

Крис Касперски

ОБРАЗ МЫШЛЕНИЯ IDA

(Отрывки из книги)

О ДИЗАССЕМБЛИРОВАНИИ ПРОГРАММ

Одним из способов изучения программ в отсутствии исходных текстов является дизассемблирование — перевод двоичных кодов процессора в удобочитаемые мнемонические инструкции. С первого взгляда кажется, что в этом нет ничего сложного, и один дизассемблер будет ничуть не хуже другого. На самом же деле, ассемблирование — это однонаправленный процесс с потерями, и, поэтому, строго говоря, автоматически восстановить исходный текст (разумеется, за исключением меток и комментариев) математически невозможно.

Часто одна инструкция имеет несколько различных кодов операций. Например, `ADD AX,1` может быть ассемблирована в следующие коды микропроцессора x86:

```
05 01 00
83 C0 01
81 C3 01 00
```

Таким образом, при повторном ассемблировании восстановленного текста мы не можем гарантировано получить тот же самый код, а, значит, полученная программа скорее всего откажется работать! Кроме того, любая попытка модификации дизассемблированного текста развалит программу окончательно. Дело в том, что ассемблер заменяет все метки на реальные смещения, т.е., иначе говоря, на константы. При внесении изменений в программу, необходимо скорректировать все ссылки на метки. Ассемблер это делает, руководствуясь директивой `OFFSET`.

Но дизассемблер не может отличить смещения от обычных констант!

исходная программа	ассемблированный	дизассемблированный текст
<code>MOV BX, Label_1</code>	<code>BB0001</code>	<code>mov bx, 0100h</code>
<code>JMP BX</code>	<code>FFE3</code>	<code>jmp bx</code>
<code>Label_1:</code>		

Дизассемблер неправильно восстановил исходный текст. Если после модификации программы `Label_1` окажется по адресу, отличному от `100h`, то переход произойдет на совершенно незапланированный участок кода, быть может даже в середину команды, что приведет к непредсказуемым результатам!

Выходит, дизассемблер должен отследить, как используется те или иные константы, и при необходимости предварять их директивой `OFFSET`. Нелегкая задача! Как насчет следующего примера?

исходная программа	ассемблированный	дизассемблированный текст
<code>MOV AX, offset Table</code>	<code>B81000</code>	<code>mov ax, 0010h</code>
<code>MOV BX, 200h ; index</code>	<code>BB0002</code>	<code>mov bx, 0200h</code>
<code>ADD AX, BX</code>	<code>01D8</code>	<code>add ax, bx</code>
<code>MOV AX, [BX]</code>	<code>8B07</code>	<code>mov ax, word ptr [bx]</code>

Ясно, что один из регистров — указатель на таблицу, а другой — индекс в этой таблице. Но что есть что, понять просто невозможно! Следовательно, с вероятностью близкой к единице полученный дизассемблером текст окажется неработоспособным.

Из этой ситуации есть два выхода — пытаться усовершенствовать алгоритм отслеживания ссылок, или организовать интерактивное взаимодействие с пользователем, полагаясь на его способности и интуицию.

Первое было реализовано в некогда уникальном дизассемблере `SOURCER`, равных которому в поиске перекрестных ссылок долгое время не было. Однако, на этом его достоинства и заканчивались. Кроме того, `SOURCER` оказался уязвимым перед различными хитрыми приемами программирования, такими, например, как самомо-

дифицирующийся (или зашифрованный) код. При этом он выдавал большое количество бессмысленных листингов, не дизассемблируя инструкции, а так и оставляя их в виде шестнадцатеричных чисел.

Ильфак Гуильфанов был первым, кто основной упор сделал не на совершенство алгоритмов, а на интерактивность взаимодействия с пользователем.

Дизассемблер из «черного ящика» превратился в чуткий и послушный инструмент, который в умелых руках мог творить чудеса. Впрочем, и обратное утверждение справедливо. Неопытный пользователь вряд ли много ожидал в подобной ситуации, и чаще всего обращался к автоматическим (наподобие SOURCER'a) дизассемблерам.

В январе 1991 года были написаны первые строки будущего дизассемблера. Очень удачным решением была поддержка программой встроенного Си-подобного языка. Это породило уникальный продукт с небывалыми до этого возможностями. Допустим, не нравится Вам, как SOURCER находит перекрестные ссылки или «спотыкается» на самомодифицирующемся коде. Что Вы можете сделать? Увы, ничего, только ждать новой версии и надеяться, что в ней это будет исправлено.

Встроенный язык позволит написать собственную версию процедуры анализа и тут же ее опробовать. Такими возможностями не обладает ни один другой дизассемблер! Если Вы хотите серьезно и глубоко заняться дизассемблированием программ, то кроме IDA Pro вряд ли подойдет что-то еще.

К сожалению, такая замечательная программа распространяется практически без документации. Это затрудняет изучение ее возможностей, большая часть из которых так и остается нераскрытой.

Данное издание является попыткой хотя бы частично восполнить этот пробел, а также дать Вам возможность разобраться в технологиях дизассемблирования.

ПЕРВЫЕ ШАГИ

Подвижность — это ключ к военному успеху, — говорил Тег. — Если ты связан крепостями, даже размером с целую планету, ты, по сути своей, уязвим.

Френк Херберт. «Еретики Дюны».

Давайте для начала рассмотрим простейший пример, который покажет, сколь сильно IDA отличается от других дизассемблеров. Возьмем программу состоящую всего из нескольких строк.

```
#include «stdafx.h»
#include <iostream.h>

int main(int argc, char* argv[])
{
    cout << «Hello, Sailor!»;
    return 0;
}
```

Однако, компилятор MS VC 6.0 сгенерировал исполняемый файл размером почти в 40 килобайт! Большая часть этого файла содержит служебный, стартовый или библиотечный код. Попытка его дизассемблировать, например WinDasm'ом, скорее всего, не увенчается быстрым успехом, поскольку над сгенерированным листингом размером в пятьсот килобайт можно на первых порах просидеть не час и не два. Что же тогда говорить о более серьезных задачах, сколько на них уйдет времени?

Попробуем тот же самый пример дизассемблировать с помощью IDA Pro. Если все настройки оставить по умолчанию, то после завершения анализа экран должен выглядеть следующим образом:

```
File Edit Navigate View Options Windows AU: idle READY 11:01:31
[ ] IDA view-A 2-[ ]
.text:00401011      retn
.text:00401011      sub_0_401000      endp
.text:00401011      ;
.text:00401012      ; align 10h
.text:00401020      ; [COLLAPSED FUNCTION start, 000000D4 bytes]
.text:004010F4      ;
.text:004010F4      ; SUBROUTINE
.text:004010F4      ;
.text:004010F4      sub_0_4010F4      proc near          ; DATA XREF: .rdata:004
.text:004010F4      mov     esp, [ebp-18h]
.text:004010F7      push  dword ptr [ebp-20h]
.text:004010FA      call   __exit
.text:004010FA      sub_0_4010F4      endp
.text:004010FA      ;
.text:004010FF      ; [COLLAPSED FUNCTION __amsg_exit, 00000025 bytes]
.text:00401124      ; [COLLAPSED FUNCTION sub_0_401124, 00000024 bytes]
.text:00401148      ; [COLLAPSED FUNCTION __cinit, 0000002D bytes]
00401020: start
Using FLIRT signature: UC v2.0/4.x/5.0 runtime
The initial autoanalysis is finished.
F1 Help C Code D Data N Name Alt-X Quit F10 Menu DISK: 1178M
```

Рис. 1. Интерфейс программы IDA Pro

«Сворачивание» функций очень упрощает навигацию по файлу, позволяя втиснуть больше информации в тесное пространство экрана. «Развернуть» функцию можно, подведя к ней курсор и нажав «+» на дополнительной цифровой клавиатуре. Соответственно, что бы свернуть, необходимо нажать «-». По умолчанию все библиотечные функции представляются свернутыми.

В нашем случае мы имеем дело с библиотечной функцией «START», а, точнее говоря, со сгенерированным компилятором стартовым кодом. Он выполняет инициализацию всех библиотек, подготавливает систему ввода-вывода и делает массу других дел, совершенно не интересующих нас на данный момент. Но в каком-то месте он передает управление функции *main()*, содержимое которой мы и пытаемся проанализировать.

Воспользуясь бы мы любым другим дизассемблером — нам пришлось бы первым делом тщательно изучить стартовый код компилятора в поисках места передачи управления на интересующую нас функцию (или заглянуть в исходные коды библиотек компилятора). Но первое трудоемко, а второе предполагает наличие у нас той же версии компилятора, что далеко не всегда выполнимо.

Все будет гораздо проще, если мы воспользуемся возможностью IDA находить перекрестные ссылки. Поскольку стартовый код вызывает только одну функцию (не считая библиотечных), то последняя и окажется искомой!

```
.text:00401000 sub_0_401000    proc near                ; CODE XREF: start+AF↓p
```

Прокрутим экран чуть дальше и рассмотрим следующую строку. Комментарий, указывающий на перекрестную ссылку, говорит, что эту процедуру вызывает стартовый код, и, если мы хотим взглянуть на него поближе, то нужно подвести курсор в границы выражения «start + AF↓p» и нажать «Enter». При этом IDA автоматически перейдет к требуемому адресу. Это, действительно, очень удобное средство навигации, аналогов которому я назвать затрудняюсь.

IDA распознает не только константы и символьные имена, но и сложные выражения и конструкции, причем независимо от того, как последние были созданы.

Попробуем нажать «Insert» и ввести следующую строку, которая будет отображена как комментарий «А сейчас мы перейдем по адресу 0x40103D». Если теперь подвести курсор к «0x40103D» и нажать «Enter», то IDA действительно перейдет по требуемому адресу! И возвращается назад клавишей «Esc». Это дает возможность организовывать в комментариях свои гиперссылки, позволяющие легко ориентироваться в исследуемом файле и быстро переключаться между разными фрагментами.

Но мы отвлеклись, вернемся назад и попробуем заглянуть в функцию *start()*. Увы, на этот раз IDA себя поведет не так, как ожидалось, и просто переместит курсор на свернутую функцию. Попробуем развернуть ее (клавишей «+» на дополнительной клавиатуре) и повторить операцию. На этот раз все проходит успешно. Интуитивно понятно, что должна быть функция авторазвертки при подобных переходах, но, по крайней мере, в версии 3.84 таковая отсутствует.

```
.text:004010A9    call    __setargv
.text:004010AE    call    __setenvp
.text:004010B3    call    __cinit
.text:004010B8    mov     eax, dword_0_408784
.text:004010BD    mov     dword_0_408788, eax
.text:004010C2    push   eax
.text:004010C3    push   dword_0_40877C
.text:004010C9    push   dword_0_408778
.text:004010CF    call   sub_0_401000
.text:004010D4    add     esp, 0Ch
.text:004010D7    mov     [ebp+var_1C], eax
.text:004010DA    push   eax
.text:004010DB    call   _exit
```

Как видно, `sub_0_401000` — единственная функция (за исключением библиотечных), вызываемая стартовым кодом. Следовательно, именно она и есть `main()`. Подведем курсор к `sub_0_401000` и нажмем `Enter`. Было бы неплохо дать ей осмысленное символьное имя, и IDA это позволяет. Для этого нужно подвести курсор к началу функции и нажать `N` (или в меню `View/Name` выбрать ее из списка всех функций, изменить которые можно нажатием «`Ctrl + E`», введя в открывшемся окне диалога любое осмысленное имя). В результате получится следующее:

```
.text:00401000 main proc near          ; CODE XREF: start+AF↓p
.text:00401000      push      offset aHelloSailor; «Hello, Sailor!»
.text:00401005      mov       ecx, offset dword_0_408900
.text:0040100A      call     ??6ostream@@QAEAAV0@PBD@Z
                                           ; ostream::operator<<(char)

.text:0040100F      xor      eax, eax
.text:00401011      retn
.text:00401011 main endp
```

Обратим внимание на строку `401000h`, а точнее на метку «`aHelloSailor`» — IDA распознала в ней строку символов и сгенерировала на основе их осмысленное имя, а в комментариях продублировала для наглядности оригинал. При этом, как уже отмечалось, IDA понимает символьные метки, и, если подвести к последней курсор и нажать на «`Enter`», то можно увидеть следующие:

```
.data:00408040 aHeloSailor db 'Hello,Sailor!',0 ; DATA XREF: main↑o
```

«`o`» — это сокращение от «`offset`», т.е. IDA позволяет уточнить тип ссылки. Ранее мы уже сталкивались с использованием в этом качестве символа «`p`», т.е. указателем (pointer). Иногда еще используется символ «`u`» (от слова `undefine`) — неопределенный, нераспознанный. О нем мы поговорим позднее. Стрелка (вверх или вниз, соответственно) указывает, где расположена указанная ссылка.

```
.text:0040100A      call     ??6ostream@@QAEAAV0@PBD@Z
                                           ; ostream::operator<<(char)
```

Сравним эту строчку с полученной ранее WinDASM'ом:

```
:0040100A          call    00403B81
```

Разумеется последняя гораздо менее информативна и потребует значительного времени на анализ функции `0403B81h` в попытках понять, что она делает, а, учитывая ее сложность и витиеватость, а так же то, что большая часть трех десятков килобайт программы и есть реализация этой функции, то может пройти не один час изучения вложенных вызовов, пока наконец ее смысл не станет ясен.

IDA сумела распознать в этой функции библиотечный оператор «`ostream::operator<<`», освободив нас от большой части работы. Но как она это проделала? Точно так, как антивирус распознает вирусы — по сигнатурам. Понятно, что бы этот механизм работал необходимо сначала создать базу сигнатур для библиотек распространенных компиляторов и оперативно ее обновлять и расширять. IDA, конечно, не всемогуща, но список поддерживаемых компиляторов очень впечатляющий (он расположен в каталоге `SIG/LIST`), при этом реально поддерживаются многие версии, даже не указанные в перечне, поскольку они часто имеют схожие сигнатуры.

Все функции имеют два имени: одно — которое дает им библиотекарь, и второе — общепринятое из заголовочных файлов. Если заглянуть в библиотеку используемого компилятора (в нашем случае это `MS VC 6.0`), то можно увидеть, что оператор «`cout <<`» есть ни что иное, как одна из форм вызова функции «`??6ostream@`

@QAEAAV0@PBD@Z», трудночитаемое имя которой на самом деле удобно для компоновщика и несет определенную смысловую нагрузку.

Вот, собственно, и все. Две следующие строки завершают выполнение *main()* с нулевым кодом возврата (эквивалентно `return 0`).

```
.text:0040100F          xor     eax, eax
.text:00401011          retn
```

Не правда ли, на анализ ушло совсем немного времени, и преимущества IDA в этом плане очевидны? Рассмотрим теперь другой, более сложный пример зашифрованной программы, который продемонстрирует эффективность встроенного языка.

Для начала попробует дизассемблировать файл `ida__0x1.exe` с помощью SOURCER'a. На самом деле это никакой не `.exe`, а самый настоящий `.com`. Операционную систему MS-DOS нельзя ввести в заблуждение неверным расширением, и она правильно определит его формат по отсутствию сигнатуры «MZ» в заголовке. SOURCER же в этой ситуации просто прекращает работу!

Переименуем файл и попробуем снова. Если все сделано правильно, SOURCER должен сгенерировать следующий листинг:

```
:0100 start:

:0100      add     si, 6
:0103      jmp     si                ;*
                                ;* No entry point to code

:0105      mov     cx, 14BEh
:0108      add     ds:data_1e[di], bp    ; (43CA:5691=0)
:010C      xor     byte ptr [si], 66h    ; 'f'
:010F      inc     si
:0110      loop   $-4                ; Loop if cx > 0
:0112      jmp     si                ;*
                                ;* No entry point to code

:0114      sbb    [bx+si], al
:0116      shr    byte ptr [bx-24h], cl    ; Shift w/zeros fill
:0119      db     6Eh, 67h, 0ABh, 47h, 0A5h, 2Eh
:011F      db     03h, 0Ah, 0Ah, 09h, 4Ah, 35h
:0125      db     07h, 0Fh, 0Ah, 09h, 14h, 47h
:012B      db     6Bh, 6Ch, 42h, 0E8h, 00h, 00h
:0131      db     59h, 5Eh, 2Bh, 0CEh, 0BFh, 00h
:0137      db     01h, 57h, 0F3h, 0A4h, 0C3h
```

Результат работы SOURCER'a очень похож на бред. Мало того, что половина кода осталась в шестнадцатеричном виде, но и то, что «распознано», дизассемблировано неверно. Листинг не позволяет понять, как работает программа.

Выше был продемонстрирован очень простой трюк против SOURCER-подобных дизассемблеров (распространяющийся, в том числе, и на TurboDebugger, а так же Hiew и QView). С первого взгляда не понятно, как можно работать с неинициализированным регистром, но на самом деле при загрузке `.com` файлов его значение всегда равно `100h`. Следовательно JMP в строке `103h` переходит по адресу `106h`, но обратите внимание, как это дизассемблировал SOURCER:

```
:0105  B9 14BE          mov     cx, 14BEh
```

Байт-«пустышку» `B9h` он принял за часть команды, в результате чего и получился такой результат. Разумеется, никакой дизассемблер не способен в совершенстве отслеживать регистровые переходы — эту часть работы должен выполнить человек, которого в отличие от машины таким простым приемом обмануть не удастся!

Проблема SOURCER'a в том, что это пакетный дизассемблер, и взаимодействие его с человеком очень затруднено. Совсем иначе дело обстоит с IDA, которая изначально проектировалась как интерактивная среда. Загрузим в нее файл и дождемся завершения автоанализа. На экране появится приблизительно следующее:

```

seg000:0100 start      proc near
seg000:0100             add     si, 6
seg000:0103             jmp     si
seg000:0103 start      endp
seg000:0103 ; -----
seg000:0105             db 0B9h ;
seg000:0106             db 0BEh ;
seg000:0107             db 14h ;

```

С первого взгляда это выглядит разочаровывающе. IDA дизассемблировала только первые две команды. А остальные? Увы, чтобы правильно распознать остальную часть кода потребовался бы нетривиальный интеллектуальный алгоритм. За неимением последнего IDA прекратила процесс, ожидая дальнейших команд от пользователя. SOURCER же самостоятельно пытается дизассемблировать с помощью различных внутренних алгоритмов как можно больше кода. Но зато, в результатах работы IDA можно быть уверенным, а SOURCER чреват ошибками в самых непредсказуемых местах.

Как уже отмечалось выше, JMP в строке 103h вызывает переход по адресу 106h. Попробуем объяснить это IDA. Добавим новую перекрестную ссылку, для чего в меню «View» выберем пункт «Cross references» и нажмем «Insert» для ввода нового элемента в список:

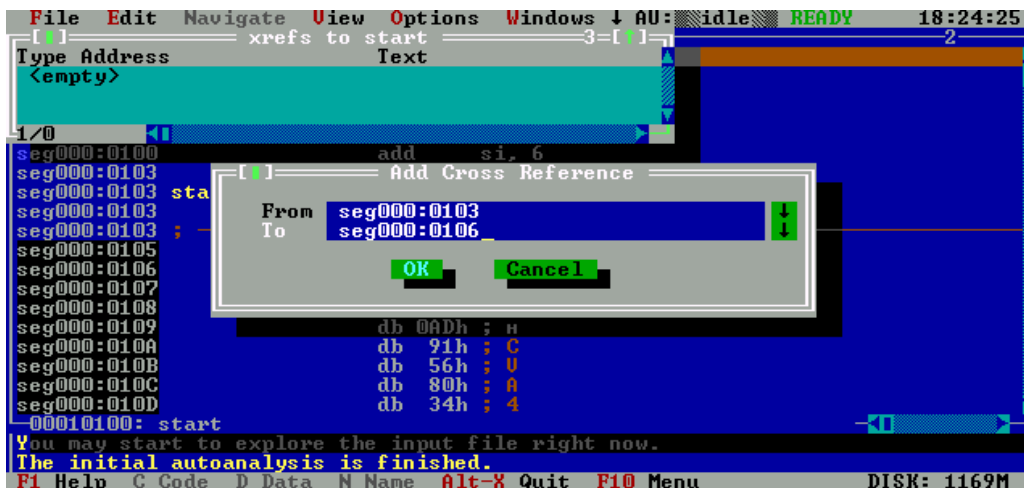


Рис.2. Добавление ссылки

В поле «From» введем адрес текущей строки, т.е. «seg000:0103», а в поле «To», соответственно — «:seg000:0106». При этом IDA автоматически начнет анализ программы:

```

seg000:0100 start      proc near
seg000:0100             add     si, 6
seg000:0103*           jmp     si
seg000:0103*start      endp
seg000:0103*
seg000:0103*; -----
seg000:0105             db 0B9h ;
seg000:0106 ; -----

```

```

seg000:0106
seg000:0106 loc_0_106:                ; CODE XREF: start+3↑u
seg000:0106                mov     si, 114h
seg000:0109                lodsw

```

Можно было бы поступить иначе — просто подвести курсор к строке 106h и нажать клавишу «С», (сокращение от «CODE»), но тогда непонятно, как этот код получает управление. В нашем примере это не критично, но в крупных проектах последнее всегда необходимо учитывать, а не пытаться все держать в голове, т.к., вернувшись к дизассемблированному тексту спустя месяц-другой (или передав его другому человеку), придется потратить немало времени, разбираясь в подобных деталях.

Обратим внимание, что IDA добавила только одну перекрестную ссылку, и, по-прежнему, «jmp si» указывает «в никуда». Для определения адреса перехода приходится выполнять вычисления в уме и помнить, чему равно значение регистра SI. Не очень удобно, правда?

Чтобы все это не держать в уме, попробуем добавить комментарий. Для этого нажмем клавишу «:» и введем строку, например, следующего содержания «SI == 106h». Это не только разгрузит нашу голову, но еще и упростит навигацию — достаточно подвести курсор к «106h» и нажать на «Enter», как IDA автоматически переместиться на искомую строку!

```

seg000:0103*                jmp     si                ; SI == 106h

```

Конечно, можно было просто добавить еще одну перекрестную ссылку, однако, никакой необходимости в этом нет.

Рассмотрим следующий фрагмент кода:

```

seg000:0106 loc_0_106:                ; CODE XREF: start+3↑u
seg000:0106                mov     si, 114h
seg000:0109                lodsw

```

Что такое 114h в строке 106h — константа или смещение? Несомненно, смещение, поскольку следующая за ним команда «lodsw» загружает в AX слово, на которое указывает регистр SI. В некоторых случаях IDA способна распознать смещения, но в большинстве случаев это, конечно, приходится за нее делать человеку.

Подведем курсор к «114h» и нажмем «O», при этом должно получиться приблизительно следующее:

```

seg000:0106 loc_0_106:                ; CODE XREF: start+3↑u
seg000:0106                mov     si, offset unk_0_114

```

Что же именно грузится в SI? Это можно узнать, проанализировав код далее по листингу:

```

seg000:0109                lodsw
seg000:010A                xchg   ax, cx
seg000:010B                push   si
seg000:010C
seg000:010C loc_0_10C:                ; CODE XREF: seg000:0110↓j
seg000:010C                xor    byte ptr [si], 66h
seg000:010F                inc    si
seg000:0110                loop   loc_0_10C
seg000:0112                jmp    si

```

Как видно, эта величина помещается в регистр CX и используется в цикле расшифровщика. Следовательно, этой переменной уже можно дать осмысленное имя! Подведем курсор к «unk_0114» и нажмем на «Enter»:

```
seg000:0114 unk_0_114      db  18h ;          ; DATA XREF: seg000:0106↑o
seg000:0115                db   0 ;
```

Но, для начала, следует правильно указать тип переменной, который, очевидно, равен слову. Если нажать «D», то IDA будет циклически перебирать все известные ей типы: byte, word, dword и т.д.

Теперь нажмем «N» и дадим метке какое-нибудь осмысленное имя. Не стоит бояться длинных имен. Не экономьте на этом время! Оно потом окупится удобочитаемостью листинга:

```
seg000:0114 LengthCryptCode dw 18h          ; DATA XREF: seg000:0106↑o
```

Переходим к следующей, наиболее трудной части анализа. Разумеется, дизассемблер не может анализировать зашифрованный код, и нам предстоит его расшифровать самостоятельно. До появления IDA, дизассемблеры были не способны ничем помочь, и приходилось прибегать к утилитах типа Niew, где вручную изменять сам файл. Сейчас же в этом нет никакой необходимости. Встроенный язык IDA позволяет с легкостью манипулировать образом загруженного файла по нашему желанию, не трогая при этом оригинал.

Теоретически, даже возможно написать подключаемый модуль (plugin), выполняющий автоматическую распаковку — это вполне осуществимая задача. И, в последствии, мы рассмотрим его в этой книге, но пока попытаемся это сделать вручную.

Собственно, все, что нам потребуется — проанализировать распаковщик программы и переписать его на встроенном Си-подобном языке.

Для начала определим длину зашифрованного фрагмента. Как видно, в строке 109h считывается слово, которое затем помещается в счетчик CX, численно равное длине шифротекста в байтах. Следом за ним начинается собственно сам шифротекст. Значение SI при этом равно $\text{offset LengthCryptCode} + 2 = 114h + 2 = 116h$. Перейдем по этому адресу и создадим новую переменную «CryptCode», а также добавим новую гиперссылку на строку 10Bh.

```
seg000:0106 mov  si, offset LengthCryptCode ; На начало расшифровываемых
                                                ; данных
seg000:0109 lodsw                ; Читаем слово
seg000:010A xchg  ax, cx          ; CX == длина шифротекста
seg000:010B push  si            ; SI == первый байт шифротекста
seg000:010C                ; SI == 116h
seg000:010C Repeat:           ; CODE XREF: seg000:0110↓j
seg000:010C xor   byte ptr [si], 66h ;Расшифровываем очередной байт
seg000:010F inc   si            ; Указатель на следующий байт
seg000:0110 loop  Repeat       ; Цикл
seg000:0112 jmp   si            ; Переход LengthCryptCode+1Ah
```

Сам цикл расшифровки невероятно прост и не должен вызвать затруднений. Давайте обратим внимание на еще один регистровый переход — JMP SI. Чему равно значение SI? Очевидно, что $\text{SI} = \text{offset CryptCode} + \text{LengthCryptCode} + 2$. При этом этот код не зашифрован и может быть немедленно дизассемблирован!

Нажмем «G» (переход по адресу) и введем, например, следующее: «LengthCryptCode + 1Ah». Нажмем «C», чтобы дизассемблировать этот фрагмент кода:

```
seg000:012E      call  $+3
seg000:0131      pop   cx
seg000:0132      pop   si
seg000:0133      sub   cx, si
```

```

seg000:0135          mov     di, 100h
seg000:0138          push   di
seg000:0139          repe  movsb
seg000:013B          retn
seg000:013B seg000  ends

```

Профессионалы, наверное, еще до завершения анализа догадались, что этот код перемещает расшифрованный фрагмент в памяти по адресу 100h. Это наводит на мысль, что шифровщик разрабатывался независимо от основной программы и является «конвертной» защитой.

Однако, не исключено, что используемые им приемы неизвестны начинающим, поэтому ниже они будут подробно рассмотрены. «CALL \$ + 3» передает управление по адресу 131h, т.е. с первого взгляда не несет никакой полезной нагрузки. На самом деле оно заносит в стек регистр IP, а следующая команда POP CX копирует это значение в CX. Эта конструкция по смыслу эквивалентна MOV CX,IP, но, поскольку такой команды в наборе процессора x86 нет, то программе приходится ее эмулировать.

Если вернуться назад (Вы ведь добавили перекрестную ссылку), то можно обнаружить, что последним в стек было занесено смещение зашифрованных (но теперь-то уже расшифрованных) данных. Следовательно, SI = offset CryptedCode.

Какую смысловую нагрузку несет CX? Это смещение конца зашифрованного фрагмента плюс три байта на команду CALL. С первого взгляда кажется, что SUB CX,SI работает некорректно, т.к. неправильно вычисляет длину. Верно, реальная длина должна быть короче на три байта, но к чему такая точность? В любом случае содержимое памяти за концом зашифрованного блока не гарантируется и не должно влиять на его работу (при условии, что он написан правильно), и можно перемещать блок любой длины, лишь бы при этом он не затер код ниже строки 138h, иначе его дальнейшее выполнение станет невозможным.

Передача управления реализована через RETN (с засылкой в стек 100h — значения регистра DI). На первый взгляд это ничуть не короче JMP SHORT 100h. На самом деле гораздо короче. Дело в том, что JMP *const* — относительный переход, а на момент компиляции приложения текущее смещение неизвестно, и его необходимо вычислить. Для этого потребуются несколько команд ассемблера. Кроме того, не всегда короткого перехода будет достаточно.

Поскольку IDA не может отследить адрес перехода посредством RETN, то добавим самостоятельно еще одну перекрестную ссылку:

```

seg000:013B locret_0_13B:          ; CODE XREF: seg000:0116↑u
seg000:013B          retn

```

Нет, на самом деле это никакая не ошибка! Конечно, «физически» RETN переходит к строке 100h, но в дизассемблере там расположен совершенно другой код, поэтому переход к строке 116h логически оправдан.

Переходим к самому сложному. К созданию скрипта, расшифровывающего зашифрованный код. Задачу можно решить двумя способами — использовать функции ввода/вывода IDA и модифицировать непосредственно изучаемый файл, а потом его перезагрузить, или манипулировать только его образом в памяти. Вряд ли требуется доказывать, что второй способ предпочтительнее.

Перед тем, как начинать работу, необходимо познакомиться с организацией виртуальной памяти IDA. Подробнее она будет рассмотрена позже, а пока рассмотрим упрощенную модель. Она имеет очень много общего с так называемой сегментной моделью памяти, используемой семейством процессоров x86. При этом положение каждой ячейки определяется парой чисел «сегмент:смещение». Если дизассембли-

руемая программа предполагает *линейную* (flat) модель памяти, то все равно создается хотя бы один сегмент (как, например, в нашем случае с .com-файлом есть один сегмент, хотя сам .com-файл об этом и «не подозревает»).

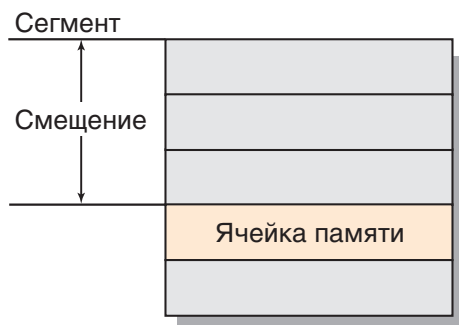


Рис. 3. Обращение к ячейке памяти

Таким образом, для доступа к произвольной ячейке нужно знать сегмент, в котором она расположена, и ее смещение. Однако, «seg000» это в действительности не нулевой сегмент, а не более чем символическое имя. Для доступа к виртуальной памяти его необходимо заменить на *базовый адрес*.

Чтобы узнать его, заглянем в меню View/Segments. Появится следующее окно:

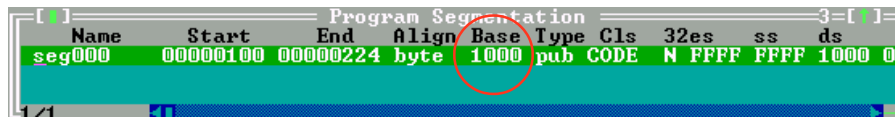


Рис.4. Информация о сегментах

«BASE» — это и есть искомый базовый адрес. Учитывая, что один параграф равен 16 байтам, можно вычислить линейный виртуальный адрес начала зашифрованного кода. Он будет равен:

$$\text{seg000:offset CryptedCode} = 1000\text{h}:116\text{h} = 1000\text{h} \ll 4 + 116\text{h} = 10000\text{h} + 116\text{h} = 10116\text{h}.$$

Осталось теперь только узнать, какими командами IDA производит чтение/запись в память. Но здесь нас ждет большое разочарование — контекстная помощь, начиная, по крайней мере, с версии 3.7 сильно ухудшена. Если раньше библиотека помощи по функциям встроенного языка была разбита на тематические категории, то сейчас все сгруппированно в один длинный список с минимальными средствами навигации.

Лучше будет использовать файл определений IDC\idc.idc, изрядно покопавшись в котором, рано или поздно мы найдем:

```
long   Byte           (long ea);           // get a byte at ea
void   PatchByte      (long ea, long value); // change a byte
```

Первая читает байт, вторая соответственно его записывает. Кроме того, предусмотрен удобный макрос для преобразования адресов МК_FP:

```
long   МК_FP          (long seg, long off); // the same as [ seg, off]
// i.e: ((seg<<4)+off)
```

Он позволит уменьшить количество вычислений. Остается только написать программу. Для этого вызовем консоль нажатием «Shift + F2» и введем следующий текст:

```
auto a;
auto temp;
for (a=0x116; a<0x116+0x18; a++)
{
    temp=Byte(МК_FP(0x1000, a));
}
```

```

temp=temp ^ 0x66;
PatchByte(МК_FP(0x1000, a), temp);
}

```

Думаю, нет необходимости объяснять, как он работает. Единственным неочевидным моментом будет объявление переменных. IDA не поддерживает привычное нам объявление типов. Вместо этого используется ключевое слово «auto», и, в дальнейшем, тип переменной определяется автоматически по ее использованию.

Запустим этот скрипт на выполнение, и, если все сделано правильно, то зашифрованный фрагмент должен немедленно измениться и выглядеть так:

```

seg000:0116 CryptedCode      db 0B4h ;           ; CODE XREF: seg000:010B↑u
seg000:0117                  db  9 ;
seg000:0118                  db 0BAh ;
seg000:0119                  db  8 ;
seg000:011A                  db  1 ;
seg000:011B                  db 0CDh ;
seg000:011C                  db 21h ; !

```

Подведем курсор к строке 116h и нажмем «С», что бы преобразовать его в код:

```

seg000:0116 CryptedCode:          ; CODE XREF: seg000:010B↑u
seg000:0116                  mov     ah, 9
seg000:0118                  mov     dx, 108h
seg000:011B                  int     21h           ; DOS - PRINT STRING
seg000:011D                  retn
seg000:011E                  db 48h ; H
seg000:011F                  db 65h ; e
seg000:0120                  db 6Ch ; l
seg000:0121                  db 6Ch ; l
seg000:0122                  db 6Fh ; o

```

Это действительно получилось! Код был успешно расшифрован, и для этого не потребовалось выходить из интегрированной среды и модифицировать оригинальный файл. Однако, часть кода после RETN не была дизассемблирована. Почему? Присмотревшись к комментариям (отображающим ASCII-представление каждого байта), нетрудно догадаться, что это и не код вовсе, а текстовая строка. Перевести ее в более читабельный вид можно нажатием «А», при этом курсор должен находиться в начале строки:

```

seg000:0116 CryptedCode:          ; CODE XREF: seg000:010B↑u
seg000:0116                  mov     ah, 9
seg000:0118                  mov     dx, 108h
seg000:011B                  int     21h           ; DOS - PRINT STRING
seg000:011D                  retn
seg000:011E aHelloSailor      db 'Hello, Sailor! ', 0Dh, 0Ah, '$

```

Однако, полученный результат, строго говоря, неверен и дизассемблированный код работать не будет. В самом деле, сравните значение загружаемое в регистр DX со смещением выводимой строки. Разумеется, они различаются, поскольку мы забыли переместить код по адресу 100h!

Но невозможно переместить код, не затерев при этом расшифровщик! В работающей программе это не вызывает проблем, ведь предыдущий код уже отработал и не нужен, но в дизассемблере это делать крайне нежелательно, т. к. при этом часть кода, а вместе с ней и логики работы, будет необратимо утеряна.

Более «цивилизованным» способом будет создание еще одного сегмента, куда следует скопировать расшифрованный код. Это можно сделать как программно, так и интерактивно. Оба способа так или иначе, сводятся к вызову следующей функции:

```
success SegCreate(long startea, long endea, long base,
                  long use32, long align, long comb);
```

Первый слева параметр `startea` — адрес начала, а второй — `endea`, соответственно, конца сегмента; `base` задает линейный виртуальный базовый адрес. Атрибуты сейчас разбирать не будем, а заполним их нулями. Это не совсем правильно, но для рассматриваемого примера вполне сойдет. Т.е. вызов функции в нашем случае должен выглядеть так:

```
SegCreate(MK_FP(0x2000, 0x100), MK_FP(0x2000, 0x118), 0x2000, 0, 0, 0);
```

2000h — это базовый адрес нового сегмента. Легко видеть, что теперь между двумя сегментами образуется довольно большое свободное пространство в виртуальной памяти, однако ввиду страничной организации виртуальной памяти и динамического выделения адресов (страница выделяется только, когда она действительно требуется), это не создает проблем, но зато экономит время на расчетах.

Другим способом создания сегмента будет переход в меню `View/Segments`, нажатие «`Insert`» и заполнение появившегося диалога аналогичным образом. Возможно, этот способ Вам покажется удобнее, тем более, что он позволит дать сегменту любое имя на ваш вкус, например, «`MySeg`».

Теперь в созданный сегмент необходимо скопировать исследуемый фрагмент. Интерактивно это сделать невозможно, и придется вновь возвращаться к консоли. Впрочем, в комплект IDA входит скрипт, реализующий копирование фрагментов памяти, и вместе с макросами можно было бы организовать неплохое интерактивное взаимодействие с пользователем, но это мы рассмотрим позднее, а сейчас попробуем написать такой скрипт самостоятельно:

```
auto a;
auto temp;
for (a=0x116; a<0x116+0x18; a++)
{
    temp = Byte(MK_FP(0x1000, a));
    PatchByte(MK_FP(0x2000, a-0x16), temp);
}
```

Собственно, ничего сложного в этом нет. И, если все было сделано правильно, наш фрагмент будет скопирован. Теперь необходимо дизассемблировать его. Однако, это можно сделать не только интерактивно, но и посредством консоли. При помощи такой функции:

```
long MakeCode (long ea)
```

В нашем случае вызов будет иметь вид:

```
MakeCode(MK_FP(0x2000, 0x100))
```

Аналогично можно перевести неопределенные данные в ASCII-строку. Однако, попробуем усилить интерактивность скрипта и не будем жестко задавать линейный адрес, а попробуем считать текущий:

```
MakeStr(ScreenEA(), BADADDR)
```

Этот пример требует небольших пояснений. «`BADADDR`» — это специально зарезервированная константа, которая указывает, что адрес конца строки не задан пользователем и будет вычисляться ядром IDA.

Но сейчас мы попробуем написать свой скрипт, который будет преобразовывать данные в ASCII-символы и самостоятельно определять длину строки. Создадим файл `String.idc` следующего содержания:

```
static MyMakeStr()
{
    auto a,b;
    auto temp;
    a=ScreenEA();
    temp=a;
    while(1)
        if (Byte(temp++)=='$') break;
    MakeStr(ScreenEA(),temp);
}
```

Это уже полноценная программа, которая после загрузки останется в памяти IDA, о чем говорит ключевое слово «static», и будет доступна для вызова с консоли при помощи «MyMakeStr();». Очень удобное средство наращивания возможностей IDA — если Вам не нравится, как работает та или иная функция, то можно создать свою!

А теперь обратим внимание, что в регистр DX по прежнему загружается константа, а не смещение:

```
MySeg:0102                mov     dx, 108h
```

Что бы исправить это, необходимо подвести курсор к «108h» и нажать «Ctrl + O». Почему «Ctrl + O», а не просто «O» (make offset)? Дело в том, что команда «make offset» определяет смещение относительно сегмента данных, а точнее того сегмента, на который указывает регистр DS. «Ctrl + O» определяет смещение относительно текущего сегмента. При сознании нового сегмента мы не позаботились о том, что бы выставить значение регистра DS, и он остался неопределенным:

```
MySeg:0100                assume es:nothing, ss:nothing, ds:nothing
```

Отредактировать это значение можно, нажав «Alt + G», и явным образом указав сегмент, в нашем случае «MySeg». При этом у Вас должно получиться следующее:

```
MySeg:0100 MySeg segment byte public '' use16
MySeg:0100         assume cs:MySeg
MySeg:0100         ;org 100h
MySeg:0100         assume es:nothing, ss:nothing, ds:MySeg, fs:nothing
MySeg:0100         mov     ah, 9
MySeg:0102         mov     dx, offset aHelloSailor_0 ; "Hello,Sailor!\r\n$"
MySeg:0105         int     21h                ; DOS - PRINT STRING
MySeg:0107         retn
MySeg:0108 aHelloSailor_0 db 'Hello,Sailor!',0Dh,0Ah,'$'
                                                ; DATA XREF: MySeg:0102↑o

MySeg:0108 MySeg         ends
```

На этом работу можно считать почти законченной, осталось только скорректировать перекрестные ссылки. Там, например, в строке `seg000:013B` ссылка будет указывать не на `seg000:0116h`, а на `MySeg000:100h`.

Но готов ли наш листинг к компиляции? Или точнее, как после внесения в него изменений получить вновь работоспособный файл? Традиционно потребовалось бы ассемблировать по отдельности разные куски выходного файла, а затем «склеить» в один, предварительно зашифровав, например, с помощью `hiew`. Довольно утомитель-

но, а, главное, это предполагает наличие ассемблера, компоновщика и еще утилиты шифрования на диске.

IDA же содержит встроенный ассемблер, а для шифрования можно использовать ранее написанный скрипт. При этом можно получить сразу готовый к употреблению .com-файл, а не .asm, как это делают другие дизассемблеры.

Давайте в качестве упражнения доработаем дизассемблированный пример, добавив в него, например, ожидание нажатия на клавишу после вывода строки и изменим саму текстовую строку. Для этого подведем курсор к строке 107h (RETN) и вызовем встроенный ассемблер — EDIT/Patch program/Assembler. Введем, например, следующую последовательность команд:

```
XOR  AX, AX
INT  16h
RETN
```

IDA ассемблирует, записав ее поверх строки «Hello, Sailor». С первого взгляда это выглядит разочаровывающе — к чему такой ассемблер? — и, вероятно, многих склонит к использованию полноценных TASM или MASM. А напрасно, как уже отмечалось, уникальность IDA в ее наращиваемой архитектуре. Если вам не нравится как работает та или иная команда... — это может быть лозунгом любого IDA-пользователя.

Конечно, наилучшим решением было бы написание собственного «полноценного» ассемблера, интегрированного в IDA с помощью механизма plugin'ов. Более простой вариант — отслеживать все ссылки и автоматически «раздвигать» их. Впрочем, для нашего примера это абсолютно не нужно.

Прежде чем менять строку, «раздвинем» границы сегмента, иначе может не хватить места. Нажатием «Alt + S» вызовем диалог редактирования атрибутов сегмента и увеличим конечный адрес, например до 20200h (этого хватит даже для очень длинной строки, а если вдруг и не хватит, значение можно будет увеличить еще раз). При этом IDA затребует подтверждения следующим диалогом:

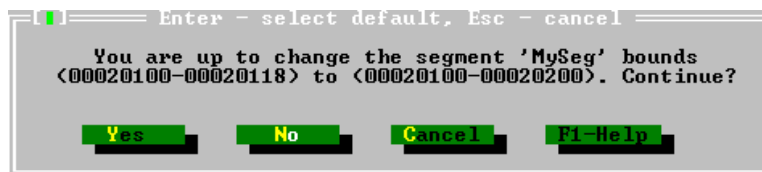


Рис.5. Подтверждение изменения границ сегмента

Заглянув в помощь, можно понять причину беспокойства IDA: «Caution: moving the first segment of the program will delete all information about the bytes between the old start of the segment and the new start of the segment! (При перемещении первого программного сегмента будет удалена вся информация о байтах, расположенных между старым началом сегмента и новым!)» Очевидно, что это к нашему случаю не относится, поэтому без колебаний нажимаем «Enter».

Изменить строку можно как простейшим скриптом, так и интерактивно. Для последнего вызовем EDIT/Patch program/Change byte... и введем, например, «Hello, IDA PRO!\$», обязательно в кавычках.

Теперь необходимо собрать и вывести дизассемблированную программу в файл. Начнем с того, что запишем в файл расшифровщик. Это можно сделать следующим образом:

```
static main() {
    auto f;
    auto a;
```

```

auto temp;
Message("aaaa");
f=fopen("ida__0x1.com", "wb");
for (a=0x100; a<0x114; a++)
    fputc(Byte(MK_FP(0x1000, a)), f);

```

Комментировать этот скрипт, я думаю, нет необходимости, т.к. файловый ввод/вывод у IDA ничем не отличается от классического Си. А вот дальше комментарии, вероятно, потребуются. Как мы помним, в оригинальной программе следующее слово содержало длину зашифрованного фрагмента. Очевидно, что после внесения изменений в последний длину необходимо вычислять заново. Причем самое интересное, что в точном задании длины в принципе никакой необходимости нет, можно взять заведомо большее значение. Конечно, при этом длина файла излишне возрастет, но так ли это критично? Добавим еще одну строчку к программе:

```
writeshort (f, 0x100, 0);
```

Теперь необходимо зашифровать «MySeg» и дописать его к файлу. Однако, не будем спешить — мы забыли откорректировать ссылку на выводимую строку:

```

MySeg:0102          mov     dx, offset loc_1000_107+1
.....
MySeg:010C aHelloIdaPro  db 'Hello, IDA PRO! $'

```

Теоретически это можно сделать специально написанным (и весьма хитрым скриптом), но пока изменим ссылку вручную. Разумеется, для этого пригодится функция *PathByte()* или встроенный ассемблер. В последнем случае необходимо ввести команду MOV DX, 10Ch и нажать «О», чтобы IDA распознала в этой константе смещение.

Следующий фрагмент шифрует код на лету и записывает его в файл:

```

for (a=0x100; a<0x200; a++)
{
    temp = Byte(MK_FP(0x2000, a));
    temp = temp ^ 0x66;
    fputc(temp, f);
}

```

Остается только в конец файла дописать последний фрагмент, который выполнит перемещение расшифрованных данных. Это можно реализовать следующим образом:

```

for (a=0x12e; a<0x130; a++)
    fputc(Byte(MK_FP(0x1000, a)), f);
fclose(f);

```

Объединим все эти фрагменты в один файл (ida__0x1.idc) и запустим его на выполнение. Если мы все сделали верно, то будет создан файл ida__0x1.com, который, при запуске, выведет "Hello, IDA PRO!" и, дождавшись нажатия на любую клавишу, выйдет в MS-DOS.

Если же что-то не работает, или работает не так, то вероятнее всего были допущены ошибки. Проверьте дизассемблированный листинг еще раз, а так же все скрипты. При этом дизассемблированный код можно перевести вновь в неопределенный, подведя курсор к нужному фрагменту и нажав «U». Однако, помните, что при этом будут необратимо утеряны все метки, комментарии и перекрестные ссылки, которые, Вы задавали, поэтому будьте внимательны!

Действительно, IDA обладает уникальными возможностями в этом плане. Обратите внимание, что файл получен в обход дизассемблирования. Мы просто читали бай-

ты из виртуальной памяти так, как они были представлены в оригинальном файле. Таким образом, нет никакого риска нарваться на ошибки дизассемблера. Даже если какие-то фрагменты были бы дизассемблированы неправильно, это никак бы не отразилось на конечном результате, поскольку файл читался из виртуальной памяти «как он есть», а не как он был дизассемблирован. Это принципиальное различие!

Если кажется, что сборка файла требует излишне много действий, то попробуйте проделать то же, например, TASM'ом. Это займет не меньше действий, а кроме того потребует написания программ для шифровки текста (или использования `hiew`, который, кстати, не может шифровать в автономном режиме).

Таким образом, IDA — это уникальный инструмент, позволяющий не только дизассемблировать, но и вносить необходимые изменения в листинг и тут же его компилировать. При этом всегда существует возможность написания сложных вставок на внешнем ассемблере с последующей их загрузкой в виртуальную память дизассемблируемого файла.

Два примера, рассмотренные выше, демонстрируют превосходство IDA над другими существующими сегодня дизассемблерами. Гибкий механизм `plugin'ов` позволяет даже создать интегрированный отладчик (как, например, в WinDasm'e). IDA — это невероятно мощный инструмент, возможности которого безграничны.

Точнее, ограничены одним лишь опытом и талантом пользователя. Разумеется, немислимо решение серьезных задач без четкого понимания архитектуры используемого инструмента. Ее рассмотрением мы сейчас и займемся.

ОРГАНИЗАЦИЯ ВИРУАЛЬНОЙ ПАМЯТИ

Память услужливо подсказала ему слова Гурни Хэллека: «Тот, кто хорошо владеет ножом, должен знать возможности любой его части: и острия, и лезвия, и щеки. Острие умеет резать, лезвие может колоть, щека может поставить ловушку противнику».

Френк Херберт. «Дюна».

Пользователи большинства дизассемблеров (например, SOURCER'a) едва ли представляют, как эта программа распоряжается выделенной ему памятью. Все что они могут сказать — какая память собственно используется (EMS или страничный файл подкачки) но не как она используется. Все это скрыто от пользователя, да и совершенно ему не интересно. К тому же никто не гарантирует, что в следующей версии не будет иначе.

Совсем другое дело — IDA. Поскольку это не только интерактивная оболочка, но и «виртуальная машина», со своим встроенным языком программирования и механизмом внешних модулей — plugin'ов, то организация ее памяти — один из первых моментов, с которым сталкиваешься при написании сложных скриптов или манипуляций сегментами, хитрыми зашифрованными программами, да и в ряде других случаях.

Не спорю, можно использовать IDA в полностью автоматическом режиме, совершенно не интересуясь ее внутренним построением. Однако, по большому счету, IDA создавалась не для автоматической работы и, по крайней мере до версии 3.7, требовала тесного взаимодействия с пользователем. Текущие версии содержат превосходные «интеллектуальные» алгоритмы, которые разгружают пользователя настолько, что в ряде случаев даже не требуют его вмешательства, и сразу генерируют правильно дизассемблированных листинг.

Однако, в более сложных случаях это не так, что и демонстрировалось в первой главе. Даже для создания простейшего скрипта из нескольких строк пришлось хотя бы поверхностно рассмотреть организацию виртуальной памяти IDA. В этой главе рассмотрим ее подробнее.

Итак, виртуальная память. «Память» — это понятно, но почему «виртуальная»? Дело в том, что IDA, как и другая «виртуальная машина», не предоставляет доступа к физической памяти компьютера. Она имеет собственное виртуальные (т.е. мнимое) пространство адресов, с которыми работают скрипты, plugin'ы и дизассемблируемые программы. Где это пространство расположено физически — на диске или в оперативной памяти — не так важно.

Прежде чем разбирать плоскости памяти, рассмотри одну уникальную, но увы, уже почему-то недокументированную возможность: а именно доступ к физической памяти из «виртуальной машины» IDA. С одной стороны это вопиющая некорректность и идет в разрез с самой идеологией «виртуальных машин», но... с другой стороны, это очень удобно. Можно, например, делать копию участка памяти работающего приложения и тут же его дизассемблировать, не выходя из оболочки. Традиционно для этого требовалось бы сохранить этот участок в файл и только потом уже загружать его в дизассемблер. Лишняя, никому не нужная, а к тому же очень утомительная работа.

Следующий скрипт копирует BIOS в созданный «налету» сегмент и дизассемблирует, например, дисковый сервис Int 13h. Удобства от такого подхода достаточно очевидны. Любой другой дизассемблер потребовал бы предварительной записи BIOS в

файл и последующих за этим объяснений, по какому адресу этот файл следует загрузить, и где находится точка входа.

```

auto a;
auto temp;
SegCreate(0xF000, 0xFFFF, 0x0F000, 0, 0, 0);
Message("Ждите... читаю BIOS...");
for (a=0; a<0xFFFF; a++)
{
    temp=_peek(MK_FP(0xF000, a));
    PatchByte(MK_FP(0xF000, a), temp);
}
Message("OK \n Дизассемблирую Int 0x13");
MakeCode(MK_FP(0xF000, 0xEC59));
Message("OK \n");
Jump(MK_FP(0xF000, 0xEC59));

```

Самое интересное, что можно не только читать, но и записывать в физическую память! Т.е. можно использовать IDA для исправления программ «налету» непосредственно в оперативной памяти! Например, для нейтрализации вируса или отладки резидента, да и просто стыковки с внешними программами. Это уже в меру Ваших нужд и фантазии.

Для этого можно использовать следующую функцию:

```

long    _poke    (long RAMEa, long value);    // poke a byte into RAM
                                              // returns old value

```

Любопытно, что функции `_peek` и `_poke` в текущей версии исключены из контекстной помощи IDA, а так же из файла определений `Idc.idc`, но тем не менее все равно успешно работают, что и доказывает приведенный выше скрипт!

Однако, необходимо помнить, что операции чтения/записи не контролируются дизассемблером и могут привести к печальным последствиям при неправильном их использовании. Думаю, нет нужды говорить, что бездумная запись в память рано или поздно приведет к сбоям или зависанию операционной системы или дизассемблера. Похожая картина наблюдается и с чтением. Этот с первого взгляда абсурдный момент требует пояснений. Разве операция чтения может вызвать фатальный сбой? Идеология MS-DOS приучила нас к бесконтрольному манипулированию операционной системой и ее служебными структурами данных. Однако, под Win32 уже можно ограничивать доступ в равной степени, как от записи, так и от чтения, поэтому вызов функции `_peek(1)`, выполненный в версии IDA для Win32 (`idaw.exe`), вызовет зависание дизассемблера. Впрочем, процесс еще можно завершить (например, закрытием окна), но увы — все данные не будут сохранены и окажутся безвозвратно утраченными!

Вероятно, поэтому данные функции и были исключены из помощи. А были оставлены только потому, что действительно в некоторых ситуациях могут быть невероятно полезными!

Итак, мы уже узнали, по крайней мере, одну плоскость памяти IDA — физическую память. При этом независимо от текущей модели и режима микропроцессора эта плоскость всегда линейна и адресуется 32-разрядным указателем. Однако, это никак не означает, что в Вашем распоряжении находится 4 гигабайта памяти. Например, выполнение функции `_peek(1000000h)` в версии IDA для MS-DOS (`idax.exe`) скорее всего приведет к ситуации, изображенной на рис. 6, т.е. мы вышли за пределы блока памяти, выделенного DPMI, что и послужило причиной аварийного завершения работы приложения. Это еще один из поводов для осторожности при работе даже с фун-

кцией чтения. Необходимо быть уверенным, что IDA для своих нужд выделила блок достаточного размера, чтобы не залезть в чужой сегмент, с не замедлившими себя ждать последствиями.

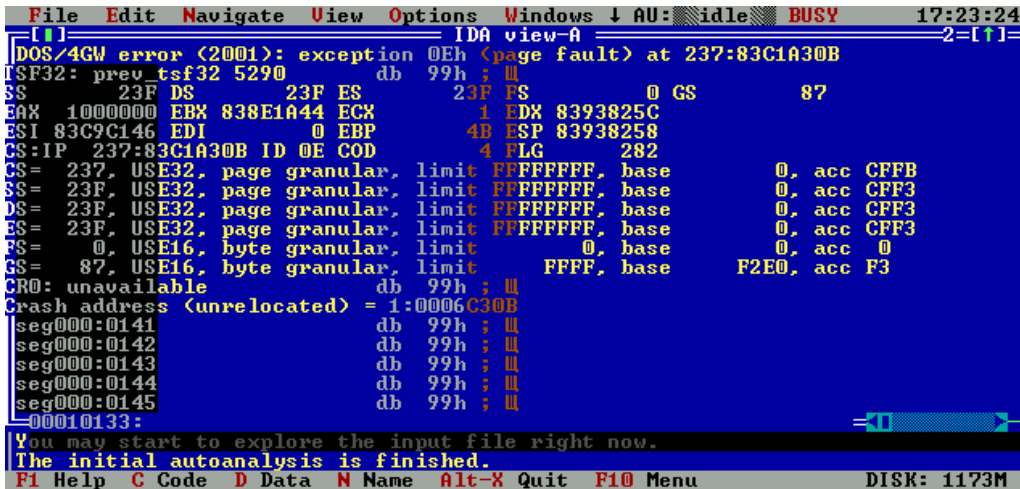


Рис.6. Ошибка при обращении к ячейке вне выделенного блока

Несмотря на то, что обе функции принимают в качестве аргумента линейный 32-битный адрес, никто не запрещает пользоваться макросом `MK_FP(segment,offset)`, который является просто другой формой записи следующего выражения:

$$ea = \text{segment} * 10h + \text{offset}$$

Под Win32 используется действительно 32-разрядная линейная (flat) модель памяти, поэтому там макрос `MK_FP` теряет смысл, однако может использоваться для доступа к памяти BIOS, которая одинаково хорошо «видна» как из под DOS, так и из Win32.

Попытка обращения к памяти, занятой IDA возможна, но не гарантирована. Например, блоки, выделяемые DPMI, могут быть разбросаны по физической памяти и об их расположении можно только гадать.

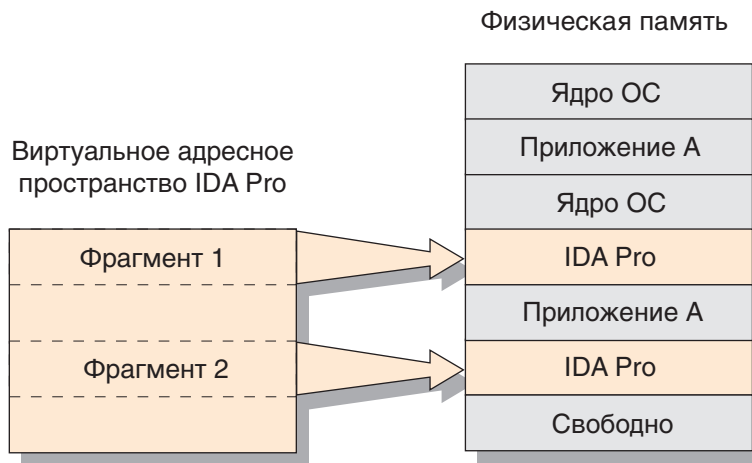


Рис. 7. Виртуальная и физическая память

На этом мы завершаем рассмотрение этой небольшой особенности «виртуальной машины» перейдем к организации памяти, непосредственно относящейся к IDA.

Виртуальная память была введена для обеспечения возможности манипулирования дизассемблируемым файлом. В первом приближении она идентична образу исследуемого файла, т.е. IDA эмулирует работу загрузчика операционной системы и можно работать с дизассемблируемым файлом точно так, как мы бы работали с ним, например, в отладчике.

Для доступа к любому байту необходимо знать его сегмент и смещение. Если исследуемый файл использует линейную модель памяти и ничего не знает о сегментах (как, например, в случае с .com-файлами), то в этом случае необходимо создать один сегмент, куда и скопировать файл. В противном случае IDA не сможет с ним работать. Будут доступны лишь низкоуровневые операции, такие, как запись/чтение ячейки, двоичный поиск и т.п. Попытка дизассемблировать фрагмент кода не даст ничего кроме следующего сообщения:



Рис. 8. Сообщение об отсутствии сегмента

т.е. мы не можем дизассемблировать код, не принадлежащий ни одному из сегментов. Уточним: не можем дизассемблировать штатными средствами IDA. Но никто нам не запрещает дизассемблировать (или обработать любым другим образом) его самостоятельно, ведь низкоуровневый доступ к памяти сохраняется!

Эта возможность может пригодиться, например, для ручной загрузки файлов, не поддерживаемых IDA, таких как, самозагружаемые модули или зашифрованные (запакованные) файлы. В этом случае файл необходимо загрузить как бинарный, распаковать специально написанным для этого скриптом, и только потом скопировать в созданный сегмент для дальнейшего анализа.

Рассмотрим достаточно нетривиальный пример использования IDA для расшифровки текста, каждый символ которого модифицирован путем выполнения над его ASCII-значением логической операции «исключающее ИЛИ» с некоторой константой. (Это, конечно, простейший пример автоматического криптоанализа, но, тем не менее, он достаточно любопытен).

```

auto a, s1, s2, temp, p;
s1=0;
for (a=0; a<0x100; a++)
{
    s2=0;
    for (p=0x10000; p<0x1024E; p++)
        if (Byte(p)==a) s2++;
    if (s2>s1)
    {
        s1 = s2;
        temp = a;
    }
}
temp=temp ^ 0x20;
for (p=0x10000; p<0x1024E; p++)
{
    a=Byte(p);
    a=a^temp;
    PatchByte(p, a);
}
for (p=0x10000; p<0x1024E; p++)
    Message(Byte(p));

```

Действительно, это выходит за рамки проблемы дизассемблирования, но именно в том и заключается уникальность IDA, что ее «виртуальную машину» можно использовать для неограниченно широкого круга задач. В контексте данного примера IDA скорее даже не дизассемблер, а интерактивная среда программирования. Конечно, данный скрипт, разве что дополненный процедурой загрузки файла, неплохо бы работал, скомпилированный любым Си-компилятором. Поэтому предположим, что он является лишь подзадачей, встретившийся в процессе дизассемблирования некоторого файла.

Чтение очередного байта осуществляется такой функцией:

```
long Byte (long ea)
```

где «ea» — линейный 32-разрядный адрес виртуальной памяти. Большинство функций IDA в качестве аргумента принимает именно линейный адрес. Кроме побайтового чтения, IDA может так же манипулировать словами и двойными словами, для чего и служат две следующие функции:

```
long Word (long ea);  
long Dword (long ea);
```

которые, соответственно, читают два и четыре байта, указываемых «ea». Однако, это не более чем «надстройки» над функцией *Byte()*. Задумаемся, что произойдет, если прочесть слово (двойное слово), начало которого находится на границе сегмента памяти, адресуемого 32 разрядами? Произойдет исключение или нет? Эксперимент показывает, что нет. При попытке чтения области памяти, выходящей за 4 гигабайта будут использованы только младшие 32 разряда адреса и функция вернет значения, находящиеся в виртуальной памяти по этому «усеченному» адресу. Сказанное выше подтверждает следующий пример:

```
if (Byte(0)==(Word(-1)>>8)) Message ("Hello!\n");
```

Такое поведение IDA не вызывает проблем и является не более чем любопытной особенностью внутренней организации виртуальной памяти IDA. На самом низком уровне IDA манипулирует исключительно байтами.

Изменить ячейку виртуальной памяти можно с помощью следующих функций:

```
void PatchByte (long ea, long value);  
void PatchWord (long ea, long value);  
void PatchDword (long ea, long value);
```

где «value» — собственно записываемое значение. А теперь одно важное замечание — виртуальная память доступна для чтения/модификации, только если принадлежит к какому-нибудь сегменту или загруженному двоичному файлу. В любом другом случае функция *Byte()* возвратит -1 (код ошибки). Однако, при этом возникает неоднозначность — то ли действительно адрес не существует, то ли просто ячейка содержит такое значение. Уточнить ситуацию помогают внутренние флаги IDA, которые можно прочесть с помощью функции *GetFlags(long ea)*. При этом в случае ошибки функция возвратит нулевое значение.

В связи с вышесказанным в IDA нет функции типа *alloc()*, и посредством скриптов нельзя выделить блок памяти под свои нужды иначе, как создав сегмент или загрузив файл. Этим и объясняется «медлительность» приведенного выше скрипта. Отсортировать за один проход массив чисел можно, только если использовать блок памяти соответствующих размеров (позже будет показано, как это сделать).

Итак, на самом низком уровне иерархии IDA оперирует с линейной 32-разрядной моделью виртуальной памяти. Однако, реально доступный размер много меньше 4 Гб и по умолчанию составляет 128 Мб. Почему так происходит? Дело в том, что на физи-

ческом уровне IDA манипулирует не 32-х, а 16-разрядными указателями. Но как же тогда она может адресовать более одного мегабайта? Очень просто — используя страничную адресацию. Т.е. указатель ссылается не на конкретный байт, а на фрагмент памяти (страницу), таким образом доступная память вычисляется как:

$$10000h * PageSize$$

и определяется размером страницы. С увеличением размера страниц увеличивается не только доступная память, но и гранулированность, а отсюда потери памяти и замедление работы за счет времени, затраченного на их подкачку.

Размер определяется в файле `ida.cfg` ключем «VPAGESIZE». По умолчанию он равен 8192 (впрочем, в различных версиях значение может меняться). Следовательно, IDA может адресовать $10000h * 8192 = 536870912$ байт физической памяти. А виртуальной? Что бы ответить на этот вопрос нужно понять ее устройство. На самом деле на каждую ячейку виртуальной памяти расходуется четыре байта физической. Один байт, хранит само значение, содержащиеся в ячейке, а остальные три расходуются на описатели или так называемые флаги. За счет этого IDA может отличать код от данных, а так же тип этих данных и т.д.

Структура флагов хорошо документирована и описана в файле `idc.idc`. Получить внутренне представление любой ячейки можно в любой момент, нажав клавишу «F» (View/Internal flags) или с помощью функции `GetFlags(long ea)`. А непосредственное значение можно увидеть и изменить (!), заглянув в сохраненную базу IDA. Давайте сделаем это на примере `ida__0x2.bin`. Необходимо только помнить, что формат базы — вещь не постоянная, и никто не гарантирует его сохранность в последующих версиях (даже более того, могу вас заверить, что он обязательно рано или поздно будет изменен).

Для начала рассмотрим фрагмент изучаемый файл в шестнадцатеричном виде:

```
00000000: 57 57 57 E6 57 96 D7 DB D9 D4 D9 57 DA D7 90 D7 ;
```

и попробуем найти «57» в созданной IDA базе:

```
00005020: 00 00 00 00 57 01 00 00 57 01 00 00 57 01 00 00 ;
00005030: E6 01 00 00 57 01 00 00 96 01 00 00 D7 01 00 00 ;
00005040: DB 01 00 00 D9 01 00 00 D4 01 00 00 D9 01 00 00 ;
```

Как нетрудно заметить, второй фрагмент есть ни что иное, как физическое представление загруженного в виртуальную память файла. Хорошо видно, что каждый байт исходного текста фактически занимает четыре байта памяти.

Попробуем изменить в шестнадцатеричном редакторе выделенный байт на `B1h` и загрузим обновленную базу в дизассемблер:

```
00005020: 00 00 00 00 57 B1 00 00 57 01 00 00 57 01 00 00 ;
```

При этом создастся новое имя (или, другими словами, метка), которое отобразит IDA:

```
0:00010000 unk_00010000 db 57h
```

Что в этом удивительного? Попробуйте создать еще одно имя, нажав клавишу «N» и получите отказ, мотивированный отсутствием сегмента. Следовательно, низкоуровневая работа с базой может дать результат не осуществимый ни интерактивным взаимодействием, ни скриптами.

С одной стороны — не очень корректно использовать в работе подобные недокументированные возможности, но иногда это единственный выход из ситуации. При этом файл `*.id1`, создаваемый при работе, представляет собой точную копию виртуальной памяти, и, маловероятно, чтобы он подвергся изменению, по крайней мере, в ближайших версиях.

Итак, доступный объем виртуальной памяти в четыре раза меньше физической, и полная формула для расчета будет выглядеть следующим образом:

$$10000h * VPAGESIZE / 4 = 4000h * VPAGESIZE$$

Следует заметить, что 128 Мб, выделяемых по умолчанию, это очень и очень много и можно уменьшить размер страниц для повышения скорости работы. Нужно только обязательно помнить, что этот размер должен быть кратен двум. Ближайшее подходящее значение — 4096 предоставит в наше распоряжение 64 мегабайта, что более чем достаточно для большинства задач. Скорость при этом (особенно на компьютерах с небольшим объемом RAM) заметно возрастет за счет того, что база большей частью будет расположена в оперативной памяти, а не на жестком диске в файле подкачки. Однако, необходимо помнить, что, если объем затребованной виртуальной памяти превысит существующий, то IDA может повести себя непредсказуемо, в лучшем случае отказавшись работать без вразумительных мотивировок.

На скорость работы также влияет размер отводимых буферов физической памяти или другими словами окно подкачки. Дело в том, что IDA держит часть базы в памяти, а часть на диске. Чем больше памяти отводится под буфер окна, тем меньше времени уходит на подкачку недостающих страниц с диска, и тем быстрее работает IDA. Однако, начиная с некоторого предела, наблюдается обратный эффект, обусловленный недостатком физической памяти. При этом операционная система сама начинает подкачку, высвобождая требуемую программе память. В результате двойного процесса подкачки работа всей системы резко замедляется. Поэтому не рекомендуется устанавливать размер окна более четверти свободной физической памяти.

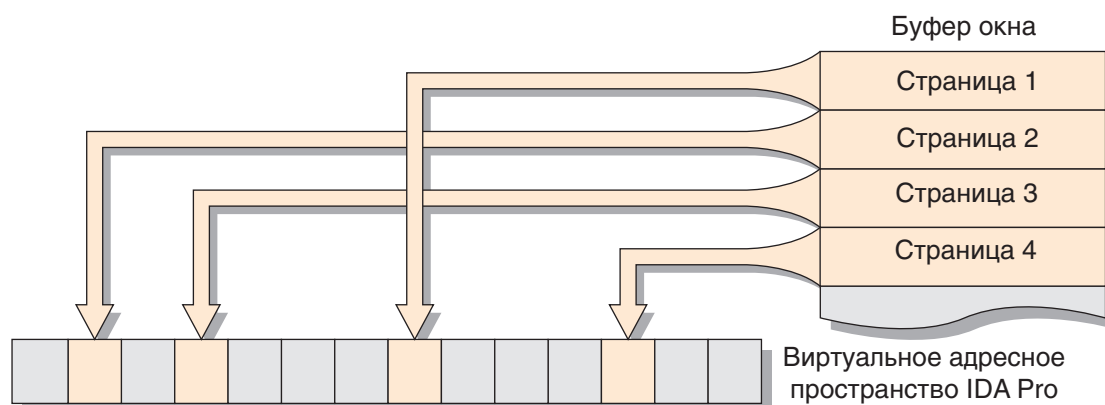


Рис. 9. Копирование страниц из буфера окна в виртуальную память

Логика работы виртуальной памяти следующая — при обращении к произвольной ячейке IDA сначала проверяет, загружена ли требуемая страница в буфер; если нет, то считывает целиком страницу в память. При последующих запросах на запись/чтение обращение будет идти к странице, расположенной в быстрой физической памяти. Однако, рано или поздно наступает такой момент, что буфер полностью заполнен, а требуется загрузить еще одну страницу. В этом случае IDA ищет немодифицированную страницу, к которой дольше всего не обращались, и замещает ее вновь загружаемой. Если таковых не оказывается, то IDA выгружает на диск любую модифицированную страницу, освобождая место для новой.

При этом, периодически, модифицированные страницы записываются на диск во избежание потери данных, например, при сбое питания. Задать желаемый интервал можно с помощью ключа «AUTOSAVE» в файле idatui.cfg. По умолчанию модифицированные буферы выгружаются на диск после 100 действий пользователя, либо по истечении 5 минут. При этом на экран выдается пояснение «Flushing buffers, please wait...». Понятно, что при увеличении размера страниц возрастает время их загрузки

ки/выгрузки на диск, а также падает максимально доступное число страниц в буфере. С другой стороны увеличивается вероятность того, что очередной требуемый байт уже окажется загруженным в оперативную память и может быть немедленно прочитан. Вобщем, виртуальная память IDA аналогична виртуальной памяти Windows и представляет собой классический пример кэша, поэтому желающие узнать больше могут обратиться к соответствующей литературе.

Если размер окна не указать принудительно, то IDA будет пытаться вычислить оптимальный, используя следующий алгоритм: на этапе загрузки файла определяется требуемый объем памяти для его загрузки. Как уже упоминалось выше, виртуальной памяти потребуется вчетверо больше, поэтому полученное значение умножается на четыре. Дальнейший выбор иллюстрируется в табл. 1, где ASPACE и есть затребованный размер виртуальной памяти.

Таблица 1. Соответствие между размером буфера окна и требуемой виртуальной памятью

требуется	размер окна
< 1 Мб	ASPACE
< 4 Мб	1 Мб
< 10 Мб	ASPACE/4
< 40 Мб	4 Мб
> 40 Мб	ASPACE/10

Обычно нет никаких причин вычислять и устанавливать это значение вручную, однако при дизассемблировании больших (свыше 5-7 мегабайт) файлов можно значительно (иногда вдвое или втрое) ускорить процесс, если оптимизировать виртуальную память, исходя из имеющийся физической. В общем случае необходимо максимально уменьшить размер страниц и полностью использовать доступную физическую память.

Размер буфера окна устанавливается в файле `ida.cfg` колючем «VPAGES» в страницах. О том, сколько страниц было выделено можно узнать при загрузке.

Например, для `ida__0x2.bin`:

```

bytes  pages size description
-----
262144  32 8192 allocating memory for b-tree...
65536   16 4096 allocating memory for virtual array...
65536   64 1024 allocating memory for name pointers...
-----

```

«allocating memory for b-tree...» — это и есть буфер окна. Для нашего файла IDA выделила 32 страниц по 8192 байт каждая. При этом IDA не расходует память (по крайней мере дисковую) для неиспользуемых страниц. Размер файла `ida__0x2.id1` в точности равен 8192 байт, т.е. одной странице. Об оперативной памяти этого сказать нельзя — независимо от того используются остальные страницы или нет, IDA при инициализации полностью «зануляет» весь буфер окна, что приводит к неэкономному расходу памяти.

При дизассемблировании пользователь добавляет имена, перекрестные ссылки, определяет функции — все это, естественно, требует памяти. IDA использует еще одну плоскость, получившую название «DATABASE_MEMORY», скрытую от пользо-

вателя и и используемую как раз для «рабочих» нужд дизассемблера и пользователя. Из командного языка доступна только ее косвенная адресация. Можно выделить блок памяти для организации массива, но при этом совершенно неизвестно, по какому линейному адресу он расположен — вся работа ведется исключительно через API IDA.

Поэтому пользователю совершенно ни к чему знать структуру этой плоскости, а автору ее документировать. Тем более, что в разных версиях IDA ее реализация различна. Любопытные читатели могут попробовать в той или иной мере выяснить ее самостоятельно, изучая файл *.idb. Может ли это иметь какой-то практический интерес? Иногда, да. Так, например, при удалении комментариев IDA физически не вычищает занимаемую ими память, и можно попробовать их восстановить! Иногда это может спасти немного времени, нервов и Вашего труда.

То же можно сказать и относительно массивов. Любопытная особенность IDA заключается в том, что создаваемые массивы она сохраняет в базе данных вплоть до момента их удаления. Это значительно упрощает сохранение результатов работы Ваших скриптов. Если же массив был ошибочно удален, то есть некоторый шанс его восстановить. (Подробнее работа с массивами будет рассмотрена в главе «Организация Массивов».)

Разумеется, что при таком подходе теряется часть памяти. Очистить ее можно, если при выходе установить флажок «collect garbage». Однако, это может потребовать некоторого времени, особенно на больших базах.

Размер требуемой под нужды дизассемблера и пользователя памяти предсказать вряд ли возможно — кто знает точно, сколько потребуется создать имен, комментариев, перекрестных ссылок? Поэтому IDA не в состоянии угадать оптимальный размер буфера окна и использует тот же самый алгоритм, что и для виртуальной памяти. Обычно проблем не возникает, но в некоторых случаях потери скорости могут быть довольно ощутимыми, и тогда приходится задавать данное значение вручную. Для этого необходимо изменить ключ «DATABASE_MEMORY» в файле ida.cfg. Обратите внимание на то, что значение задается уже не в страницах, а в байтах. Однако, ввиду страничной организации памяти, IDA выделит не строго требуемый объем, а округлит его до числа используемых страниц.

Однако изменять конфигурационный файл для каждого отдельного случая не очень удобно — лучше задавать эти значения из командной строки. Для этого используются два ключа: «-d» и «-D». Отличаются они тем, что «-d» выполняется при первом проходе конфигурационного файла (на этапе его загрузки), а второй — при следующем (определение процессора). Инициализация виртуальной памяти выполняется уже при первом проходе. Все последующие попытки изменения установленного размера страниц не дают никакого эффекта. При этом задать требуемое значение можно только один раз на стадии формирования базы данных IDA. В дальнейшем его уже нельзя изменить будет. Поэтому, прежде чем изменить значение по умолчанию, лучше все хорошо продумать, чем потом проиграть в производительности и замедлить всю свою работу.

Впрочем, в последних версиях IDA все же существует выход из этой ситуации. Необходимо сохранить базу данных в виде IDC файла, после чего можно создать новую с другими параметрами виртуальной памяти.

Еще один буфер создается для указателей имен. Он не является жизненно необходимым для системы, поскольку все имена сохранены в базе данных, но значительно ускоряет работу. Рассмотрим файл *.nam, чтобы изучить его структуру. Загрузим, например, ida__0x1.idb и заглянем в файл ida__0x1.nam:

```
00000400: 00 01 01 00 0C 01 01 00 14 01 01 00 16 01 01 00
00000410: 1E 01 01 00 0C 01 02 00 00 00 00 00 00 00 00 00
```

и сравним эти значения с:

```
seg000:010C Repeat:
seg000:0114 LengthCryptCode:
```

Нетрудно видеть, что выделенные байты представляют собой линейный адрес созданных имен! Эту же информацию можно найти и в `ida__0x1.id1` файле, но с гораздо большими трудозатратами, ввиду сложности его структуры. Файл `ida__0x1.nam` есть ни что иное, как массив указателей на имена, входящие в «Name List», в противоположность «Dummy Name». Его удаление не влечет за собой никаких последствий, разве что, некоторой потери времени на подгрузку этой информации из основной базы (однако, следует помнить, что это не обязательно будет сохранено в последующих версиях. Возможно, что IDA даже откажется загружать такую базу).

Размер буфера указателей имен не так критичен, как все остальное. По умолчанию резервируется 64 страницы размером в 1024 байта каждая. Таким образом, в него помещается 16384 указателей имен (каждый указатель занимает 4 байта). Редкая программа содержит так много имен! Максимально доступное число имен еще больше. При размере страниц в 1024 байта можно создать до 16 777 216 имен! Маловероятно, что этот предел кому-нибудь удастся превысить. Поэтому рекомендуется уменьшить размер страниц хотя бы до 512 байт, что можно сделать, изменив ключ «NPAGESIZE». С другой стороны, программы, откомпилированные в Delphi иногда содержат сотни тысяч имен, и это надо учитывать при установке значений.

Поднимемся теперь на уровень выше, и рассмотрим два новых понятия — *сегмент* и *селектор*. Сегментом называется непрерывная область памяти, имеющая начало и конец. Это определение не претендует на полноту, которая невозможна в кросс-платформенной системе IDA. Для разных процессоров понятия «сегмента» могут немного отличаться, но в любом случае остаются «прозрачными», потому что IDA везде манипулирует не сегментными, а линейными адресами. Сегменты не более, чем искусственная надстройка над памятью IDA.

Перевести сегментный адрес в линейный очень просто:

```
ea = segment*0x10+offset
```

Т.е., в первом приближении для доступа к произвольно взятому байту нужно знать сегмент, в котором он расположен, и смещение.

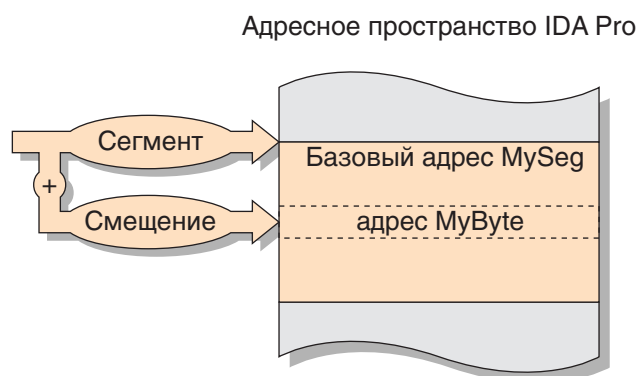


Рис. 10. Преобразование сегментного адреса в линейный

При этом линейный адрес «MySeg» называется *базовым* и положение каждого байта в сегменте определяется следующей формулой (приведенной в контекстной помощи IDA):

```
VirtualAddress = LinearAddress - (SegmentBase << 4);
```

Возможно, для большей ясности, следовало бы переименовать «VirtualAddress» в «SegmentAddress», т.е. адрес в сегменте, в отличие от линейного. Попробуем привести это выражение к следующему виду:

$$\text{VirtualAddress} = \text{LinearAddress} - (\text{SegmentBase} \ll 4) = \text{segment} * 10\text{h} + \text{offset} - (\text{SegmentBase} \ll 4) = \text{offset}.$$

Таким образом, положение каждого байта в сегменте однозначно определяется одним лишь смещением. Сам же сегмент однозначно определяется базовым, начальным и конечным адресами. Чаще всего начальный адрес нулевой, но в некоторых случаях, как, например у .com-файла, он равен 100h.

Однако, еще раз обратите внимание, что физически никаких сегментов IDA не создает и требует линейного адреса, а не смещения. И пусть Вас не смущает возможная форма записи

```
Byte([0x1000, 0x10]);
```

На самом деле функция *Byte()* принимает на вход только линейный адрес, а [segment, offset] — не более, чем оператор, возвращающий линейный адрес. Рассмотрим следующий пример:

```
Message(«%x \n», [0x1000, 0x10]);
```

При его выполнении на экран будет выведено ни что иное, как «0x10010». Такая ситуация сложилась по причине отсутствия в IDA функций, непосредственно работающих с сегментными адресами. С другой стороны, для пользователя линейные адреса вообще не видимы, он работает исключительно с сегментами и смещениями и может совершенно не знать, как они устроены «изнутри».

С точки зрения пользователя программа ida__0x3.exe будет выглядеть так:

```
seg000:0000 start      proc near
seg000:0000             mov     ax, seg dseg
seg000:0003             mov     ds, ax
seg000:0005             assume ds:dseg
seg000:0005             mov     ah, 9
seg000:0007             mov     dx, offset aHelloSailor
seg000:000A             int     21h
seg000:000C             mov     ah, 4Ch
seg000:000E             int     21h
seg000:000E start      endp
seg000:000E seg000     ends
dseg:0000 dseg         segment para public 'DATA' use16
dseg:0000 aHelloSailor db 'Hello,Sailor!', 0Dh, 0Ah, '$'
dseg:0000 dseg         ends
dseg:0000             end start
```

Он будет манипулировать сегментами и смещениями, при этом, может быть, так и не узнав о существовании виртуальной памяти и линейных адресов. И такое «может быть» встречается в жизни достаточно часто: практика показывает, что большинство пользователей IDA используют ее большей частью в интерактивном (или даже полностью автоматическом) режиме, и далеко не каждый из них, хотя бы однажды, воспользуется встроенным языком.

Попробуем написать скрипт, выводящий байт, находящийся под курсором. На первый взгляд, сперва требуется определить смещение и текущий сегмент, получить на их основе линейный адрес..., но мы забыли, что сегментная адресация реально не существует, и IDA позволяет определить непосредственный линейный адрес находя-

щегося под курсором байта. Для этого существует функция `ScreenEA()`, ну а сам скрипт может выглядеть следующим образом:

```
Message(«%x \n», Byte(ScreenEA()));
```

Гораздо сложнее по линейному адресу узнать, к какому сегменту он принадлежит. Но обычно это и не требуется. Скрипты редко манипулируют сегментами, а все больше линейными адресами. При этом у сегментов есть одно серьезное ограничение. Базовый адрес выражается 16-разрядным значением, и, следовательно, не может превышать `10000h`. Фактически это означает, что в нашем распоряжении не более одного мегабайта доступной памяти, и за его пределами ни один сегмент создать не удастся! До некоторого времени такое ограничение не было существенным, однако сегодня большинству программ этих мегабайт требуется не один и не два.

Выход был найден в использовании *селекторов*. В первом приближении селектор — это не более чем 32-разрядный базовый адрес. Теперь 16-разрядный базовый адрес ссылается на конкретный селектор в таблице, который в свою очередь содержит 32-разрядную базу сегмента.

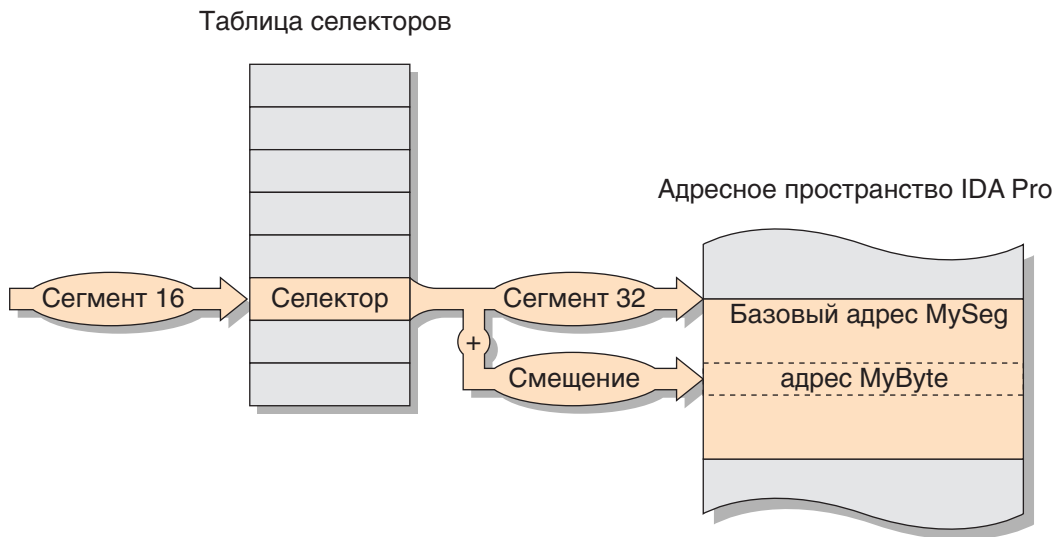


Рис. 11. Адресация с использованием селекторов

При этом данная архитектурная особенность виртуальной памяти скрыта от пользователя. При попытке создать сегмент за пределами первого мегабайта, IDA автоматически использует для его адресации селектор. Продемонстрируем это на следующем примере. Дадим команду

```
SegCreate(0x200000, 0x200200, 0x20000, 0, 0, 0);
```

а затем вызовем окно просмотра сегментов «View/Segments».

Name	Start	End	Align	Base	Type	Cls	32es	ss	ds	fs	gs	
seg000	00000000	00000010	byte	1000	pub CODE	N	FFFF	FFFF	1001	FFFF	FFFF	0
dseg	00000000	00000010	para	1001	pub DATA	N	FFFF	FFFF	1001	FFFF	FFFF	0
seg002	00000000	00000200	at	0001	pri	N	FFFF	FFFF	FFFF	FFFF	FFFF	0

Рис. 12. Параметры вновь созданного сегмента

«0001» — это ничто иное, как автоматически созданный селектор. Таким образом, работа с селекторами прозрачна не только на уровне пользователя, но и API IDA. Но как же в этом случае преобразовать селектор в линейный адрес? Очевидно, для этого необходимо просмотреть таблицу селекторов «View\Selectors» или воспользоваться функцией `AskSelector(1)`. Вызовем консоль и наберем следующую команду:

```
Message(«%x \n», AskSelector(1));
```

которая выдаст базовый адрес сегмента.



Sel	Value
0001	00020000

Рис. 13. Таблица селекторов

Попробуем теперь создать сегмент с базовым адресом 1. Нас постигнет глубокое разочарование. Несмотря на все наши усилия IDA откажется это сделать. Дело в том, что поддержка селекторов, хотя и позволила прорваться за пределы первого мегабайта, но и повлекла за собой серьезное ограничение — невозможно создать сегмент с базовым адресом уже существующего селектора. На этом уровне IDA не отличает сегментов от селекторов. Это бремя ложиться на программиста, который, получив базовый адрес, должен проверить, не соответствует ли ему какой-нибудь селектор? И, если да, то получить его значение.

Как избежать совпадения селекторов и сегментов? Очень просто — сначала определить все сегменты, после чего выделить оставшиеся адреса под селекторы. Селектору абсолютно все равно, какой индекс он имеет, он от этого хуже работать не станет. Единственное, что индексы обрезаются до 16 разрядов, хотя и представляют собой длинное целое. Однако, очень трудно представить программу, которой понадобилось бы более 65535 селекторов одновременно.

ЗАГРУЗКА ФАЙЛА. ОПЦИИ КОМАНДНОЙ СТРОКИ

Любопытной особенностью IDA является то, что она поддерживает формат .zip и позволяет работать непосредственно с упакованными файлами. Если имя файла не указано в командной строке, то IDA сама запросит его при запуске. В процессе дизассемблирования IDA не работает с выбранным файлом (и его безболезненно можно удалить), а создает набор файлов .id* — собственной базы данных, с которой и взаимодействует. Поэтому при повторных загрузках файла грузиться не он сам, а созданная ранее база. Понимание этого крайне важно, т.к. при этом не отслеживаются никакие модификации файла, как уже упоминалось исследуемый файл однократно загружается в базу и в дальнейшем не принимает никакого участия в работе.

Для особых ситуаций предусмотрен ключ командной строки «-с», удаляющий перед запуском ранее созданные дизассемблером базы. Разумеется вместе со всеми заботливо созданными вами комментариями, именами, метками..., уничтожая всю проделанную работу. Конечно, это не может служить решением проблемы — необходимо перегрузить только модифицированный фрагмент, а не уничтожать всю базу! И хотя штатно такая возможность не была предусмотрена, встроенный язык IDA позволяет написать подобный загрузчик самостоятельно.

При первой же загрузке файла появится следующий диалог:

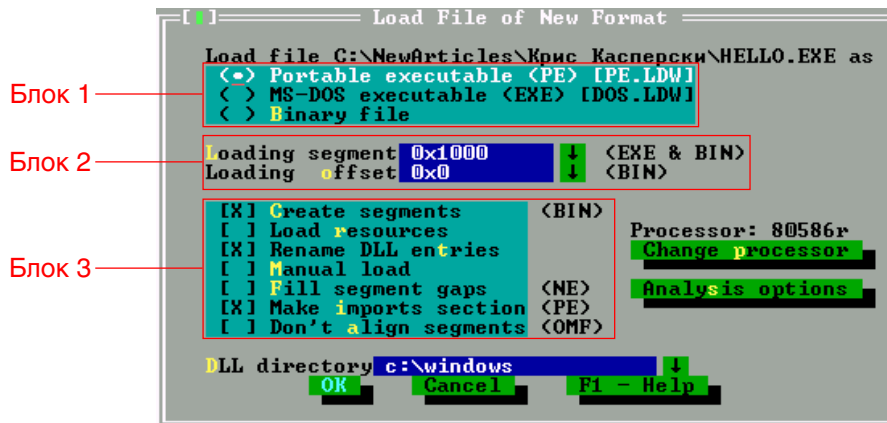


Рис. 14. Диалог загрузки файла

Блок 1. В большинстве случаев IDA автоматически определяет тип загружаемого файла, но все же оставляет конечное решение за Вами. Это действительно бывает полезно во многих случаях — Вы всегда можете загрузить файл как бинарный и работать с его структурами вручную. Например, текущая версия IDA не поддерживает самозагружающиеся модули и завершает работу при попытке их загрузки.

Однако, хоть и редко, но с такими файлами исследователь рано или поздно сталкивается. Единственный выход загружать файл как бинарный и дизассемблировать его с помощью собственных скриптов. Не очень вдохновляющая перспектива, конечно, но можно утешить себя тем, что другие дизассемблеры не умеют и этого.

Остается только надеяться, что рано или поздно разработчики разрешат эту проблему, но и сегодня IDA обгоняет остальные дизассемблеры числом и качеством поддерживаемых форматов файлов (см. табл. 2).

В действительности этот список не исчерпывает возможности IDA, гибкая архитектура которой позволяет работать с произвольными типами файлов, кроме того в каждой новой версии число поддерживаемых форматов все увеличивается.

Блок 2. Сегмент и смещение загрузки на первом этапе освоения IDA лучше не изменять. Смещение актуально только для двоичных (BIN) файлов. Например, для дизассемблирования копии загрузочного сектора диска (MBR) — его необходимо загрузить со смещением 7C00h, иначе все смещения будут указывать в «никуда». Для

типизированных файлов IDA игнорирует установленное смещение загрузки, извлекая эту информацию из соответствующих полей заголовка.

Базовый адрес сегмента на самом деле не имеет никакого отношения к исследуемому файлу, лишь к его загрузке в виртуальную память IDA. Это значение необходимо только при написании собственных скриптов, а в остальных случаях оно никак не отразится на процессе дизассемблирования.

Таблица 2. Типы файлов, обрабатываемых IDA Pro

EXE	Исполняемый файл	MS-DOS
COM	Исполняемый файл	MS-DOS, CP/M
SYS	Устанавливаемый драйвер	MS-DOS
NE	New Executable Format	Windows 3.x, OS/2
LX	Linear Executable Format	OS/2 2.x, OS/2 Warp
LE	Linear Executable Format	Windows VxD
PE	Portable Executable Format	Win32
OMF	Intel Object Module Format	MS-DOS, Windows 3.x, OS/2
LIB	Library of Object Files	MS-DOS, Windows 3.x, OS/2
AR	Library of Object Files	UNIX, Win32
COFF	Common Object File Format	UNIX
NLM	Novell Netware Loadable Modules	
ZIP	Archive files	
JAVA	Java classes	

Блок 3. Здесь расположены сразу несколько опций:

Create segments

Создавать или не создавать сегменты для двоичных файлов. По умолчанию всегда создается по крайней мере один сегмент. Эту опцию можно выключить, когда Вы хотите расставить сегменты по своему усмотрению (например, при анализе самозагружаемых модулей). Но создавать хотя бы один сегмент необходимо в любом случае. Внутренняя архитектура IDA такова, что большинство команд работает только с сегментами. Попытка дизассемблировать фрагмент, не принадлежащий ни к одному из сегментов, будет отвергнута с выдачей следующего сообщения:



Рис. 15. Сообщение об отсутствии сегмента

Однако, другие команды, такие, например, как чтение/запись памяти будут успешно работать и могут быть использованы, например, в скрипте расшифровки файла. В этом случае, разумеется, в создании сегмента никакой необходимости нет.

Load resources

Указывает на необходимость загрузки ресурсов. Актуально только для PE и NE файлов. По умолчанию выключено, однако довольно часто в ресурсах расположены

текстовые строки или даже некоторые данные, тогда ресурсы лучше загрузить. В остальных же случаях это только лишний расход времени и памяти.

Rename DLL entries

IDA умеет распознавать большинство библиотечных функций (подробнее об этой уникальной возможности см. главу, посвященную технологии FLIRT). При этом она может заменять функции, импортируемые по порядковому номеру, их непосредственными именами. Если же по каким-то причинам для Вас это не приемлемо, то данную опцию необходимо отключить. При этом IDA добавит символьные имена в повторяемые комментарии. Сравните:

```
; Imports from MFC42.DLL
?DoMessageBox@CWinApp@UAHPBDII@Z    dd    ?
?SaveAllModified@CWinApp@UAHXZ      dd    ?
?InitApplication@CWinApp@UAHXZ      dd    ?
```

и

```
; Imports from MFC42.DLL
MFC42_2512    dd    ?                ; DATA XREF: j_MFC42_251234r
; ?DoMessageBox@CWinApp@UAHPBDII@Z:
MFC42_5731    dd    ?                ; DATA XREF: j_MFC42_573134r
; ?SaveAllModified@CWinApp@UAHXZ:
MFC42_3922    dd    ?                ; DATA XREF: j_MFC42_392234r
; ?InitApplication@CWinApp@UAHXZ:
MFC42_1089    dd    ?                ; DATA XREF: j_MFC42_108934r
```

Разумеется, гораздо удобнее, когда IDA заменяет ординалы осмысленными именами, и найдется немного ситуаций, в которых эту опцию приходится отключать.

Manual load

«Ручная» загрузка некоторых типов файлов. В основном используется для NE, LX и LE форматов. При этом пользователь получает возможность для каждого из объектов файла задать селектор и базовый адрес загрузки с помощью следующего диалога:

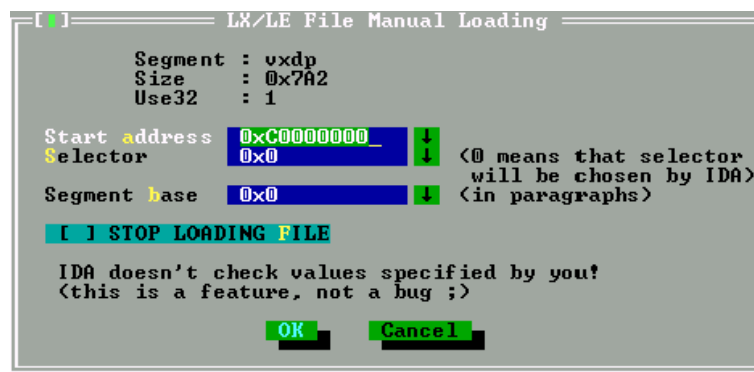


Рис. 16. Диалог задания параметров объекта для загружаемого LX/LE файла

«Start Address» указывает, по какому адресу будет расположен загружаемый объект. Это значение вычисляется дизассемблером автоматически, и обычно нет причин менять его. Подробнее об этом будет рассказано в главе, посвященной анализу .vxd файлов.

Селектор, разумеется, имеет отношение не к исследуемому файлу, а непосредственно к дизассемблеру, а точнее к организации внутренней (виртуальной) памяти IDA. Сейчас мы не будем заострять на этом свое внимание, отметим только, что селектор

необходим для доступа к сегментам, написанных Вами скриптов, а в остальных случаях его значение будет пользователю не нужно.

Базовый адрес связан с виртуальным адресом следующей формулой:

$$\text{VirtualAddress} = \text{LinearAddress} - (\text{SegmentBase} \ll 4);$$

Т.е. одному и тому же виртуальному адресу могут соответствовать различные пары SegmentBase:LinearAddress. Подробности см. в главе, посвященной организации виртуальной памяти IDA.

«STOP LOADING FILE», разумеется, означает прекращение загрузки остальных объектов (сегментов) файла. Может быть использовано для экономии времени — если остальные объекты/секции Вас не интересуют, то к чему тратить время и ресурсы на их загрузку?

PE файлы имеют более простую организацию, в первом приближении представляя собой просто образ памяти процесса. Ручная загрузка сводится только к произвольному выбору загружаемых секций. Ни базовый адрес, ни адрес загрузки изменить невозможно.



Рис. 17. Запрос на загрузку сегмента PE файла

Make imports section

По умолчанию IDA преобразует секцию импорта `.idata` PE файлов в набор директив «extern» и усекает ее. Обычно это работает нормально, но никто не гарантирует, что в секции импорта не окажутся размещенными некоторые данные. Так, например, поступают некоторые вирусы, размещая свое тело в таблице адресов. Разумеется, при этом IDA их «не увидит». В таких случаях данную опцию следует отключить.