`" соответственно. Чтобы не вызывать каждый раз по "длинному" имени, создайте короткий алиас, обозвав ее хоть msgbox, хоть mb, хотя, во избежании путаницы, большинство программистов все-таки сохраняет оригинальные API-имена и убирает только "_" и "$x".

Законченный вариант программы будет выглядеть так.

```
.386
.model flat

extern _ExitProcess@4:near
extern _MessageBoxA@16:near

.data
s0 db 'hello, world',0

.code
_start:
        push 0
        push 0
        push offset s0
        push 0
        call _MessageBoxA@16

        push 0
        call _ExitProcess@4
end _start
```

Листинг 8 ассемблерная программа подготовленная к трансляции MASM'ом

Программа нормально транслируется и даже работает, но не дает ответа на вопрос — почему же ее создатель не выполнил все эти действия заранее?! Да потому, что программа предназначалась для TASM'a, библиотекарь которого именует функции так, как написано, а не так, как диктует соглашение о stdcall-вызовах. Но почему бы тогда не транслировать программу TASM'ом?! Тому есть свои причины. Во-первых, TASM заброшен и уже не развивается (впрочем, MASM ни хуя не развивается тоже), во-вторых, объектные файлы, сгенерированные TASM'ом трудно интегрировать в другие проекты. В-третьих, мыщъх'и испытывают к багдаду стойкую антипатию, непреодолимую даже пивом. Но если кому-то нравится TASM, то, пожалуйста! Компилируйте программу так:

```
rem ассемблируем
tasm32 /ml h2.asm

rem готовим библиотеки из dll
```

```
implib -c  user32.lib C:\WINNT\system32\user32.dll
implib -c  kernel32.lib C:\WINNT\system32\kernel32.dll

rem линкуем
tlink32 h2.obj -Tpe -aa -L user32.lib -L kernel32.lib
```

Листинг 9 ассемблирование программ на TASM'е

А нельзя ли ассемблировать нашу программу замечательным (и притом совершенно бесплатным) транслятором FASM? Увы! Различия в синтаксисе FASM'а очень значительные и без капитальной правки листинга здесь не обойтись. Вот только один пример. Широко распространенная конструкция "`DB 669h DUP(?)`" приводит FASM в состояние замешательства и ее приходится заменять на "`rb 669h`", что, несомненно, короче, но… это же сколько лишней работы по переносу делать приходится! Отсутствие offset'а мы уже отмечали. Привычных директив тоже нет. Макросредства как бы есть, но совсем не те, что в MASM'е и работают они совсем не так!

Однако, используя MASM, мы льем воду на мельницу Microsoft, а Microsoft это такая четырехсотфунтовая горилла, которая всех ебет. Какое к ней может быть отношения? Ну, отношения на самом деле бывают очень разные! Некоторые даже уверяют, что им это нравится. Дескать, у них с гориллой любовь, взаимность и полное согласие. Некоторые воспринимают горалу как объективную данность и стараются по возможности ее не замечать, ведь она совсем не злобная эта горилла, и ебет деликатно, совсем не маньячит. Другие ебут еще хуже. Но большинство гориллу все-таки ненавидит и хочет ее завалить. Что ж, вполне естественное желание, только в мире не одна горилла и после кончины m$ нас будет ебать кто-то другой... FASM не самый лучший ассемблер, тем не менее, мыщъх его любит и призывает всех его использовать.

После переделки под FASM программа будет выглядеть так:

```
include 'INCLUDE\win32ax.inc'

.code
start:
        push 0
        push 0
        push s0
        push 0
        call [MessageBox]

        push 0
        call [ExitProcess]
.data
s0 db 'hello, world',0

.end start
```

Листинг 10 программа hello.asm портированная в FASM

Как видно, исчезли директивы ".386", ".model", а в начале ключевого слова "end" появилась точка, которой раньше не было. Включаемый файл "win32ax.inc" содержит все необходимые определения и потому API-функции вызываются по их именам, заключенных в квадратики (косвенный вызов по ссылке, если делать иначе — все рухнет на хрен). Ключевое слово "offset" исчезло из инструкции "`push s0`". Вот, пожалуй, и все. Теперь транслятор пережевывает программу и не давится: "`fasm hello.asm`". Нам же остается только запустить созданный exe-файл на выполнение и порадоваться как хорошо он работает.

Рисунок 10 в мире FASM

заключение

Ассемблирование чужих программ представляет серьезную проблему, особенно если они распространяются без make-файлов и без указания чем и как их транслировать. Больше всех страдают начинающие астматики, совершенно не понимающие чего от них хочет эта тупая машина и неужели их руки настолько кривы, что даже не могут откомпилировать то, что написали другие. Не стоит высаживаться на измену и комплектовать по этому поводу. Если программа транслируется с ошибками (или не транслируется вообще) — она либо предназначена для другого транслятора, либо ноги вырвать ее создателю. Руки отрывать уже поздно, как говорит в этих случаях великий и могучий Юрий Харон, создатель не менее могучего линкера ulink и эмулятора NT, свободно умещающегося вместе с FAR'ом на одной дискете. Это настоящее искусство программирования, до которого нам, молодым, еще расти и расти!