

Полная перекомпиляция программы write.exe или тестируем демо-версию IdaPro 5.6

by Erfaren

<http://erfaren.narod.ru>

erfaren@rambler.ru

Введение

В настоящее время существует широко известный и достаточно высококачественный интеллектуальный дизассемблер **IdaPro** от выпускника мехмата МГУ **Ильфака Гильфанова (Ifak Guilfanov)**. Всегда интересно знать, насколько хорош этот инструмент. Например, как много и что именно нужно изменить в декомпилированном коде, выдаваемом **IdaPro**, скажем для простейшей **GUI** программы, которую можно найти в папке **Windows**, чтобы скомпилированная версия этой программы заработала снова, практически ничем не отличаясь от исходной программы? Т.е., чем меньше телодвижений нужно делать, чтобы перекомпилировать дизассемблерный файл в работающее приложение, тем, очевидно, более качественен инструмент нашего **Ильфака** 😊. Ну и конечно эти телодвижения не должны быть слишком уж изощренными, по крайней мере для небольших программ.

Естественно, тестировать можно любую версию дизассемблера. Но поскольку, на данный момент, на сайте разработчика <http://hex-rays.com/idapro/idadown.htm> доступна уже демо-версия **5.6**, то мы и воспользуемся ею, тем более что нам достаточно будет даже ограниченных возможностей этой версии.

Главное ограничение демо-версии, это отсутствие возможности сохранения ассемблерного кода в файл, что не позволит использовать интерактивные возможности «Иды» длительное время, однако для наших целей тестирования это не очень актуально. Тем не менее, **IdaPro** оставила нам лазейку сохранения листинга кода в файл через буфер обмена (**Ctrl-Ins** или **Ctrl-C / Shift-Ins** или **Ctrl-V**). Правда, при этом сохраняются и адреса команд, но их очень легко убрать в полученном файле листинга. К этому мы еще вернемся, а пока займемся настройками «Иды».

Мы выберем наименьшую **GUI**-шную программу, которую можно найти на наших компьютерах под управлением **Windows NT**. У меня это программа **write.exe** (обёртка для **WordPad.exe** из каталога по умолчанию **\Program Files\Windows NT\Accessories**).

Для тестирования у меня было три системы. Две под управлением **XP**юши, с сервис паком **3**, но разных сборок и одна под **Windows 2003 Server**. У **XP**юши **write.exe** имел одну и ту же версию **5.1.2600.0**, но был разных размеров **5632** и **22528** байт. Как оказалось, «лишние» байты занимал не вирус или троян 😊, а данные ресурсов. В данном случае вместо двух иконок для первого файла мы имеем четыре иконки для второго. Общий размер ресурсов (вместе с версионной информацией), соответственно, **2092** и **18692** байт.

Для **Windows 2003** у **write.exe** версия **5.2.3790.0**, размером **5632** байта. Кстати, этот файл отличается от подобного файла **XP**юши практически на треть (и не только за счет ресурсов). Так что испытывать мы будем все эти версии, но начнем с меньшей версии из **Windows XP**.

Настройка IdaPro v. 5.6

Для максимальной эффективности декомпиляции воспользуемся следующими настройками.

На странице «**Welcome to the PE executable file loading Wizard**» отмечаем галку в опции «**Analysis options**». Если мы хотим, чтобы эта галка стояла по умолчанию, то тогда просто заменяем в файле **IDAPro56\cfg\kernel.xml** строку

```
<checkbox X="analysis_c" caption="Analysis options" onClick="onClick_analysis_c">
```

на

```
<checkbox checked="true" X="analysis_c" caption="Analysis options" onClick="onClick_analysis_c">
```

Однако изменять файл **kernel.xml** следует в последнюю очередь, так как после этого становятся недоступны некоторые страницы настроек.

Следующую страницу Визарда «**Segment Creation**» оставляем без изменений (сегмент импорта создаем, а сегмент ресурсов – нет).

Далее, на странице «**Kernel Options**» нам надо снять галку по умолчанию с опции «**Automatically hide library functions**», так как нам не незачем скрывать функции, которые мы собираемся перекомпилировать. Чтобы не делать это постоянно, надо отредактировать файл **\IDAPro56\cfg\ida.cfg**. Там мы видим следующее определение:

```
#define AF2_HFLIRT 0x0004 // Automatically hide library functions
```

Это значение входит в содержимое переменной **ANALYSIS2**, которое нам следует уменьшить на **4**. Однако эта переменная встречается в данном файле несколько раз, определяя по умолчанию различные настройки для разных процессоров. Можно изменить их все (заменим последнюю шестнадцатеричную цифру **D** на **9**), а можно найти только нужный нам случай. Это будет секция:

```
#ifdef __PC__          // INTEL 80x86 PROCESSORS
...
ANALYSIS2 = 0x3FFD    // Enable 'noret' analysis
                    // Enable SP analysis
#endif // __PC__
```

Вместо исходной строки пишем:

```
ANALYSIS2 = 0x3FF9
```

Следующие страницы настроек Визарда будут «**PC Processor Options**» и «**File loading**». Их мы оставляем без изменений. Проверяем, смотрим. Все как надо. Теперь можно поменять и файл **kernel.xml**, описанный выше. После этого страницы «**Kernel Options**» и «**PC Processor Options**» станут недоступными, но тем лучше для нас. Нужные нам настройки там останутся, а лишних кликов мышью делать придется меньше 😊.

Дизассемблинг **write.exe**

Загрузим нашего «подопытного кролика», в данном случае, файл **\WINDOWS\system32\write.exe** версии **5.1.2600.0**, размером **5632** байта, в дизассемблер **IdaPro v. 5.6**. Работаем под управлением **XP**юши. Интернет доступен.

Думаю, что процесс загрузки достаточно очевиден, и мы можем, наконец-то лицезреть вожаемый ассемблерный листинг (**рис. 1**).

```

IDA View-A | x Hex View-A | x Structures | x En Enums | x Imports | x Exports
.text:01001137 public start
.text:01001137 start proc near
.text:01001137
.text:01001137 Code = dword ptr -80h
.text:01001137 var_7C = dword ptr -7Ch
.text:01001137 StartupInfo = _STARTUPINFOA ptr -78h
.text:01001137 var_34 = dword ptr -34h
.text:01001137 var_30 = dword ptr -30h
.text:01001137 var_2C = byte ptr -2Ch
.text:01001137 var_28 = byte ptr -28h
.text:01001137 var_24 = byte ptr -24h
.text:01001137 var_20 = dword ptr -20h
.text:01001137 var_1C = dword ptr -1Ch
.text:01001137 ms_exc = CPPEH_RECORD ptr -18h
.text:01001137
.text:01001137 push 70h
.text:01001139 push offset stru_1001098
.text:0100113E call __SEH_prolog
.text:01001143 xor ebx, ebx
.text:01001145 push ebx ; lpModuleName
.text:01001146 mov edi, ds:GetModuleHandleA
.text:0100114C call edi ; GetModuleHandleA
.text:0100114E cmp word ptr [eax], 5A4Dh

```

Рис. 1. Дизассемблерный листинг программы **write.exe**.

Однако сразу бросается в глаза отсутствие прототипов структур **write.exe** на вкладке **Structures** (рис. 2) и «облегченное» использование прототипов функций в секции **.idata** (рис. 4).

```

IDA View-A | x Hex View-A | x Structures | x En Enums | x Imports | x Exports
; Ins/Del : create/delete structure
; D/A/* : create structure member (data/ascii/array)
; N : rename structure or structure member
; U : delete structure member

```

Рис. 2. Отсутствие структур в **IdaPro 5.6** для программы **write.exe**.

Как оказалось, со структурами просто вышел глюк. Если удалить вкладку со структурами (**Alt-F3**) и затем снова восстановить ее (через меню, либо по **Shift-F9**), то все нужные структуры появятся (рис. 3). Только не забудьте прокрутить окно вверх, а то будет казаться, что восстановлена только одна структура вместо трех необходимых.

```

IDA View-A | x Hex View-A | x Structures | x En Enums | x Imports | x Exports
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member (data/ascii/array)
00000000 ; N : rename structure or structure member
00000000 ; U : delete structure member
00000000 ; [00000044 BYTES. COLLAPSED STRUCT _STARTUPINFOA. PRESS KEYPAD "+" TO EXPAND]
00000000 ; [00000018 BYTES. COLLAPSED STRUCT CPPEH_RECORD. PRESS KEYPAD "+" TO EXPAND]
00000000 ; [0000000C BYTES. COLLAPSED STRUCT _msEH. PRESS KEYPAD "+" TO EXPAND]

```

Рис. 3. Восстановление структур после переоткрытия окна.

Кстати, эти структуры можно сразу сохранить в файл (предварительно раскрыв их «серым» плюсом) через буфер обмена, скажем, в файл **write.lst**.

```
IDA View-A | x Hex View-A | x Structures | x En Enums | x Imports | x Exports
.idata:01001000
.idata:01001000 ; Segment type: Externs
.idata:01001000 ; _idata
.idata:01001000 ; void __stdcall GetStartupInfoA(LPSTARTUPINFOA lpStartupInfo)
.idata:01001000 ; extrn GetStartupInfoA:dword ; CODE XREF: sub_10010C0+47↓p
.idata:01001000 ; ; start+148↓p
.idata:01001000 ; ; DATA XREF: ...
.idata:01001004 ; LPSTR __stdcall GetCommandLineA()
.idata:01001004 ; extrn GetCommandLineA:dword ; CODE XREF: sub_10010C0+6↓p
.idata:01001004 ; ; DATA XREF: sub_10010C0+6↓r
.idata:01001008 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
.idata:01001008 ; extrn GetModuleHandleA:dword ; CODE XREF: start+15↓p
.idata:01001008 ; ; start+16C↓p
.idata:01001008 ; ; DATA XREF: ...
.idata:0100100C
.idata:01001010 ;
.idata:01001010 ; Imports from SHELL32.dll
.idata:01001010 ;
.idata:01001010 ; HINSTANCE __stdcall ShellExecuteA(HWND hwnd, LPCSTR lpOperation, LPCSTR
.idata:01001010 ; extrn ShellExecuteA:dword ; CODE XREF: sub_10010C0+6B↓p
.idata:01001010 ; ; DATA XREF: sub_10010C0+6B↓r ...
```

Рис. 4. «Облегченное» использование прототипов функций в секции **.idata**.

По поводу прототипов, **IdaPro**, например, пишет:

```
; Imports from SHELL32.dll
; HINSTANCE __stdcall ShellExecuteA(HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile,
; LPCSTR lpParameters, LPCSTR lpDirectory, INT nShowCmd)

extrn ShellExecuteA:dword .
```

В данном случае правильной будет запись (так как мы имеем **6** параметров, каждый размером **4** байта, всего **6*4 = 24** байта)

```
extrn ShellExecuteA@24:dword
```

либо

```
extrn _imp__ShellExecuteA@24:dword
```

В таком виде (плюс лидирующий символ подчеркивания «**_**», который добавляет компилятор) эти имена функций присутствуют в библиотечных **lib**-файлах, которые нам нужно будет подгружать при компиляции ассемблерного кода. Понятно, что мы можем внести все нужные изменения вручную, но зачем, если по-хорошему, то это задача «Иды».

Конкретно, это означает, что отладочная информация из соответствующих **pdb**-файлов не была автоматически загружена из Интернета (с сайта **microsoft.com**). По этому поводу **Ильфак Гильфанов** пишет в своей документации, что в этом случае, он типа не виноват, это все происки зловредной Виндозы. Мол, он грузит отладочные символы из Интернета или локально не сам, а через системную **\WINDOWS\system32\imagehlp.dll**. А она де может отказать в загрузке, по только ей известным причинам. Но мы то верим в это с большим трудом, так как в предыдущей версии **IdaPro 5.5** (ее уже нет на сайте **Ильфака**, но при желании можно найти в бескрайних просторах Сети) отладочные **pdb**-файлы прекрасно грузятся автоматом из Интернета, даже через прокси-сервер. Но не будем ради этого возвращаться на предыдущую версию. Ситуация в нашем случае такова. Под **Windows XP** (разных сборок) демо-версия «Иды» категорически отказывается подгружать автоматом или пулеметом 😞, то бишь, вручную отладочные символы, а **Windows 2003 Server** «соглашается» делать это только локально, т.е. загружать **pdb**-файлы с жесткого диска. «Ида» при этом ругается, на сервере:

pdb(. . \IDapro56\Temp\write.exe): Unknown error, code: 0x806D0012,

а в русскоговорящей **XP**юше:

pdb(. . \IDapro56\Temp\write.exe): Класс не зарегистрирован.

Но поскольку лезть во внутренности «Иды», в нашу задачу не входит, то мы продолжим наши исследования в серверной ОСи и расскажем, как можно работать без отладочных символов, если они, по каким-либо причинам, недоступны.

Заметим, что в **Masm32** (<http://www.masm32.com/masmdl.htm>) прототипы функций определяются по-другому. Тот же пример функции **ShellExecuteA** там (в **Include\shell32.inc**) описан следующим образом:

```
ShellExecuteA PROTO :DWORD, :DWORD, :DWORD, :DWORD, :DWORD, :DWORD  
ShellExecute equ <ShellExecuteA>
```

а сам файл прототипов подключается к проекту с помощью инструкции:

```
include Include\shell32.inc
```

Ничто не мешает нам использовать уже готовые файлы прототипов функций (стиль **Iczelion**'а). Это, кстати, уменьшает зависимость от **pdb**-файлов. Однако, лично мне кажется, что для наших целей вариант с использованием директивы **extrn** (или **extern**) более предпочтителен. Впрочем, окончательный выбор за вами.

Загрузка **pdb**-файлов из Интернета

Но чтобы продолжить нашу работу, нам надо вручную скачать у мелкософта отладочные символы. Проще всего это сделать, загрузив всю упаковку предлагаемых **pdb**-файлов для выбранной операционки с конкретным сервис паком. Для этого идем на стартовую страницу загрузки отладочных символов: <http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx#d> .

Для **Windows XP, sp. 3** инсталяк **pdb**-файлов занимает **214** Мб, а для **Windows 2003 Server, sp. 2** – около **160** Мб. Можно, конечно, пытаться делать это индивидуально для конкретных файлов. Версия 5.5 «Иды» посылает следующий **Get** запрос, для **XP**-шной версии:

```
srv*d:\Temp\Ida*http://msdl.microsoft.com/download/symbols/write.pdb/3b7d841d1/write.pdb
```

Для серверной ОСи, аналогичный запрос:

```
srv*d:\Temp\Ida*http://msdl.microsoft.com/download/symbols/write.pdb/3e8000e61/write.pdb
```

Хотя это и **Get** запрос, только бесполезно передавать его непосредственно в браузер. По-видимому, нужно еще обмениваться файлами **cookie** с мелкософтом. Короче, геморно, это все. Проще скачать всю упаковку (или даже несколько) с символами и не париться. Что мы и сделаем.

Итак, обе упаковки с отладочными символами получены (в виде **pdb**-файлов) для обеих, рассматриваемых нами, операционных систем.

Ручная подгрузка **pdb**-файлов в **IdaPro 5.6**

Вручную это можно сделать двумя способами.

Первый способ заключается в том, чтобы положить файл **write.pdb** рядом с файлом **write.exe**. По крайней мере, серверная ОСь понимает это правильно, и подгружает сама нужный нам файл, в результате чего мы получаем то, к чему стремились (**рис. 5**).

```

IDA View-A | x Hex View-A | x Structures | x En Enums | x Imports | x Exports
.idata:01001000 ; Segment type: Externs
.idata:01001000 ; _idata
.idata:01001000 ; void __stdcall GetStartupInfoA(LPSTARTUPINFOA lpStartupInfo)
.idata:01001000 ; extrn __imp__GetStartupInfoA@4:dword
.idata:01001000 ; CODE XREF: WinMain(x,x,x,x)+47↓p
.idata:01001000 ; _WinMainCRTStartup+148↓p
.idata:01001000 ; DATA XREF: ...
.idata:01001004 ; LPSTR __stdcall GetCommandLineA()
.idata:01001004 ; extrn __imp__GetCommandLineA@0:dword
.idata:01001004 ; CODE XREF: WinMain(x,x,x,x)+6↓p
.idata:01001004 ; DATA XREF: WinMain(x,x,x,x)+6↓r
.idata:01001008 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
.idata:01001008 ; extrn __imp__GetModuleHandleA@4:dword
.idata:01001008 ; CODE XREF: _WinMainCRTStartup+15↓p
.idata:01001008 ; _WinMainCRTStartup+16C↓p
.idata:01001008 ; DATA XREF: ...
.idata:0100100C ; extrn _KERNEL32_NULL_THUNK_DATA:byte:4
.idata:01001010 ;
.idata:01001010 ; Imports from SHELL32.dll
.idata:01001010 ;
.idata:01001010 ; HINSTANCE __stdcall ShellExecuteA(HWND hwnd, LPCSTR lpOperation, LPCSTR lp
.idata:01001010 ; extrn __imp__ShellExecuteA@24:dword
.idata:01001010 ; CODE XREF: WinMain(x,x,x,x)+6B↓p

```

Рис. 5. Прототипы функций после загрузки отладочных символов.

Вот! Теперь мы вполне удовлетворены полученным результатом. Распознавание прототипов функций осуществлено успешно. Теперь, быстренько (так как время работы одного сеанса демо-версии ограничено и составляет порядка **30** минут) выделяем весь полученный листинг и сохраняем в тот же файл **write.lst**, вставив ассемблерный листинг после описания структур.

Второй способ ручного использования **pdb**-файлов заключается в его непосредственном выборе из **IdaPro** через пункт меню **File / Load file / PDB file . . .** Мы просто указываем путь, выбрав маску файлов (*.*) для нужного файла символов. Заметим только, что выбор не того файла приведет к непредсказуемым последствиям по перекодировке ассемблерного листинга.

Напомним, что в двух различных сборках **XP**юши, «Ида» отказывается это делать, мотивируя тем, что «класс не определен». Какой класс и чего, нам объяснять не считают нужным. Можно только предположить, что это глюки самой демо-версии **5.6** либо ее плагина **pdb.plw**, хотя на серверной ОСи этот плагин вполне успешно работает в предыдущей версии «Иды» (и даже автоматом грузит символы из Интернета, в том числе, через прокси-сервер).

На случай, если нет интернета или файла **write.pdb**, покажем, как вручную сделать замену символов в рассматриваемом листинге **write.lst**. Для этого во всем файле нужно сделать замену строк из левого столбца таблицы на соответствующие строки из правого.

Таблица 1. Ручная замена символов в листинге **write.lst**

№	До замены	После замены	Модуль dll
1	GetStartupInfoA	_imp__GetStartupInfoA@4	Kernel32.dll
2	GetCommandLineA	_imp__GetCommandLineA@0	Kernel32.dll
3	GetModuleHandleA	_imp__GetModuleHandleA@4	Kernel32.dll
4	ShellExecuteA	_imp__ShellExecuteA@24	shell32.dll
5	_exit	_imp__exit	msvcrt.dll
6	__imp__XcptFilter	_imp__XcptFilter	msvcrt.dll
7	_cexit	_imp__cexit	msvcrt.dll
8	exit	_imp__exit	msvcrt.dll
9	_acmdln	_imp__acmdln	msvcrt.dll

10	__getmainargs	_imp__getmainargs	msvcrt.dll
11	_c_exit	_imp__c_exit	msvcrt.dll
12	__setusermatherr	_imp__setusermatherr	msvcrt.dll
13	_adjust_fdiv	_imp__adjust_fdiv	msvcrt.dll
14	__p_commode	_imp__p_commode	msvcrt.dll
15	__p_fmode	_imp__p_fmode	msvcrt.dll
16	__set_app_type	_imp__set_app_type	msvcrt.dll
17	__imp__controlfp	_imp__controlfp	msvcrt.dll
18	__imp__except_handler3	_imp__except_handler3	msvcrt.dll
19	__imp__initterm	_imp__initterm	msvcrt.dll

В данном случае, это основная суть того, что делает «Ида», подгружая файл с отладочными символами. Как видим, не очень много и, может быть, не стоило бы так заморачиваться с **pdb**-файлами. Однако, для больших проектов, без отладочных символов трудоемкость работы может существенно возрасти.

Получение из файла листинга ассемблерного файла

Полученный файл дизассемблерного листинга после копирования через буфер обмена, содержит не нужные, для целей компиляции, адреса строк кода, перед собственно кодом ассемблера (**рис. 6**).

```

00000000
00000000 _msEH      struc ; (sizeof=0xC)
00000000 _unk      dd ?          ; base 16
00000004 FilterProc dd ?          ; offset
00000008 ExitProc  dd ?          ; offset
0000000C _msEH      ends
0000000C

.idata:01001000 ;
.idata:01001000 ; +-----+
.idata:01001000 ; | This file has been generated by The Interactive Disassembler (IDA) |
.idata:01001000 ; | Copyright (c) 2009 by Hex-Rays, <support@hex-rays.com> |
.idata:01001000 ; | Evaluation version |
.idata:01001000 ; +-----+
.idata:01001000 ;
.idata:01001000 ; Input MD5 : A916B47D484DDF010305686F77D8F4B5
.idata:01001000 ;
.idata:01001000 ; File Name : \IDApr56\Temp\write.exe
.idata:01001000 ; Format : Portable executable for 80386 (PE)
.idata:01001000 ; Imagebase : 10000000
.idata:01001000 ; Section 1. (virtual address 00001000)
.idata:01001000 ; Virtual size : 0000055C ( 1372.)
.idata:01001000 ; Section size in file : 00000600 ( 1536.)
.idata:01001000 ; Offset to raw data for section: 00000400
.idata:01001000 ; Flags 60000020: Text Executable Readable
.idata:01001000 ; Alignment : default
.idata:01001000 ;
.idata:01001000 ; Imports from KERNEL32.dll
.idata:01001000 ;
.idata:01001000 ;
.idata:01001000 ; .686p
.idata:01001000 ; .mmx
.idata:01001000 ; .model flat
.idata:01001000 ;

```

Рис. 6. Файл дизассемблерного листинга, полученный через буфер обмена.

В общем-то, столбец с адресами можно попытаться удалить вручную. Так, например, многие редакторы (лучше с моноширинным шрифтом) поддерживают выделения столбца текста с помощью **Alt** + левая кнопка мыши, либо **Shift** + **Alt** + управление стрелками / **PageUp** / **PageDown** / **Home** / **End**. Но нам это делать как-то лениво 😊, поэтому мы лучше напишем небольшой скрипт, который будет выполнять за нас всю эту «черную» работу. Лично мне привычен **FoxPro**, который, на уровне ядра, очень неплох,

для любых версий. Однако в данном случае это неважно. Вы можете навать собственный скрипт, на любом удобном вам языке.

Вот пример файла **lst.prg**, выполняющийся под управлением **Visual FoxPro**:

```
*****
CLEAR
SET TALK OFF
SET SAFETY OFF

filename = "write"

infile = filename + ".lst"
outfile = filename + ".asm"

file1 = FOPEN(infile)

IF file1 < 0
    WAIT 'Не могу открыть файл: ' + infile
    QUIT
ENDIF

file2 = FCREATE(outfile)

IF file2 < 0
    WAIT 'Не могу создать файл: ' + outfile
    QUIT
ENDIF

End1 = FSEEK(file1, 0, 2) && Определяем размер файла
Top1 = FSEEK(file1, 0) && Идем в начало. Сама переменная не нужна

IF End1 <= 0 && Если файл пуст
    MESSAGEBOX('Пустой файл: ' + infile)
    QUIT
ENDIF

pos = 0

strline = FGETS(file1)

DO WHILE NOT EOF(file1) && На самом деле мы не можем доверять этой проверке
    Curpos1 = FSEEK(file1, 0, 1)

    IF Curpos1 >= End1 && Достигнут конец файла?
        pos = AT(" ", strline)
        FPUTS(file2, SUBSTR(strline, pos + 1))
        EXIT
    ENDIF

    IF LEFT(strline, 1) = " "
        FPUTS(file2, strline)
        LOOP
    ENDIF

    pos = AT(" ", strline)

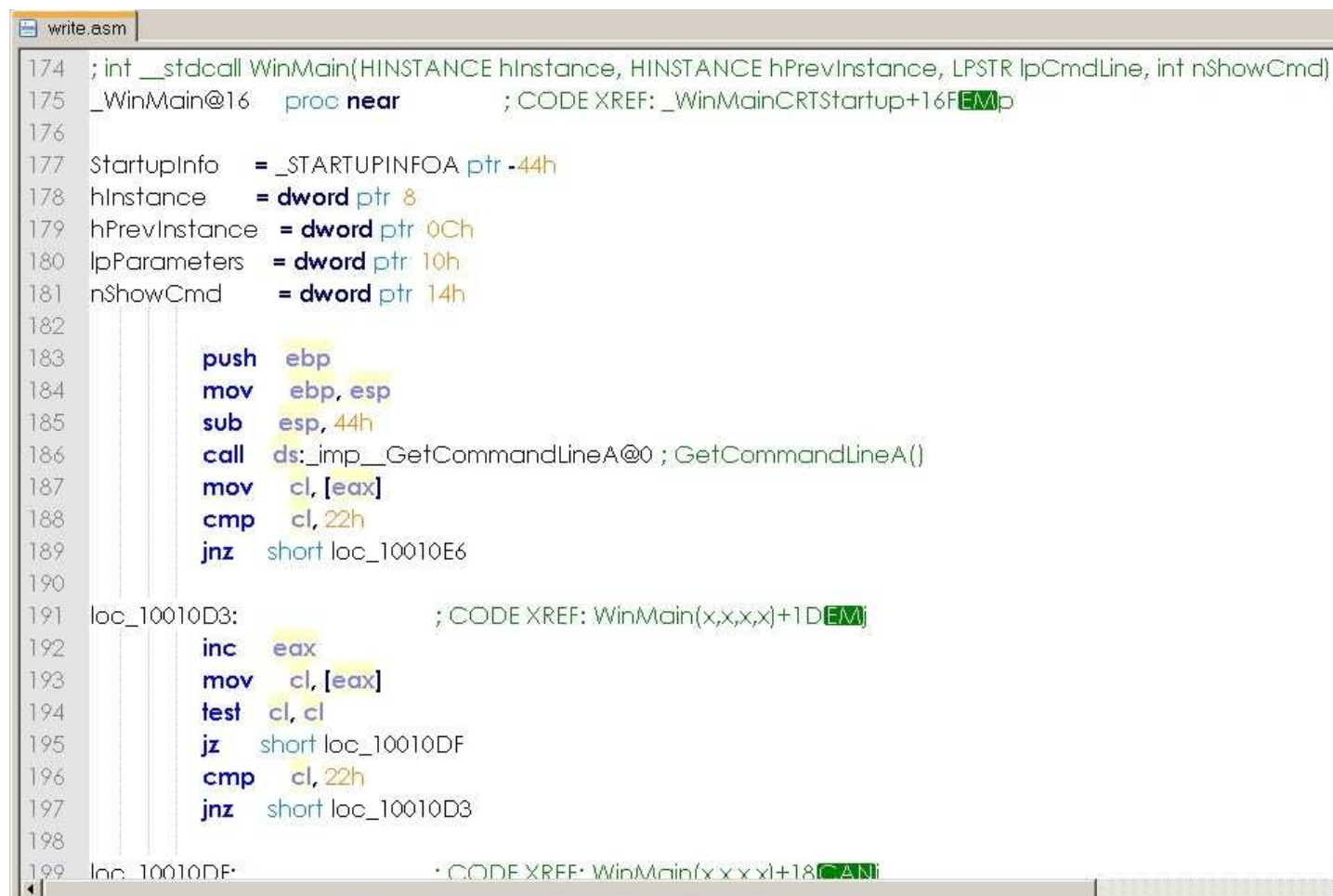
    IF pos > 0
        FPUTS(file2, SUBSTR(strline, pos + 1))
    ELSE
        FPUTS(file2, "")
    ENDIF

    strline = FGETS(file1)
ENDDO

CLOSE ALL
```

QUIT

Итак, в нашем распоряжении ассемблерный файл **write.asm** (рис. 7), который уже можно пытаться компилировать. Поэтому сейчас самое время поговорить о способах компиляции **asm** кода.



```
174 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
175 _WinMain@16 proc near ; CODE XREF: _WinMainCRTStartup+16F [EIP]
176
177 StartupInfo = _STARTUPINFOA ptr -44h
178 hInstance = dword ptr 8
179 hPrevInstance = dword ptr 0Ch
180 lpParameters = dword ptr 10h
181 nShowCmd = dword ptr 14h
182
183 push ebp
184 mov ebp, esp
185 sub esp, 44h
186 call ds:_imp__GetCommandLineA@0 ; GetCommandLineA()
187 mov cl, [eax]
188 cmp cl, 22h
189 jnz short loc_10010E6
190
191 loc_10010D3: ; CODE XREF: WinMain(x,x,x,x)+1D [EIP]
192 inc eax
193 mov cl, [eax]
194 test cl, cl
195 jz short loc_10010DF
196 cmp cl, 22h
197 jnz short loc_10010D3
198
199 loc_10010DF: ; CODE XREF: WinMain(x,x,x,x)+18 [EIP]
```

Рис. 7. Файл **write.asm**.

Компиляция ассемблерного кода

Есть множество версий разного рода ассемблеров и соответствующих программных компиляторов. Также немало и всевозможной документации по этим вопросам. Но лично мне кажется, что начинать нужно с классики жанра – знаменитого tutorиала от **Iczelion**'а ([http://wasm.ru/docs/1/Iczelions_Tutorials_\(for_print\).zip](http://wasm.ru/docs/1/Iczelions_Tutorials_(for_print).zip)) и используемого им пакета **Masm32** (<http://www.masm32.com/masmdl.htm>) .

Пакет **Masm32** (на сегодня 10-я версия) на самом деле имеет компилятор макро ассемблера **ml.exe** от 1999 года. Т.е. он много лет практически не меняется. Возможно это последняя бесплатная версия, ибо более поздние компиляторы **ml.exe** принадлежат **Microsoft** и входят в состав продуктов **MS Visual Studio**, по крайней мере, начиная с 7-й версии (2000 год). Так что, если у вас есть **MS VC7** или выше из **VS**, то вы можете воспользоваться более новой версией ассемблерного компилятора.

В принципе нас устраивает любая версия **Masm32**, хоть бесплатная, хоть от **MS**. Поэтому, для чистоты эксперимента будем работать с **Masm32** и компилятором от 1999 года.

Вот перечень файлов, которые нам понадобятся. Они (или аналогичные) есть также и в **Visual C++** из **MS Visual Studio**.

Каталог **Bin**:

ml.exe – компилятор макро ассемблера;
ml.err – пустой текстовый файл, для вывода ошибок компилятора;
link.exe – линковщик;
rc.exe – компилятор ресурсов;
rcdll.dll – библиотека компилятора ресурсов;
cvtres.exe – дополнительный компилятор ресурсов;
mspdb50.dll – общая библиотека поддержки.

Каталог **Lib** (для наших целей достаточно):

kernel32.lib
shell32.lib
msvcrt.lib

Выше этих каталогов мы будем располагать следующие файлы:

write.asm – полученный с помощью **IdaPro** ассемблерный листинг программы **write.exe**;
headers.inc – файл заголовка для **write.asm**, описан ниже;
write.res – файл ресурсов для **write.asm**, описан ниже;
asm.bat – командный файл, описан ниже.

Опишем используемые нами дополнительные файлы.

Файл заголовка **headers.inc**

Поскольку нам нужны будут в обязательном порядке **lib**-файлы, для используемых в ассемблерном коде функций из системных **dll**-лек, то сразу вставим строку

include headers.inc

после строки

.model flat

в файле **write.asm**. Содержимым файла **headers.inc** будут строки:

```
=====
includelib Lib\kernel32.lib
includelib Lib\shell32.lib
includelib Lib\msvcrt.lib
=====
```

Командный файл **asm.bat**

Вот пример командного файла **asm.bat**, который мы будем использовать для наших целей:

SET FILENAME=write

:: Компиляция ресурсов (не нужна, если мы берем готовый res-файл от Resource Builder)
:: Bin\rc /r /v %FILENAME%.rc >a.a

:: Компиляция ассемблерного кода
Bin\ml /c /coff /Cp /Cx /Zp1 /Gz /Zm %FILENAME%.asm > b.a
:: /c – компиляция без линковки

```
:: /coff – обязательная опция для формата объектного (obj) файла
:: /Cr – резервирование регистра для пользовательских идентификаторов
:: /Cx – резервирование регистра для глобальных и экспортируемых символов
:: /Zp[n] – установить выравнивание структур в n
:: /Gz – применение вызова функций типа Stdcall
:: /Zm – совместимость с версией MASM 5.10
```

:: **Линковка** (создание **exe**-файла)

```
Bin/link /BASE:0x1000000 %FILENAME%.obj %FILENAME%.res /subsystem:windows >c.a
```

:: /BASE:0x1000000 – поскольку в листинге «Иды» указана эта база

:: Первоначально текст %FILENAME%.res можно убрать, для компиляции без ресурсов

```
del *.obj
:: del *.res
```

Здесь в текстовых файлах **a.a**, **b.a** и **c.a** записываются сообщения о выполнении процессов компиляции и линковки данного кода. Описание ошибок, которые мы сейчас начнем разбирать, находятся в файле **b.a**.

Разбор ошибок

В основном работа с ошибками будет заключаться в удалении ненужного (fake) кода. Будем исходить из случая, когда используются отладочные символы. Если мы делали замену символов вручную, на основе **таблицы 1**, но некоторых ошибок не будет.

Итак, компилятор ругается на строки вида:

```
extrn _KERNEL32_NULL_THUNK_DATA:byte:4
...
extrn _SHELL32_NULL_THUNK_DATA:byte:4
...
extrn _msvcrt_NULL_THUNK_DATA:byte:4
```

Эти экспортируемые переменные не используются в нашем коде. Зачем они вставлены из **pdb**-файлов, непонятно. Ну, да ладно, сносим их не глядя.

Далее, в нашем коде есть строка:

```
align 1000h
```

которая явно не нравится компилятору. Максимум, что готов принять компилятор это значение вида

```
align 10h
```

так что этот вариант и оставим.

Также не любит компилятор строки вида:

```
mov    eax, large fs:0
```

Экспериментально выяснилось, что его смущает директива **large**. Убираем ее полностью, делая автозамену по всему файлу. Тем самым уменьшаем общее количество ошибок на три.

Остальные ошибки, как оказалось, связаны с директивой **rva**, которая встречается много раз. Например, в коде:

```
; -----
align 4
__IMPORT_DESCRIPTOR_SHELL32 dd rva off_10013D8 ; Import Name Table
dd 0FFFFFFFFh             ; Time stamp
dd 0FFFFFFFFh             ; Forwarder Chain
dd rva aShell32_dll       ; DLL Name
dd rva __imp__ShellExecuteA@24 ; Import Address Table
...
```

Однако этот и оставшийся код (до конца секции) **_text**

```
...
db 'GetModuleHandleA',0
align 4
dd 29h dup(0)
dd 280h dup(?)
_text     ends
```

связан со старой таблицей импорта, которая в нашем случае будет формироваться заново. Поэтому лучшим решением будет удалить этот блок кода полностью. Что мы и делаем, избавляясь заодно от последних ошибок компилятора. Однако для успокоения совести добавим вместо него другой код:

```
align 10h
```

Ура! Компилятор согласился с нашими довольно радикальными операциями и скомпилировал объектный код в виде **write.obj**. Однако теперь настала очередь ругаться линковщику 😞 (файл **c.a**). Практически ему не нравятся все имена функций перечисленных в правой части **таблицы 1**, к которым, оказались, добавлены лидирующие символы подчеркивания (как в **lib**-файле), т.е. перед префиксом **imp** два символа нижнего подчеркивания «**__**» (а в описании ошибок линковщика даже три), а не один, как у нас. Но это уже не проблема, а всего лишь задача, как говорят студенты 😞, которая решается в два хлопка автозаменой «**__imp**» на «**_imp**» в файле **write.asm**. Короче, хлопаем в ладоши, лишние символы удаляются, и наш код прекрасно собирается в **exe**-шник **write.exe**. И о, чудо! Программа запускается и можно работать с вызванным ею редактором **WordPad** (рис. 8).

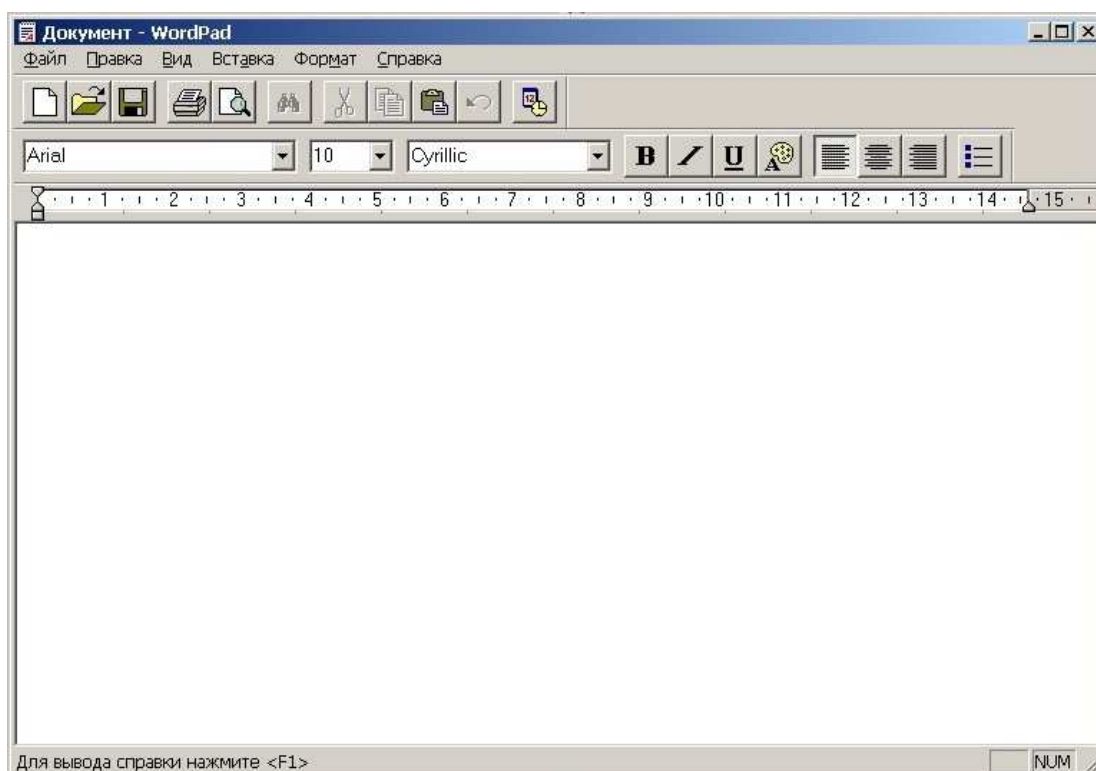


Рис. 8. Запуск перекомпилированного файла **write.exe** – обёртки для **WordPad**.

В данном случае, мы проигнорировали линковку бинарного файла ресурсов **write.res**, в командном файле **asm.bat**, так как пока у нас нет этого файла, что однако не помешало запуститься нашей конфетке (в смысле обёртке 😊) .

Работа с ресурсами

Теперь, когда мы добились принципиальной работоспособности нашего рекомпиляционного кода, можно, после небольшого перекура (быстренько два часа отдохнем, потом пять минут пор-р-р-работаем, пор-р-р-работаем 😊), заняться ресурсами. Ибо не факт, что и любая другая подопытная программа, после декомпиляции, будет также хорошо работать без собственных ресурсов.

IdaPro позволяет создавать сегмент ресурсов (но мы то специально эту галку не ставили). Только это будет что-то вроде секции данных, что не очень удобно для анализа. Лучше найти подходящий редактор ресурсов, который позволит извлекать ресурсы из исполнимых файлов в нужном нам виде. И такой редактор сейчас действительно есть. Речь идет о такой чудесной программе как **Resource Builder**, триальную версию которой вы можете скачать с <http://www.resource-builder.ru/download.html> . Для демонстрации наших исследований ее возможностей вполне достаточно.

Мы не будем описывать принципы работы с **Resource Builder**. В программе поддерживается русский язык, и документации у нее хватает. Достаточно сказать, что мы можем извлечь с помощью нее нужные нам ресурсы, как в бинарном виде **write.res**, так и в виде описания сценариев ресурсов – файл **write.rc**. Заодно мы потренируемся в модификации манифеста ресурсов. Для этого изменим, например, строку описания программы (**рис. 9**). Затем компилируем измененные ресурсы во внешние файлы, которыми можно будет уже воспользоваться.

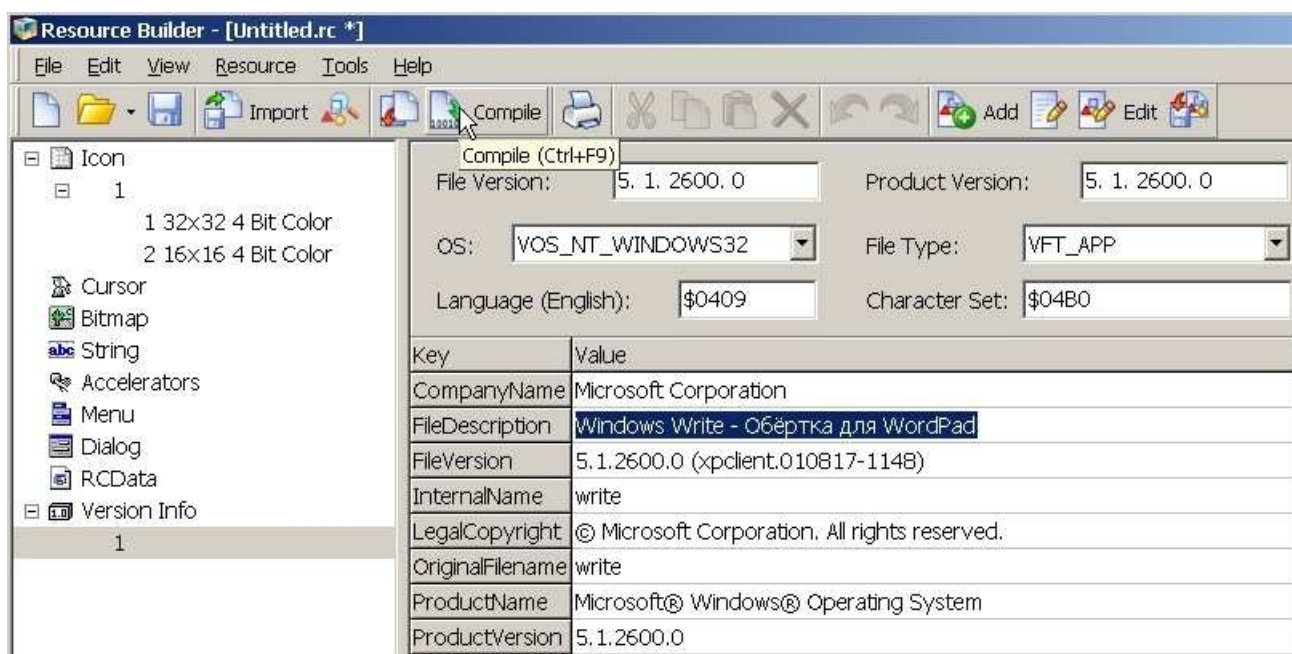


Рис. 9. Извлечение и модификация ресурсов **write.exe** с помощью **Resource Builder**.

Для наших целей мы можем воспользоваться либо уже готовым для применения бинарным файлом **write.res**, либо, после небольшого редактирования, файлом сценариев **write.rc**. В последнем случае нам надо будет раскомментировать строку:

```
Bin\rc /r /v %FILENAME%.rc >a.a
```

в файле **asm.bat**.

Изменения в файле описания ресурсов **write.rc** касаются, в основном, явных определений используемых констант (которые можно найти в **h**-файлах **MS VC++**) и редактирования пути для иконки **write.ico**, которая также создается **Resource Builder**'ом. Приведем конечное его содержание:

```

/*****
File: WRITE.RC
Generated by Resource Builder (3.0.3.25).
*****/

/*
OutputExt=res
*/

//MS Visual C++ compatible header
// #define APSTUDIO_READONLY_SYMBOLS
// #include "afxres.h"
// #undef APSTUDIO_READONLY_SYMBOLS
//end of MS Visual C++ compatible header

// Наши определения, взятые в MS Visual C++
#define LANG_ENGLISH          0x09
#define VOS_NT_WINDOWS32      0x00040004L
#define VFT_APP                0x00000001L

// Иконку, полученную в Resource Builder, переименовываем в write.ico и помещаем рядом с write.asm
LANGUAGE LANG_ENGLISH, 1
1 ICON "write.ico"

// Version Info
1 VERSIONINFO
FILEVERSION 5, 1, 2600, 0
PRODUCTVERSION 5, 1, 2600, 0
FILEOS VOS_NT_WINDOWS32
FILETYPE VFT_APP {
    BLOCK "StringFileInfo" {
        BLOCK "040904B0" {
            VALUE "CompanyName", "Microsoft Corporation"
            VALUE "FileDescription", "Windows Write - Обёртка для WordPad"
            VALUE "FileVersion", "5.1.2600.0 (xpclient.010817-1148)"
            VALUE "InternalName", "write"
            VALUE "LegalCopyright", "© Microsoft Corporation. All rights reserved."
            VALUE "OriginalFilename", "write"
            VALUE "ProductName", "Microsoft® Windows® Operating System"
            VALUE "ProductVersion", "5.1.2600.0"
        }
    }

    BLOCK "VarFileInfo" {
        VALUE "Translation", 1033, 1200
    }
}

```

Компиляция с явным использованием ресурсного скрипта также проходит успешно. Раньше о подобных возможностях приходилось только мечтать! В **Total Commander**’е можно посмотреть параметры сгенерированного нами файла **write.exe** (рис. 10). Действительно, внесенные нами изменения имеют место. При желании можно также поменять и иконку. Но это оставим вам, в качестве домашнего задания 😊.



Рис. 10. Просмотр свойств нового **write.exe** с помощью **Total Commander**.

Заметим, что размер нашего нового файла в **MASM32 v.10** (с **ml.exe 1999** года), получился не намного больше исходного **exe**-шника (разница всего **512** байт). Можно, конечно пытаться уменьшать размер результирующего файла, но это уже не принципиально для наших целей, ибо главную задачу мы выполнили.

Перекомпиляция других версий **write.exe**

Для полноты картины нам еще нужно перекомпилировать **XP**-шную версию **write.exe** (который мы переименуем в **write2.exe**) с исходным размером **22528** байт и версию **write.exe** от **Win2003** (который мы переименуем в **write3.exe**), размером также **5632** байта.

Рекомпиляция **write2.exe** прошла абсолютно по тому же сценарию, что и первый случай. Даже **pdb**-файл мы не меняли. Разницей оказались, как уже упоминалось, другие иконки (**4** штуки). В коде явных различий не обнаружено. Поэтому сразу перейдем к третьему случаю.

Рекомпиляция **write3.exe**, несмотря на небольшое отличие в коде (связанного в основном со сканированием собственного **PE**-заголовка и выбора **Image Base**, отличного по умолчанию) также прошла успешно.

Таким образом, все поставленные перед собой задачи мы полностью выполнили.

Заключение

Теперь можно подвести итоги нашего тестирования демо-версии **IdaPro 5.6** на примере простейшего **GUI**-приложения **write.exe**.

Как дизассемблер (отладочные способности мы не тестировали) **IdaPro 5.6** вполне хорош, если не считать его глюки с загрузкой **pdb**-файлов. Будем надеяться, что **Ильфак Гильфанов** обратит на это внимание. Но даже ручная правка отладочных символов (или, как альтернатива, использование стиля

программирования **Iczelion**'а) не портит впечатления от «Иды». Это действительно высококачественный продукт. Интересно будет еще попробовать мощь **IdaPro** на других приложениях, в качестве «подопытного кролика», но на более сложных, чем используемое нами.

Вот примерный перечень **GUI**-приложений (для **XP**юши) – потенциальных кандидатов на тестирование:

regedt32.exe	3584 байт
winver.exe	5632 байт
dcomcnfg.exe	6144 байт
winhlp32.exe	8192 байт
control.exe	8192 байт
eventvwr.exe	8704 байт
winmsd.exe	11776 байт

и другие.

(Оказывается, **regedt32.exe** имеет даже меньший размер, чем **write.exe**, но обнаружил это я только сейчас 😊.)

Примечание

К данному тексту приложен файл **IdaPro56Test.zip** (<http://erfaren.narod.ru/Asm/IdaPro56Test.001> - измените расширение в **zip**), с результатами тестирования. Можно также посмотреть **html** версию этой статьи (<http://erfaren.narod.ru/Asm/Erfaren001.htm>) либо ее **pdf**-файл (<http://erfaren.narod.ru/Asm/Erfaren-001-Recompilation-write-exe.pdf>).