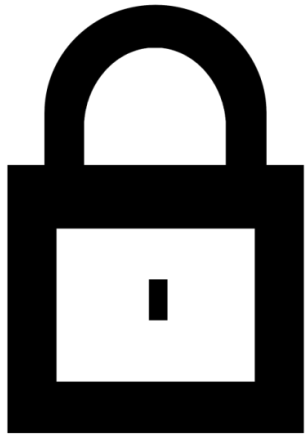
А.А. Оголюк

ЗАЩИТА ПРИЛОЖЕНИЙ ОТ МОДИФИКАЦИИ. ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

Учебное пособие



Санкт-Петербург

2014

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХТЕХНОЛОГИЙ, МЕХАНИКИИ ОПТИКИ

А.А. Оголюк

ЗАЩИТА ПРИЛОЖЕНИЙ ОТ МОДИФИКАЦИИ. ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

Учебное пособие

Санкт-Петербург

2014

Оголюк А. А. Защита приложений от модификации. Дополнительные материалы: учебное пособие. – СПб: СПбГУ ИТМО, 2014. – 122с.

Данное учебное пособие является дополнением к основному учебному пособию «Защита приложений от модификации». Пособие глубже раскрывает вопросы изучения и отладки приложений, реконструкции алгоритмов, а также включает новые приёмы работы с популярным инструментом дизассемблирования IDA. Материал пособия разбит на шесть глав. Каждая глава содержит краткий теоретический материал.

Пособие предназначено для студентов, специализирующихся в области информационных технологий, и может быть использовано при подготовке магистров по направлению 230100 «Информатика и вычислительная техника».

Рекомендовано Советом факультета Компьютерных технологий и управления 07 июля 2013 г., протокол №7

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого были определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Оглавление

ВВЕДЕНИЕ

Мы продолжаемизучение предмета «Защита приложений от модификации», и переходим от ранее изученных основных возможностей IDA к большему количеству расширеннх функций.
Учебное пособие содержит данные о полезных параметрах конфигурации IDA, дополнительных утилитах, разработанных, чтобы улучшить аналитические возможности IDA, создании файлов сигнатур FLRT. Также описывается, как работает отладчик IDA, и каких результатов можно достичь при сочетани отладчика и дизассемблера.
Полученные в ходе изучения данного курса знания позволят эффективно защищать программы от модификации и несанкционированного копирования, а также создавать более оптимизирование приложения.

ГЛАВА 1. НАСТРОЙКИ IDA

Проведя некоторое время за работой с IDA, у вас могли появиться предпочитаемые настройки, которые вы бы хотели использовать по умолчанию каждый раз, когда открываете новую базу данных. Некоторые ранее измененные вами опции будут уже перенесены из одной сессии в другую, однако другие требуют установки каждый раз при загрузке новой базы данных. В этой главе мы изучим различные пути, которыми вы можете изменять поведение IDA при помощи файлов конфигурации и опций меню. Также мы изучим, где IDA хранит различные настройки, и обсудим разницу между глобальными настройками и настройками базы данных.

1.1. Файлы конфигурации

Режим работы IDA определяется настройками, которые содержатся в различных файлах конфигурации. Большая часть этих файлов хранится в папке <IDADIR>/cfg, однако есть одно заметное исключение — файл конфигурации дополнений находится в <IDADIR>/plugins/plugins.cfg (подробнее данный файл рассматривается в главе 17). Вы могли заметить, что в папке конфигурации находится большое количество файлов, но большая часть из них используется процессорным модулем и применяется только когда производится анализ определенных типов процессоров. Тремя главными файлами конфигурации являются ida.cfg, idagui.cfg и idatui.cfg. Опции общие для всех версий IDA обычно находятся в файле ida.cfg, в то время как в файлах idagui.cfg и idatui.cfg хранятся опции специфичные для версии с графическим интерфейсом, и для текстовой версии IDA. Из-за того что версии для операционных систем отличных от ОС Windows ограничены только консольным режимом, дистрибутивы IDA для Linux и OS X не содержат idagui.cfg.

1.1.1 Основной файл конфигурации: ida.cfg

Во время загрузки этот файл читается для того, чтобы назначить типы процессоров по умолчанию для различных расширений файлов и для того, чтобы настроить параметры использования памяти IDA. После того как определился тип процессора, файл читается второй раз для обработки дополнительных опций. Опции, содержащиеся в

ida.cfg, применяются ко всем версиям IDA, независимо от используемого интерфейса пользователя.

Наиболее важными опциями в ida.cfg, являются опция выполнения резервного копирования (CREATE_BACKUPS), а также имя внешней программы просмотра графов (GRAPH_VISUALIZER).

Большое количество опций, определяющих формат строк дизассемблированного кода, также содержатся в ida.cfg, включая значения по умолчанию для различных опций, доступных из меню Options ► General. Эти настройки включают значения по умолчанию: количество байт для отображения опкода (OPCODE_BYTES), какой длины отступ должен быть у инструкций (INDENTATION), должно ли отображаться смещение указателя стека рядом с каждой инструкцией (SHOW_SP), а также максимальное число перекрестных ссылок для отображения в строке дизассемблированного кода (SHOW_XREFS). Дополнительные опции отвечают за формат строк дизассемблированного кода в режиме графа.

Глобальная опция, определяющая максимальную длину имени для именованных локаций программ (named program locations) (в отличие от переменных стека) содержится в ida.cfg и называется MAX_NAMES_LENGTH. По умолчанию эта опция установлена равной 15 символам, и вы будете получать предупреждение от IDA каждый раз, когда вводите имя длиннее установленного предела. Значение по умолчанию выбрано небольшим из-за того что некоторые ассемблеры не поддерживают имена длиннее 15 символов. Если вы не планируете запускать в ассемблере код, дизаассемблированный в IDA, то вы можете без опасений увеличить текущий предел.

Список символов, допустимых в назначаемых пользователем именах регулируется опциями NameChars. По умолчанию этот список допускает использование букв и цифр, а также четыре специальных символа _$?@. Если IDA выдает предупреждение об используемых вами символах при назначении новых имен для локаций или переменных стека, тогда вы можете добавить их в набор символов NameChars. Например, если вы хотите разрешить использование символа точки (.), то вам нужно изменить NameChars. Не рекомендуется использовать двоеточие, точку с запятой, запятую и символ пробела в именах из-за того что это может привести к путанице, так как данные символы обычно используются как разделители для различных частей строк дизассемблированного кода.

Последние две опции, о которых стоит упомянуть влияют на поведение IDA во время парсинга заголовочных файлов C. Опция C_HEADER_PATH определяет список папок, в которых IDA будет производить поиск для решения зависимостей #include. По умолчанию

назначается папка используемая Microsoft Visual Studio. Если вы пользуетесь другим компилятором или ваши заголовочные файлы находятся в другой папке, то вам стоит отредактировать эту опцию. Опция C_PREDEFINED_MACROS может быть использована для определения списка макросов препроцессора, которые IDA будет включать независимо от того, встретились ли они во время парсинга заголовочных файлов C. При помощи этой опции можно реализовать ограниченное решение для включения макросов, которые могут быть определены в заголовочных файлах, к которым у вас нет доступа.

Во второй половине ida.cfg содержатся опции, специфичные для различных процессорных модулей. Единственная доступная для этих опций документация представлена в виде комментариев к ним. Данные опции определяют значения настроек по умолчанию в разделе Processor options диалога загрузки файлов IDA.

Последний этап в обработке ida.cfg это поиск файла с названием <IDADIR>/cfg/idauser.cfg. Если он существует, то этот файл воспринимается как расширение ida.cfg, и любые опции указанные в нем переопределяют соответствующие им опции из файла ida.cfg. Если вы не желаете редактировать ida.cfg, то вам стоит создать idauser.cfg и добавить в него те опции, которые вы хотите переопределить. В дополнение, использование idauser.cfg является самым простым способом переноса настроек из одной версии IDA в другую. Например, если у вас есть файл idauser.cfg, то вам не нужно редактировать ida.cfg каждый раз, когда вы обновляете IDA, вместо этого просто копируйте idauser.cfg в папку конфигурации вашей новой установки IDA после каждого обновления.

Данный файл не поставляется вместе с IDA. Пользователь должен сам создать этот файл, если он хочет чтобы файл был обнаружен IDA.

1.1.2. Файл конфигурации графического интерфейса: idagui.cfg

Настройки Windows-версии IDA с графическим интерфейсом содержатся в файле <IDADIR>/cfg/idagui.cfg. Этот файл разбит на три секции: основные настройки графического интерфейса, горячие клавиши, а также конфигурация расширений файлов для диалога открытия файлов File ► Open dialog. В этой части мы обсудим несколько интересных опций. Полный список доступных опций вы можете посмотреть в файле idagui.cfg, в большинстве случаев рядом с ними расположены комментарии, описывающие их предназначение.

IDA позволяет указать дополнительный файл справки, используя опцию HELPFILE. Учтите, что файл указанный здесь не

заменяет основной файл справки IDA. Основная задача этого файла заключается в предоставлении доступа к справочной информации, которая необходима в некоторых специфичных ситуациях реверс-инжиниринга. Если справочный файл указан, то при нажатии CTRL-F1 он будет открываться, и автоматически будет производиться поиск раздела, совпадающего со словом, которое расположено под курсором. В случае если не будет найден совпадающий раздел, то будет открыто оглавление справочного файла. Например, если не считать автоматические комментарии, IDA не предоставляет никакой справочной информации относительно мнемоник инструкций в дизассемблированном коде, поэтому во время анализа исполняемого файла для архитектуры x86 может быть удобным отображение описания инструкций x86, доступного по команде. Если у вас есть справочный файл, в котором есть разделы для каждой инструкции x86, тогда справочная информация по ним может быть открыта всего лишь по нажатию горячей клавиши. Однако стоит предупредить, что IDA поддерживает только справочные файлы старого формата (.hlp), файлы нового формата (.chm) в качестве дополнительных справочных файлов не поддерживаются.

Примечание: Существует два типа справочных файлов Windows – 16-битные и 32-битные. Microsoft Windows Vista не поддерживает 32-битные справочные файлы, так как WinHlp32.exe не включен в состав Vista. Более подробную информацию вы можете получить в статье 917607 Microsoft Knowledge Base.

Самый распространенный вопрос по поводу использования IDA это “Как можно пропатчить исполняемые файлы при помощи IDA?”, в общем, ответ “Никак”, однако более подробно мы обсудим это в главе 14. Что вы можете сделать, так это пропатчить базу данных, чтобы изменить инструкции или данные почти как угодно. После того как мы обсудим скрипты (Глава 5), вы поймете что модификация базы данных, это не так уж и сложно, но в случае если вы не заинтересованы в этом или не готовы изучать скриптовый язык IDA, то IDA содержит меню патча базы данных, которое не отображается по умолчанию. Опция DISPLAY_PATCH_SUBMENU определяет показывать ли это меню в Edit ► Patch Program. Опции, доступные в этом меню будут рассмотрены в главе 4.

После того как вы разберетесь со скриптами в IDA, у вас может появиться желание пользоваться короткими скриптовыми командами. Обычно для того чтобы запустить скрипт в IDA, вам требуется воспользоваться скриптовым диалогом доступным из меню, но вы также можете использовать опцию DISPLAY_COMMAND_LINE для того, чтобы в IDA отображалось однострочное текстовое поле,

непосредственно под окном сообщения. Это текстовое поле является «командной строкой» IDA и может использоваться для ввода однострочных скриптовых выражений. Учтите, что в этом поле вы не сможете выполнять команды операционной системы.

Секция конфигурации горячих клавиш в idagui.cfg используется для назначения действий IDA на различные сочетания клавиш. Переназначение горячих клавиш может быть полезным во многих ситуациях, которые включают в себя: назначение дополнительных команд, изменение горячих клавиш на более удобные для запоминания или для того чтобы исключить конфликты между горячими клавишами операционной системы или приложения терминала (в основном полезно при использовании текстовой версии IDA).

Теоретически, каждая опция IDA, которая доступна через меню или панель инструментов перечислена в этой секции. К сожалению, названия многих команд как правило не соответствуют названиям пунктов меню, поэтому вам придется приложить некоторые усилия для того, чтобы определить какая именно опция в файле конфигурации соответствует опции меню. Например, команда Jump►Jump to Problem соответствует опции JumpQ (которая к удивлению совпадает с горячей клавишей для нее:CTRL-Q) в idagui.cfg. В дополнение, в то время как у многих команд есть соответствующие им комментарии, у некоторых команд нет вообще никакого описания, поэтому вам остается только догадываться о том, что же делает эта команда, исходя из ее названия в файле конфигурации. Есть небольшая хитрость, которая может помочь вам выяснить какому пункту меню соответствует действие из файла конфигурации — поищите это действие в справочной системе IDA, обычно это может привести вас к описанию пункта меню, соответствующего действию.

Следующие строки являются примером назначения горячих клавиш в idagui.cfg:

```
"Abort"   =   0        // Abort IDA, don't save changes
"Quit"    =   "Alt-X"   // Quit to DOS, save changes
```

Первая строка предназначена для назначения горячей клавиши для команды Abort, в данном случае никакое сочетание не назначено, т.к. установлено значение 0 без кавычек. Во второй строке показано, что для команды Quit назначено сочетание Alt-X. Горячие клавиши указываются в виде строки с названиями клавиш заключенной в кавычки. Множество примеров назначения горячих клавиш находится в файле idagui.cfg.

В заключительной части idagui.cfg находятся соответствия описаний типов файлов и назначенных им расширений, а также определяется, какие типы файлов будут перечислены в раскрывающемся списке внутри диалога открытия файлов File ► Open. Большое количество различных типов файлов уже описано в файле конфигурации, однако, если вы часто работаете с другими типами файлов, то можете самостоятельно добавить новые типы в список. Следующая строка показывает пример назначения типа файла:

```
CLASS_JAVA,  "Java Class Files",      "*.cla*;*.cls"
```

Строка содержит три части разделенных запятой: имя ассоциации (CLASS_JAVA), описание, а также шаблон имени файла. В шаблонах имени файла допускается использование масок, а указать несколько шаблонов можно через точку с запятой. Второй тип ассоциаций позволяет сгруппировать несколько существующих ассоциаций в одну категорию. Следующая строка показывает, как группируются все ассоциации, начинающиеся с EXE_ в одну ассоциацию с именем EXE.

```
EXE,        "Executable Files",       EXE_*
```

Стоит отметить, что в данном случае шаблон не заключается в кавычки. Мы можем определить собственные файловые ассоциации вот таким образом:

```
IDA_BOOK,   "Ida Book Files",         "*.book"
```

Мы можем выбрать любое имя, которая нам нравится, лишь бы оно уже не было занято, но в любом случае просто добавления новой ассоциации в FILE_EXTENSIONS недостаточно для того, чтобы она появилось в диалоге File ► Open. Опция DEFAULT_FILE_FILTER является списком имен всех ассоциаций, которые появятся в диалоге File ► Open. Для того чтобы завершить процесс добавления новой ассоциации, необходимо добавить IDA_BOOK в список DEFAULT_FILE_FILTER.

Если вы предпочитаете не вносить изменения в файл idagui.cfg напрямую, то аналогично файлу idauser.cfg, вы можете воспользоваться файлом idauserg.cfg, ссылка на который содержится в последней строке файла idagui.cfg. Просто создайте файл idauserg.cfg и любые опции, которые вы там укажете, будут переопределять опции указанные в idagui.cfg.

1.1.3. Файл конфигурации консольной версии: idatui.cfg

Аналогом файла idagui.cfg для консольной версии является файл <IDADIR>/cfg/idatui.cfg. Этот файл очень похож по своей структуре и функциональности на файл idagui.cfg. Назначение горячих клавиш производится точно также как и в idagui.cfg. Т.к. Эти два файла очень схожи, то мы рассмотрим только их различия.

Во-первых, опции DISPLAY_PATCH_SUBMENU и DISPLAY_COMMAND_LINE не доступны в консольной версии и не включены в idatui.cfg. Диалог File ► Open используемый в консольной версии гораздо проще, чем диалог в графической версии, поэтому все файловые ассоциации доступные в idagui.cfg отсутствуют в idatui.cfg.

С другой стороны, есть несколько опций доступных только для консольной версии IDA. Например, вы можете использовать опцию NOVICE для того, чтобы запустить IDA в режиме для новичков, в котором отключены некоторые сложные функции в целях упрощения изучения IDA. Одним из самых заметных отличий в режиме для новичков, является полное отсутствие подвидов (subviews).

Обычно люди, которые пользуются консольной версией IDA, чаще используют горячие клавиши. Для облегчения автоматизации часто используемых горячих клавиш, в IDA присутствует синтаксис назначения макросов клавиатуры. Несколько примеров макросов вы можете посмотреть в файле idatui.cfg, однако, наилучшим местом для размещения разработанных вами макросов является <IDADIR>/cfg/idausert.cfg (аналог файла idauserg.cfg для консольной версии). Пример макроса, который находится в idatui.cfg выглядит примерно так (в действительности в idatui.cfg этот макрос закомментирован):

```
MACRO    "Alt-H"    // this sample macro jumps to "start" label
{
          "G"
     's' 't' 'a' 'r', 't'
"Enter"
}
```

Определения макросов должны начинаться с ключевого слова MACRO, после него должна быть указана клавиша, которая должна быть ассоциирована. Сама последовательность макроса должна быть заключена в фигурные скобки, и должна представлять собой набор ключевых имен слов и символов, которые сами могут быть последовательностью горячих клавиш. Приведенный выше макрос

активируется по сочетанию клавиш ALT-H, открывает диалог перехода по адресу (Jump to Adress) при помощи горячей клавиши G, затем вводит в диалог метку start последовательно по одному символу и далее завершает диалог нажатием клавиши ENTER. Учтите, что мы не могли использовать "start" для ввода имени символа, потому что было бы воспринято как имя горячей клавиши, и в результате привело бы к ошибке.

Примечание: Режим для новичков и макросы недоступны в графической версии IDA.

В качестве заключения об опциях конфигурационных файлов важно указать, что в случае обнаружения ошибок во время парсинга, IDA мгновенно завершает работу и выдает сообщение об ошибке, описывающее причину проблемы. Невозможно запустить IDA не устранив причину ошибки.

1.2. Дополнительные опции конфигурации IDA

В IDA есть огромное количество дополнительных опций, которые могут быть установлены при помощи интерфейса пользователя. Дополнительные опции IDA доступны при помощи меню Options и в большинстве случаев все сделанные вами изменения применяются только к текущей открытой базе данных. Значения опций сохраняются в соответствующих файлах баз данных при закрытии. Исключениями являются опции цвета (Options ► Colors) и шрифта (Options ► Font) — будучи установленными один раз они применяются ко всем последующим сессиям работы с IDA. В Windows-версии IDA значения опций хранятся в реестре в ключе HKEY_CURRENT_USER\Software\Hex-Rays\IDA. В версиях для других систем значения хранятся в доамшней директории пользователя в файле проприетарного формата с названием $HOME/.idapro/ida.cfd.

Другая часть информации хранящейся в реестре относится к настройкам диалогов, в которых вы выбрали опцию "Do not display this dialog box again". Данное сообщение иногда появляется в виде чекбокса в нижней правой части диалогов информационных сообщений, показ которых в будущем вы можете отключить. В случае если вы выберете эту опцию, то новое значение появится в ключе HKEY_CURRENT_USER\Software\Hex-Rays\IDA\Hidden Messages. Если вы захотите отменить эту опцию, чтобы убранный ранее диалог появлялся снова, то вам необходимо удалить соответствующее значение в указанном ключе реестра.

1.2.1. Цвета IDA

Цвет почти любого элемента на экране IDA может быть изменен при помощи диалога Options ► Colors, показанного на рис.1.

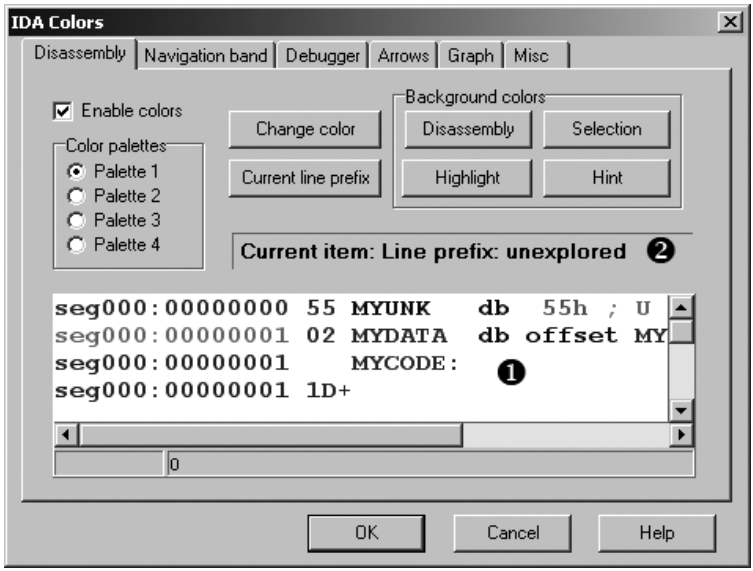


Рис.1. Диалог выбора цвета

При помощи вкладки Disassembly вы можете изменять цвета различных частей, каждой линии дизассемлированного кода. Пример того как выглядят различные типы текста дизассеблированного кода отображается в специальном окне. Когда вы выбираете в этом окне какую-либо часть строки, то ее тип отображается выше, в поле. При помощи кнопки Change color вы можете назначить любой части тот цвет, который пожелаете.

Диалог изменения цвета содержит вкладки для назначения цветов панели навигации, отладчика, стрелок переходов расположенных слева от текста дизассеблированного когда, а также различных компонент вида графа. Если точнее, то при помощи вкладки Graph вы можете изменять цвета узлов графа, их подписей и ребер, которые соединяют каждую точку, в то время как при помощи вкладки

Disassembly вы можете изменять цвета дизассемблированного кода в виде графа. Во вкладке Misc вы можете изменять цвета, используемые в окнах сообщений IDA.

1.2.2. Настройка панелей инструментов IDA

В дополнение к меню и горячим клавишам в графической версии IDA есть большое количество кнопок расположенных на более чем десяти панелях инструментов. Панели инструментов обычно «прицеплены» в области расположенной под меню IDA. Каждая панель инструментов может быть отделена и перемещена в любое место на экране по вашему вкусу. Если какая-то из панелей вам не нужна вообще, то при помощи меню View ► Toolbars (см. рис.1.1), вы можете убрать ее с экрана.

Данное меню также появляется, в случае если вы кликните правой кнопкой мыши внутри области расположенной под главным меню. Отключение панели инструментов Main убирает все панели инструментов с экрана и полезно, в случае если вам необходимо больше пространства для окна с дизассемблированным кодом. Любые изменения, которые вы производите с расположением панелей инструментов, сохраняются в текущей базе данных. Если вы откроете другую базу данных, то расположение панелей инструментов вернется в состояние, которое было установлено при сохранении этой базы данных. При открытии нового бинарного файла для создания новой базы данных, расположение панелей инструментов вернется в расположение по умолчанию в соответствии с настройками IDA.

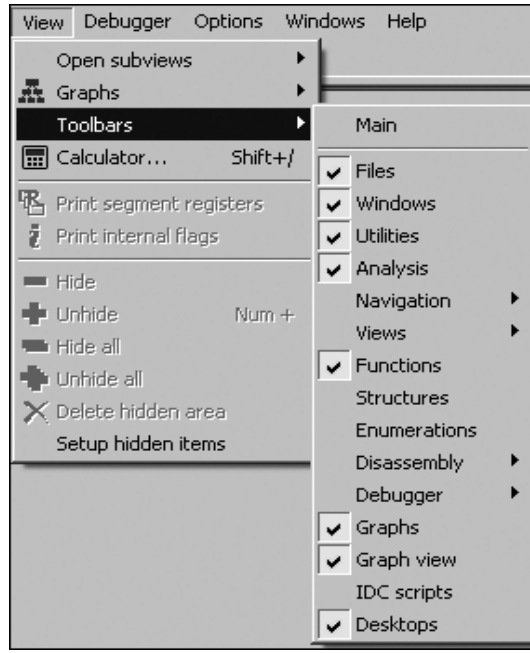


Рис.1.1. Меню настройки панели инструментов.

Если вы захотите сохранить определенное расположение панелей, и сделать его используемым по умолчанию, то вы можете воспользоваться меню Windows ► Save Desktop, которое вызывает диалог, приведенный на рис.1.2.

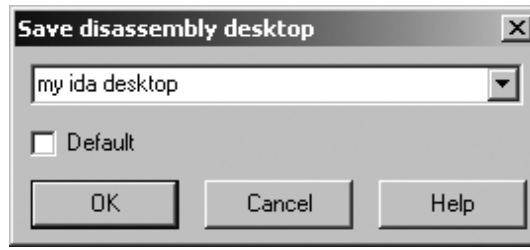


Рис.1.2. Диалог сохранения расположения элементов

Каждый раз, когда вы сохраняете конфигурацию рабочего окна, вам потребуется ввести название для данной конфигурации. В случае если вы отметите галочкой «Default», то данная конфигурация в дальнейшем будет использоваться по умолчанию во всех новых базах данных, а также при использовании сброса настроек рабочего окна Windows ► Reset desktop. Для того чтобы загрузить одну из конфигураций, сохраненных вами ранее, воспользуйтесь меню Windows ► Load Desktop и выберите нужную конфигурацию. Сохранение и загрузка конфигураций рабочего окна полезно, в случае если вы используете несколько мониторов различного размера или с различными разрешениями (например, при подключении монитора к ноутбуку или при использовании проектора).

ГЛАВА 2. РАСПОЗНАВАНИЕ БИБЛИОТЕК ПРИ ПОМОЩИ СИГНАТУР FLIRT

С данного момента мы перейдем к углубленному изучению возможностей IDA и разберемся, что же делать после сообщения "The initial autoanalysis has been finished". В данной главе мы обсудим различные методы для распознавания стандартных кодовых последовательностей, таких как код библиотек содержащийся в статически связанных бинарных файлах или стандартной инициализации, а также вспомогательных функций добавляемых компиляторами.

Когда вы собираетесь приступить к реверс-инжинирингу бинарного файла, то вам вряд ли захочется заниматься также и реверс-инжинирингом библиотечных функций, потому что гораздо проще разобраться с их работой путем чтения руководств, просмотра исходного кода или даже поиска в интернете. Единственной проблемой является то, что в бинарных файлах со статическим связыванием граница между кодом приложения и кодом библиотеки размыта из-за того что они объединены в один монолитный исполняемый файл. К счастью, в IDA есть инструменты для распознавания и выделения библиотечного кода, благодаря им мы можем сосредоточиться на уникальном коде внутри приложения.

2.1. Технология быстрого определения и распознавания библиотек (Fast Library Identification and Recognition Technology)

Для распознавания последовательностей кода библиотек в IDA используется технология быстрой идентификации и распознавания

библиотек, более известная как FLIRT. В основе FLIRT лежат алгоритмы сравнения шаблонов, которые позволяют IDA быстро определять, какие из дизассемблированных функций совпадают с известными сигнатурами. Файлы сигнатур, которые поставляются вместе с IDA, хранятся в папке <IDADIR>/sig. Большая часть — это сигнатуры библиотек, которые поставляются с распространенными Windows-компиляторами, хотя также присутствуют сигнатуры и из других ОС.

Файлы сигнатур хранятся в особом формате, в котором большая часть данных является сжатой и обернутой в специальный заголовок IDA. В большинстве случаев название файла сигнатур не полностью отражает, для какой конкретно библиотеки он был сгенерирован. В зависимости от способа создания, в некоторых файлах сигнатур может содержаться комментарий с именем библиотеки. Если посмотреть на несколько первых строк ASCII-содержимого файла сигнатур, то чаще всего можно обнаружить подобный комментарий. Следующая Unix-подобная команда позволяет обнаружить комментарий во второй или третьей строке вывода:

В IDA существует два способа просмотреть комментарии, связанные с файлами сигнатур. Первый способ - вы можете открыть список сигнатур, примененных к бинарному файлу через меню View ► Open Subviews ► Signatures. Второй способ — вы можете просмотреть их во время процесса ручного применения сигнатур, который запускается при помощи меню File ► Load File ► FLIRT Signature File.

2.2. Применение сигнатур FLIRT

Во время первого открытия бинарного файла IDA пытается применить специальные файлы сигнатур, назначенных как сигнатуры запуска (startup signature), к точке входа бинарного файла. В действительности происходит так, что точки входа, создаваемые разными компиляторами, существенно отличаются, поэтому определение подходящей точки входа полезно для определения компилятора, который использовался для создания данного бинарного файла.

Напоминаем, что точка входа программы это адрес инструкции, которая будет выполнена первой. Множество С-программистов ошибочно считают, что это адрес функции main, но это не так. Как программа принимает аргументы командной строки, зависит от типа файла программы, а не от языка ее разработки. Для того чтобы программа могла получить аргументы, несмотря на различия в методе их передачи загрузчиком и форме в которой их ожидает

программа, должна быть выполнена некоторая инициализация, код которой запускается до передачи управления функции main. Именно эта инициализация назначается как точка входа программы и называется _start.

Код инициализации также отвечает за любые задачи, которые должны быть выполнены до запуска main. В программах, написанных на языке C++, этот код отвечает за проверку того, что конструкторы глобально определенных объектов вызываются до запуска main. Аналогично вызывается код очистки, после завершения функции mainдля того, чтобы вызвать деструкторы для глобальных объектов до завершения программы

Если IDA удается, определить какой компилятор, использовался для создания определенного бинарного файла, то загружаются определенные файлы сигнатур для соответствующих библиотек компилятора и далее применяются к бинарному файлу. Сигнатуры, которые поставляются вместе с IDA, обычно относятся к проприетарным компиляторам, таким как Microsoft Visual C++ или Borland Delphi. Такой выбор обусловлен тем, что с данными компиляторами поставляется большое количество библиотек. В случае со свободно распространяемыми компиляторами, такими как GNU gcc, проблема в том, что существует слишком много различных версией библиотек, также как много и операционных систем с которыми поставляется компилятор. Например, каждая версия FreeBSD поставляется вместе с уникальной версией стандартной библиотеки C. Для того, чтобы определение работало, необходимо создавать файлы сигнатур для каждой версии библиотеки. Учитывая то количество различных версий libc.a, которые поставляются вместе с различными дистрибутивами Linux, это было бы просто не практично. Отчасти проблема в изменениях исходного кода библиотек, что приводит к получению иного скомпилированного кода, но также огромное количество различий получается вследствие использования различных опций оптимизации и использования различных версий компиляторов. Из-за вышеперечисленногоIDA поставляется с очень малым набором сигнатур библиотек для свободно распространяемых компиляторов. Но есть и хорошие новости — в скором времени вы увидите, что Hex-Rays делает доступными инструменты для самостоятельного создания сигнатур из статических библиотек.

Итак, в каких же все-таки случаех вам может потребоваться самостоятельно применять сигнатуры к одной из своих баз данных? Обычно IDA достаточно точно определяет использованный для создания бинарного файла компилятор, однако сигнатуры для него могут отсутствовать. В таком случае, у вас есть выбор либо обойтись

без них вообще, либо найти версию статической библиотеки, которая использовалась при создании бинарного файла, и создать собственные сигнатуры. Другой случай — когда IDA не удалось определить компилятор, а,следовательно, невозможно выбрать какие сигнатуры использовать. Это часто может встречаться в ситуациях, когда вы работаете с обфусцированным кодом, в котором процедура запуска была существенно изменена для того, чтобы предотвратить определение компилятора. В подобном случае первое, что стоит сделать — использовать де-обфускатор, может быть, в таком случае появится возможность найти подходящие сигнатуры.

В любом случае, вы можете загрузить сигнатуры вручную при помощи меню File ► Load File► FLIRT Signature File, которое вызовет диалог выбора сигнатуры, показанный на рис.2.

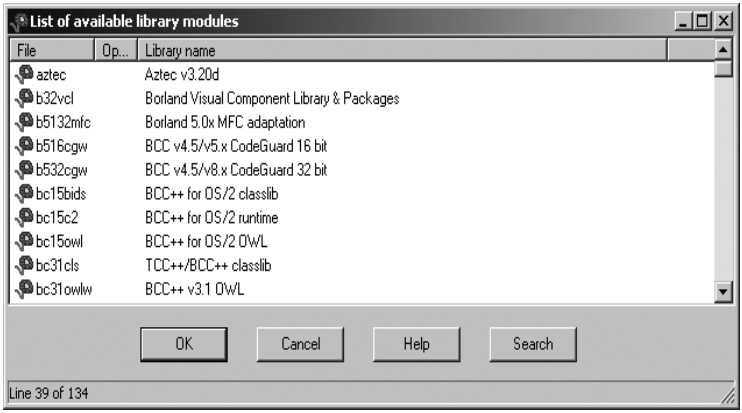


Рис.2. Выбор FLIRT сигнатур.

В столбце File находятся имена всех файлов с расширением .sig, находящихся в директории <IDADIR>/sig. Заметим, что нет необходимости указывать дополнительные пути для сигнатур. Если вы когда-нибудь создадите свои сигнатуры, то вам стоит просто поместить их в <IDADIR>/sig вместе с остальными .sig-файлами. Столбец Library name отображает название библиотеки, которое хранится в каждом файле сигнатур. Учтите, что данное значение задается создателем файла сигнатур (а им можете быть и вы!) по его желанию.

Когда вы выбираете модуль библиотеки, сигнатурысодержащиеся в соответствующем .sig-файле загружаются и

сравниваются со всеми функциями в базе данных. Только один набор сигнатур может быть загружен за раз, поэтому если вы хотите применить несколько различных файлов сигнатур, то вам необходимо повторять процесс загрузки для каждого файла в отдельности. В случае обнаружения соответствия функции сигнатуре, она автоматически отмечается как библиотечная функция, и переименовывается соответственно совпавшей сигнатуре.

Предупреждение: Только функции, которым присвоено стандартное имя-заглушка, могут быть автоматически переименованы. Иными словами, если вы переименовали функцию, и после этого она совпала с сигнатурой, то ее имя не будет изменено. Поэтому важно применять сигнатуры как можно раньше во время начала анализа.

Напоминаем, что при использовании статически связанных библиотек, граница между их кодом и кодом приложения размывается. Если вам посчастливилось получить библиотеку, в которой символы не были перемешаны, то как минимум у вас будут полезные имена функций (настолько полезные, насколько ответственный программист их назначал), который помогут вам разобраться с кодом. В любом случае, если в бинарном файле все было перемешано в кучу, у вас, скорее всего, будут сотни функций с автоматически заданными IDA именами и без обозначения, что каждая из них делает. В обоих случаях IDA сможет определить библиотечные функции только, если доступны сигнатуры (имена функций сами по себе не несут достаточной информации для определения их предназначения). На рис.2.1. показан Overview Navigator для статически связанной библиотеки.

Рис.2.1. Статическое связывание без сигнатур

На этом экране нет функций, которые были бы определены как библиотечные функции, вам может потребоваться разбирать гораздо больше кода, чем вы собирались. После применения сигнатур все будет выглядеть несколько иначе, как показано на рис.2.2.

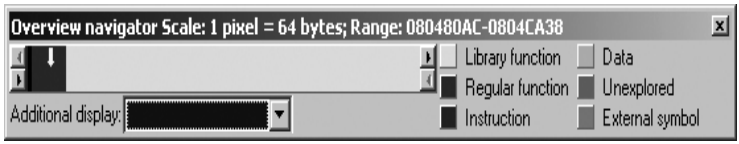


Рис.2.2. Статическое связывание с примененными сигнатурами

Как вы можете увидеть, окно Overview Navigator отлично показывает, насколько эффективным был определенный набор сигнатур. В случае высокого процента совпадения с сигнатурами, большие части кода будут отмечены как библиотечные, и соответственно переименованы. В примере, приведенном на рис.2.2, код приложения, скорее всего, расположен в крайней левой части экрана.

Есть две важные вещи, которые стоит помнить во время применения сигнатур. Во-первых, сигнатуры полезны при работе с бинарным файлом, в котором код не был «обрезан», в данном случае сигнатуры используются больше для определения назначения функций, чем простого их переименования. Во-вторых, статически связанные бинарные файлы могут состоять из нескольких различных библиотек, поэтому может потребоваться применение нескольких сигнатур. После каждого применения сигнатуры экран Overview Navigator будет изменяться для отражения степени определения библиотечного кода. На рис.2.3. отображен пример, в котором бинарный файл статически связан со стандартной библиотекой C и криптографической библиотекой OpenSSL.

Рис.2.3. Статический бинарный файл с одним набором сигнатур

В данном случае были применены сигнатуры для версии OpenSSL, использованной в данном приложении. Небольшой участок (светлый участок слева) был отмечен как библиотечный код. Бинарные файлы со статическим связыванием часто создаются путем помещения кода приложения в начале и расположение кода библиотек после. Посмотрев на данный рисунок, мы может придти к выводу, что область памяти справа от библиотеки OpenSSL, скорее всего, занята кодом

других библиотек, в то время как код приложения расположен левее библиотеки OpenSSL. Если мы продолжим применять сигнатуры к данному бинарному файлу, то в результате мы получим то, что показано на рис.2.4.

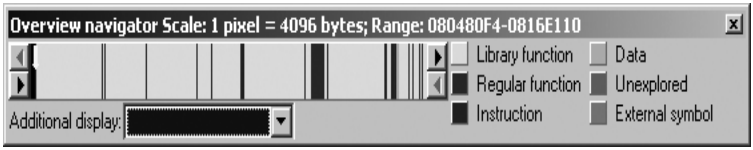


Рис.2.4. Статический бинарный файл с несколькими наборами сигнатур

В данном случае мы применили сигнатуры библиотек libc, libcrypto, libkrb5, libresolv и прочих. В некоторых случаях мы выбирали сигнатуры исходя из текстовых строк, найденных в бинарном файле, в иных случаях мы выбирали библиотеки, которые тесно связаны с уже ранее найденными библиотеками. Однако на экране все еще остается два больших темных участка, в крайней левой части и немного правее середины. Для того чтобы определить из чего состоят эти оставшиеся части файла, необходимо разбираться далее. В данном случае мы можем определить, что участок справа это какая-то неопределенная библиотека, а участок слева — код приложения.

2.3. Создание файлов сигнатур FLIRT

Как мы уже сказали ранее — было бы непрактично, если бы с IDA поставлялись файлы сигнатур для всех существующих библиотек. Для того чтобы предоставить пользователям IDA инструменты и необходимую информацию для создания собственных сигнатур Hex-Rays распространяет набор инструментов под названием Fast Library Acquisition for Identification and Recognition (FLAIR). FLAIR поставляется вместе с IDA на CD, также он доступен для скачивания с сайта Hex-Rays для авторизованных пользователей. Как и другие дополнения IDA, FLAIR распространяется в виде Zip-файла. Для IDA версии 5.2 соответствующий набор инструментов FLAIR содержится в файле flair52.zip. Новые версии FLAIR появляются реже, чем новые версии IDA, поэтому вам стоит использовать последнюю версию FLAIR, которая не превышает вашу версию IDA.

Установка FLAIR представляет собой обычную распаковку Zip-файла, однако мы настоятельно рекомендуем создать отдельную

папку для инсталляции, так как файлы в Zip-архиве не собраны в одну папку верхнего уровня. В дистрибутиве FLAIR вы найдете несколько текстовых файлов, которые представляют собой документацию для набора инструментов FLAIR. Наиболее значимыми из них являются:

readme.txt. В данном файле описывается процесс создания сигнатур.

plb.txt. Этот файл содержит описание использования парсера статических библиотек plb.exe. Более подробно парсеры библиотек рассмотрены в секции 2.3.3.

pat.txt. В данном файле уточняются особенности формата файлов шаблоном, которые являются первым пунктом в процессе создания сигнатур. Файлы шаблоном также рассмотрены в секции 2.3.3.

sigmake.txt. Здесь описывается использование sigmake.exe для создания .sig-файлов из файлов шаблонов. Подробности в секции 2.3.4.

В папке bin содержатся все исполняемые файлы из набора инструментов FLAIR.

В папке startup содержатся файлы шаблонов для распространенных последовательностей запуска различных компиляторов и связанных с ними типов файлов (PE, ELF и т.д.). Важно заметить, что, FLAIR состоит из инструментов, работающих только под Windows, получившиеся файлы сигнатур могут использоваться с версиями IDA для любых операционных систем (Windows, Linux и OS X).

2.3.1. Обзор процесса создания сигнатур

Процесс создания сигнатур обычно не выглядит сложным и может быть представлен четырьмя пунктами:

- Найдите файл статической библиотеки, для которого вы хотите создатьфайл сигнатур.
- Воспользуйтесь одним из парсеров FLAIR для того, чтобы создать файл шаблона для библиотеки.
- Запустите sigmake.exe для того, чтобы обработать получившийся файл шаблона и сгенерировать файл сигнатур.
- Добавьте новый файл сигнатур в IDA, скопировав его в <IDADIR>/sig.

К сожалению, на практике только последний пункт выполняется так же легко, как и выглядит. В следующих разделах мы рассмотрим первые три шага более подробно.

2.3.2. Определение и приобретение статических библиотек

Первым пунктом в процессе создания сигнатур, является нахождение файла статической библиотеки, для которой вы хотите эти сигнатуры создать. Это может быть довольно трудно по множеству причин. Первым препятствием является определение того, какая же именно вам нужна библиотека. Если бинарный файл не подвергся «удалению лишнего», то вам может повезти, и вы найдете названия функций в дизассемблированном коде, по которым в Google вы сможете найти подходящие вам библиотеки.

Однако не всегда в бинарных файлах информация об их происхождении может быть доступна так просто. В случае отсутствия имен функций, можно попробовать поискать по текстовым строкам, которые могут оказаться достаточно уникальны, и позволят определить библиотеку, как например, явно видно в следующем примере:

OpenSSL 0.9.8a 11 Oct 2005

Замечания о копирайте и строки ошибок также могут быть достаточно уникальны для того, чтобы воспользоваться их поиском в Google. Если вы будете пользоваться командой string из командной строки, не забывайте использовать опцию -a, которая запускает принудительный поиск по всей библиотеке, в противном случае вы можете пропустить потенциально важную информацию.

В случае если вы работаете с библиотекой с открытым исходным кодом, вы сможете легко данный код найти. К сожалению, несмотря на то, что исходный код может быть полезен для понимания поведения бинарного файла, вы не сможете использовать его для создания файла сигнатур. Конечно, можно воспользоваться исходным кодом для компиляции собственной версии библиотеки и затем создания файла сигнатур. Однако, скорее всего вам не удастся воспроизвести процесс компиляции абсолютно идентично, поэтому полученная библиотека будет сильно отличаться от анализируемой вами библиотеки, и сгенерировать точные сигнатуры будет невозможно.

Лучше всего попытаться определить точное происхождение бинарного файла. Под этим мы подразумеваем точное определение операционной системы, версии операционной системы, дистрибутива. В случае владения такой информацией, наилучшим вариантом будет скопировать библиотеку из точно также настроенной операционной системы. Здесь, собственно, появляется первый вопрос, как же определить в какой ОС был создан бинарный файл? Для этого

рекомендуем воспользоваться файловой утилитой для получения базовой информации о файле. Мы наблюдали примерный вывод информации из файла. Ниже находится еще один пример вывода из файла:

```
$ file sample_file_1
sample_file_1: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD),
for FreeBSD 5.4, statically linked, FreeBSD-style, stripped
```

В данном случае мы можем сразу же перейти к системе FreeBSD 5.4 и достать libc.a из нее. А вот в следующем примере все не так прямолинейно:

```
$ file sample_file_2
sample_file_2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.9, statically linked, stripped
```

То, что файл был создан в системе Linux видно сразу же, однако этого недостаточно, так как количество дистрибутивов Linux огромно. Поискав далее по строкам, мы можем обнаружить следующее:

```
GCC: (GNU) 4.1.1 20060525 (Red Hat 4.1.1-1)
```

Здесь мы уже можем сократить поиск до дистрибутивов Red Hat (либо производных от них) с gcc версии 4.1.1. Метки GCC встречаются довольно часто в скомпилированных в данном компиляторе бинарных файлах и, к счастью для нас, они не удаляются во время процесса очищения(stripping), и остаются доступными.
Помните, что файловая утилита это не единственный способ для идентификации файла. В следующем примере показывается простой случай, когда тип файла известен, но информация не конкретизирована:

```
$ file sample_file_3
sample_file_3: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
```

Данный пример был сделан в системе Solaris 10 x86. Опять же, это было выяснено при помощи утилиты strings.

2.3.3. Создание файлов шаблонов

Для того чтобы приступить к выполнению данного пункта у вас уже должны быть библиотеки, для которых вы хотите сгенерировать сигнатуры. Данный пункт заключается в создании файла шаблона для каждой библиотеки. Файлы шаблонов создаются при помощи парсера из набора инструментов FLAIR. Также как и исполняемый файлы, файлы библиотек могут быть различных форматов. В наборе инструментов FLAIR есть парсеры для нескольких популярных форматов. В файле readme.txt описано, какие парсеры вы можете обнаружить в папке bin:

plb.exe - парсер для библиотек OMF (используется компиляторами Borland)
pcf.exe - парсер для библиотек COFF (используется компиляторами Microsoft)
pelf.exe - парсер для библиотек ELF (используются во многих Unix-системах)
ppsx.exe - парсер для библиотек Sony PlayStation PSX
ptmobj.exe - парсер для библиотек TriMedia
pomf166.exe - парсер для файлов объектов Kiel OMF 166

Для того чтобы создать файл шаблона для библиотеки, укажите соответствующий ее формату парсер, имя библиотеки и имя файла шаблона, который будет сгенерирован. Для библиотеки libc.a из системы FreeBSD 6.1 вы можете воспользоваться следующей командой:

```
$ ./pelf libc.a libc_FreeBSD61.pat
libc.a: skipped 0, total 986
```

Парсер выводит название использовавшегося файла, количество пропущенных функций (0) и количество шаблонов сигнатур, которые были сгенерированы (986). Опции командной строки у различных парсеров немного отличаются. Для того чтобы узнать их, достаточно просто запустить парсер без опций. В файле plb.txt содержится более подробная информация о парсере plb.exe. Этот файл является хорошим источником основной информации, так как многие из описанных опций также принимаются и другими парсерами. Для того, чтобы сгенерировать файл шаблона, достаточно указать название библиотеки и название результирующего файла.

Парсеры plb и pcf могут пропускать некоторые функции в зависимости от опций командной строки и от структуры файла библиотеки.

Файл шаблонов представляет собой текстовый файл, в котором в каждой строке по одному расположены шаблоны извлеченных функций. Далее приведены несколько строк из файла шаблонов созданного ранее:

```
5589E58B55108B450C8B4D0885D2EB06890183C1044A75F88B4508C9C
3..... 00 0000 001D :0000 _wmems
et
5589E58B4D1057C1E102568B7D088B750CFCC1E902F3A55E8B45085F
C9C3.... 00 0000 001E :0000 _wmemc
py
5589E556538B751031DB39F38B4D088B550C73118B023901751183C104
83C204 19 A9BE 0039 :0000 _wmemc
mp
```

Формат шаблонов описан в файле FLAIR.txt. Если вкратце, то в первой части шаблона записана начальная последовательность байт функции, максимум 32 байта. Есть некоторые байты, которые могут изменяться в результате перемещения записей. Такие байты отображаются двоеточием. Также они используются для дополнения шаблона до 64 символов, в случае если функция короче 32 байт (как функция _wmemset в предыдущем примере). После начальных 32 байт располагается дополнительная информация для более точного сравнения сигнатур. В каждой строке шаблона также присутствует контрольная сумма CRC16, которая вычисляется из части функции, длины функции и списка символьных имен на которые ссылается функция. В основном, для длинных функций, которые ссылаются на большое количество символьных имен, получаются очень длинные шаблоны. В ранее сгенерированном файле libc_FreeBSD61.pat некоторые строки длиннее 20000 символов.

Два символа на один байт, для отображения 32 байт требуется 64 шестнадцатеричных символа.

Это 16-битное контрольное значение. Реализация CRC16 используемая при генерации шаблонов, включена в набор инструментов FLAIR в файле crc16.cpp.

Несколько сторонних программистов создали утилиты, предназначенные для генерации шаблонов из баз данных IDA (ing). Одной из таких утилит является модуль для IDA с названием IDB_2_PAT написанный Дж. С. Робертсом, модуль позволяет генерировать шаблоны для одной или сразу нескольких функций из базы данных. Подобные утилиты полезны, в случае если вы ожидаете встретить сходный код в других базах данных, и не имеете доступа к

оригинальным файлам библиотек, которые использовались при создании анализируемого бинарного файла.

2.3.4. Создание файлов сигнатур

После того, как вы создали файл шаблонов, следующим пунктом будет процесс создания сигнатур для генерации файла .sig подходящего для IDA. Формат файла сигнатур IDA существенно отличается от формата файла шаблонов. В файлах сигнатур используется проприетарный формат, предназначенный для того, чтобы минимизировать занимаемое информацией из файла шаблонов место и сделать сравнение сигнатур более эффективным. Описание формата файла сигнатур доступно на сайте Hex-Rays.

Для создания файла сигнатур из файла шаблонов используется утилита sigmake.exe из FLAIR. Разделение процессов генерации сигнатур и шаблонов позволяет добиться их полной независимости, что в свою очередь позволяет использовать сторонние парсеры. В простейшем виде, генерация сигнатур представляет собой использование sigmake.exe для парсинга .pat-файла, и создания .sig-файла как показано ниже:

```
$ ./sigmake libssl.pat libssl.sig
```

В случае если все прошло успешно, будет создан .sig-файл готовый к помещению в папку <IDADIR>/sig. Однако процесс генерации сигнатур редко проходит гладко.

Примечание: В файле документации sigmake.txt рекомендуется использовать для файлов сигнатур короткие имена в формате MS-DOS 8.3. На самом деле это не обязательно, просто в случае использования длинных имен файлов в диалоге выбора сигнатур будут отображаться только первые 8 символов.

Генерация сигнатур часто является итеративным процессом, и все возможные коллизии должны быть решены во время текущей фазы. Одной из коллизий может быть то, что две функции имеют идентичные шаблоны. Если не решить эту проблему, то во время сравнения сигнатур будет невозможно определить, какая же была использована функция. Поэтому sigmake должна иметь возможность привести каждую сигнатуру только к одной функции. В случае если это невозможно из-за присутствия совпадающих шаблонов, sigmake не сгенерирует .sig-файл, вместо этого будет сгенерирован файл исключений .exc. Довольно часто первое использование sigmake над файлом шаблонов может вывести следующее:

```
$ ./sigmake libc_FreeBSD61.pat libc_FreeBSD61.sig
See the documentation to learn how to resolve collisions.
: modules/leaves: 13443631/970, COLLISIONS: 911
```

Под документацией в данном случае подразумевается sigmake.txt, в котором описывается использование sigmake и процесс разрешения коллизий. В действительности, каждый раз, когда запускается sigmake, производится поиск соответствующего файла исключений, в котором может содержаться информация о разрешении коллизий, которые могут встретиться во время обработки файла шаблонов. В случае отсутствия файла исключений и появления коллизии, sigmake сгенерирует подобный файл вместо файла сигнатур. В предыдущем примере мы бы обнаружили только что созданный файл libc_FreeBSD61.exc. В первый раз файл исключений создается как текстовый файл, в котором детально описаны конфликты, встреченные sigmake во время обработки файла шаблонов. Для того чтобы предоставить sigmake информацию для разрешения конфликтов шаблонов, необходимо отредактировать файл исключений.

Все сгенерированные sigmake файлы исключений начинаются со следующих строк:

```
;--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules
```

Назначение этих строк заключается в том, чтобы напомнить, как разрешать коллизии. Важным является удалить строки, начинающиеся с точки с запятой, иначе sigmake не удастся выполнить парсинг файла исключений во время следующего запуска. В качестве следующего шага необходимо указать, каким образом вы хотите разрешить коллизии. Вот несколько строк из libc_FreeBSD61.exc:

```
___ntohs          00 0000 0FB744240486C4C3.....
..
___htons          00 0000 0FB744240486C4C3..... .....
..

_index                                                    00      0000
538B4424088A4C240C908A1838D974074084DB75F531C05BC3.....
..
```

```
_strchr                                                    00      0000
538B4424088A4C240C908A1838D974074084DB75F531C05BC3.....
..

_rindex                                                    00      0000
538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3.....
...
..
_strrchr                                                   00      0000
538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3.....
...
```

В данных строках указываются три различных коллизии. В текущем случае нам сообщается, что фукнция ntohs неотличима от htons, у index такая же сигнатура как у strchr, а между rindes и strrchr коллизия. Если вы знакомы с этими функция, то это вряд ли вас удивит, так как эти функции по существу идентичны (например, inder и strchr выполняют одно и то же действие).

Вы сами контролируете процесс, поэтому вы должны указать для sigmake по одной функции ассоциированной с сигнатурой в каждой группе. Функцию можно выбрать, поставив перед ее именем плюс (+), если вы хотите чтобы она соответствовала сигнатуре в базе данных или минус (-), если вы просто хотите чтобы каждый раз при совпадении сигнатуры добавлялся комментарий. В случае если вы не хотите чтобы применялись какие-от имена, то просто не добавляейте никаких символов. В следующих строках отражается один из возможных путей разрешения вышеперечисленных коллизий:

```
+___ntohs          00 0000 0FB744240486C4C3.....
..
___htons           00 0000 0FB744240486C4C3.....
..

_index                                                     00      0000
538B4424088A4C240C908A1838D974074084DB75F531C05BC3.....
..
_strchr                                                    00      0000
538B4424088A4C240C908A1838D974074084DB75F531C05BC3.....
..

_rindex                                                    00      0000
538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3.....
...
```

..
-_strrchr 00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3.....
...

В данном случае мы выбрали, чтобы использовалось имя ntohs каждый раз, когда совпадает сигнатура, а также чтобы добавлялся комментарий о strrchr каждый раз тогда когда совпадает третья сигнатура. Перечисленные ниже пункты полезны при разрешении коллизий:

- Для минимального разрешения коллизий, просто удалите первые четыре закомментированные строки в начале файла исключений.
- Никогда не добавляйте больше одного плюса или минуса в каждой отдельной группе коллизий.
- Если в группе коллизий находится только одна функция, не добавляйте к ней плюс или минус, просто проигнорируйте ее.

В случае неудачного срабатывания sigmake к файлам исключений добавляются данные, включая комментарии. Эти данные нужно удалять, и корректировать оригинальные данные (если данные были правильными изначально, то sigmake сработает нормально во второй раз) перед запуском sigmake.

После того как вы сделаете все необходимые изменения в файле исключений, вы должны сохранить его и заново запустить sigmake, используя те же опции командной строки, что использовались изначально. Во второй раз sigmake должен определить файл исключений и воспользоваться им и успешно создать .sig-файл. В случае подобного успешного завершения вы не увидите сообщений об ошибках, зато появится .sig-файл:

```
$ ./sigmake libc_FreeBSD61.pat libc_FreeBSD61.sig
```

После того как был успешно сгенерирован файл сигнатур вы можете скопировать его в папку <IDADIR>/sig. После этого новые сигнатуры станут доступны через меню File ► Load File ► FLIRT Signature File.

Заметим, что мы пропустили множество опций доступных как для парсеров шаблонов, так и для sigmake. Список опций вы можете посмотреть в файлах plb.txt и sigmake.txt. Единственная опция, о которой мы хотели бы упомянуть это опция -n для sigmake. Эта опция позволяет добавить имя-описание в файл сигнатур. Это имя отображается во время выбора сигнатур и может быть полезно при

сортировке списка доступных сигнатур. Следующая команда добавляет строку имени "FreeBSD 6.1 C standard library" в генерируемый файл сигнатур:

```
$ ./sigmake -n"FreeBSD 6.1 C standard library" libc_FreeBSD61.pat libc_FreeBSD61.sig
```

Также имена библиотек могут указываться в файлах исключений. Однако не факт что файл исключений понадобится во время создания сигнатур, поэтому опция командной строки может быть более полезна. Для более подробной информации обратитесь к sigmake.txt.

2.3.5. Стартовые сигнатуры

IDA может распознавать особую форму сигнатур, называемую стартовыми сигнатурами. Стартовые сигнатуры применяются, когда бинарный файл загружается в базу данных первый раз для определения, использованного компилятора. Если IDA сможет определить использованный компилятор, то дополнительный сигнатуры, ассоциированные с ним, будут загружены автоматически во время начального анализа бинарного файла.

Исходя из того, что изначально тип компилятора неизвестен при первой загрузке файла, стартовые сигнатуры сгруппированы и выбираются исходя из типа бинарного файла. Например, если загружается бинарный файл Windows PE, то будут применены соответствующие сигнатуры для того чтобы определить какой компилятор использовался для создания подобного файла.

Для того чтобы сгенерировать стартовые сигнатуры, sigmake обрабатывает шаблоны, которые описывают стартовые последовательности, генерируемые различными компиляторами и группирует их в один специальный файл сигнатур. Папка startup в наборе инструментов FLAIR содержит стартовые шаблоны, используемые в IDA, а также скрипт startup.bat используемый для создания соответствующих сигнатур из шаблонов. Примеры создания старторвых сигнатуры вы можете посмотреть в startup.bat

В случае если вам приходится работать с PE-файлами, вы можете заметить несколько файлов pe_*.pat в папке startup, описывающих стартовые шаблоны используемые популярными Windows-компиляторами, включая pe_vc.pat для Visual Studio и pe_gcc.pat для Cygwin/gcc. Если вы хотите добавить дополнительные стартовые шаблоны для PE-файлов, то вам нужно либо добавить их в

существующие файлы, либо создать новый файл шаблонов с префиксом pe_ для того, чтобы скрипт создания стартовых сигнатур мог корректно найти шаблоны, и совместить ихс существующими.

Последнее замечание по поводу стартовых шаблонов касается их формата, который несколько отличается от шаблонов для библиотечных функций. Разница между ними в том, что в стартовых шаблонах могут указываться дополнительные сигнатуры, которые нужно применить в случае совпадения. Кроме примеров стартовых шаблонов в папке startup больше нет никакого описания их формата в текстовых файлах, включенных в FLAIR.

ГЛАВА 3. РАСШИРЕНИЕ БАЗЫ ЗНАНИЙ IDA

Мы полагаем на данный момент,очевидно, что качественное дизассемблирование включает в себя больше чем просто набор мнемоник и операндов извлеченных из последовательностей байт. Для того чтобы дизассемблирование приносило пользу, необходимо также пользоваться информацией полученной во время изучения API, а точнее таких вещей как прототипы функций и стандартных типов данных. В этой главе мы будем изучать утилиты idsutils и loadint. Данные утилиты доступны на диске IDA либо могут быть загружены с сайта Hex-Rays.

3.1. Вставка информации о функциях

IDA получает знание о функциях из двух источников: библиотек типов (type library, .til) и файлов утилиты IDS (.ids). Во время начального этапа анализа, IDA использует информацию хранящуюся в данных файлах для того, чтобы повысить точность дизассемблирования и сделать код более читаемым. Это достигается при помощи включения имен параметров функций и типов, а также комментариев, которые идут с различными библиотечными функциями.

Ранее мы обсуждали файлы библиотек типов (type library files) и механизм, при помощи которого IDA хранит расположение сложных структур данных. Эти файлы также используются в качестве средств, которые IDA испльзует для записи информации о том, как вызывать функцию и последовательность ее параметров. IDA использует информацию из сигнатур функций несколькими путями. Во-первых, когда в бинарном файле используются разделяемые библиотеки, IDA не

может узнать какие особенности вызова могут применяться для этих функций. В таких случаях IDA пытается сравнить библиотечные функции с ассоциированными с ними сигнатурами в файле библиотеки типов. Если подходящая сигнатура успешна,найдена, то IDA может понять, как вызывается функция и сделать некоторые изменения к указателю стека в случае необходимости (напоминаем, что функции stdcall выполняют очистку стека самостоятельно). Второй путь испсплользования сигнатур функций заключается в том, чтобы прокомментировать параметры, передаваемые функции для обозначения какой именно параметр вдвигается в стек до вызова функции. Количество информации представленной в комментарии зависит от количества информации обнаруженной IDA во время парсинга сигнатуры функции. Ниже приведены две сигнатуры,являющиеся корректными объявления C (legal C declarations), во второй функции больше информации, так как в ней предоставляется имя параметра в дополнение к типам данных.

```
LSTATUS _stdcall RegOpenKey(HKEY, LPCTSTR, PHKEY);
LSTATUS _stdcall RegOpenKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult);
```

Библиотеки типов IDA содержат сигнатуры для функций большого количества распространенных API, включая значительную часть Windows API. Стандартный дизассемблированный код вызова функции RegOpenKey выглядит так:

```
.text:00401006  00C   lea   eax, [ebp+hKey]
.text:00401009  00C   push  eax          ; phkResult
.text:0040100A  010   push  offset SubKey  ; "Software\\Hex-Rays\\IDA"
.text:0040100F  014   push  80000001h    ; hKey
.text:00401014  018   call  ds:RegOpenKeyA
.text:0040101A  00C   mov   [ebp+var_8], eax
```

Заметьте, что IDA добавлены комментарии в правой части, отображающие, какой параметр передается в каждой инструкции, ведущей к вызову RegOpenKey. В случае если в сигнатуре функции присутствуют формальные имена параметров, то IDA попытается пойти еще дальше, и автоматически добавит имена переменных, соответствующие определенным параметрам. В двух вышеприведенных случаях видно, что присвоено имя одной локальной переменной (hKey) и одной глобальной (SubKey), на основании их соответствия формальным параметрам в прототипе RegOpenKey. Если

во время парсинга у прототипа функции были обнаружены только информацию о типах без формальных имен параметров, то тогда в предыдущем примере отображались бы имена типов данных соответствующих аргументов вместо имен параметров. В случае с параметром lpSubKey имя параметра не отображается, так как параметр указывает на строковую переменную, поэтому отображается содержимое строки. В завершение заметим, что RegOpenKey была распознана IDA как функция stdcall, поэтому автоматически был скорректирован указатель стека. Вся данная информация была получена из сигнатуры функции, которую IDA также отображает как комментарий в дизассемблированном коде в соответствующем месте в таблице импорта, как показано в листинге ниже:

```
.idata:0040A000 ; LSTATUS __stdcall RegOpenKeyA(HKEY hKey,
LPCSTR lpSubKey, PHKEY phkResult)
.idata:0040A000                 extrn RegOpenKeyA:dword ; CODE XREF:
_main+14p
.idata:0040A000                                 ; DATA XREF: _main+14r
```

Комментарий, отображающий прототип функции был взят IDA из файла .til, содержащего информацию о функциях Windows API.

В каком случае вам может потребоваться сгенерировать свои собственные сигнатуры типов параметров функций? В любом случае когда вам встречается бинарный файл, динамически или статически связанный с библиотекой, для которуй у IDA отсутствует информация о прототипах функций, вы можете сгенерировать сигнатуры типов для всех функций содержащихся в библиотеке для того, чтобы предоставить IDA возможность автоматически комментировать дизассемблированный код. Примерами подобных библиотек могут послужить библиотеки с изображениями или криптографические библиотеки, которые не являются стандартной частью Windows, но могут быть довольно распространены. В качестве конкретного примера можно привести криптографическую библиотеку OpenSSL.

Мы можем добавлять информацию о прототипах функций в .til-файл, также как мы добавляли информацию о сложных типах данных в тот же файл при помощи парсинга прототипов функций доступного через меню File ► Load File ► Parse C Header File. К сожалению, как мы ранее обсуждали, на данный момент это единственный способ добавлять содержимое в .til-файлы, и это содержимое будет ассоциировано только с базой данных, в которой проводился парсинг. Так как .til-файлы архивируются в .idb-файлы во время закрытия базы данных, то единственный путь извлечь

информацию, это скопировать .til-файл из рабочей директории IDA, когда база данных открыта. Процесс создания библиотеки типов представлен следующими пунктами:

Загрузите исполняемый файл в новую базу данных. Какой именно файл не имеет значения, так как нам просто надо получить доступ к возможности IDA производить парсинг файлов C. Для показательных целей мы будем использовать исполняемый файл C:\IdaBook\ch13_examples\example_13_1.exe.

Произведите парсинг заголовочных файлов C содержащих прототипы функций, который вы хотите добавить. Для этого может потребоваться модифицировать заголовочные файлы (например убрать все нестандартные типы, такие как uchar или dword), чтобы они корректно воспринимались IDA. Информация, извлеченная во время парсинга будет добавлена в локальный .til-файл. В данном случае имя файла будет C:\IdaBook\ch13_examples\example_13_1.til.

Перед закрытием базы данных скопируйте .til-файл в <IDADIR>/til. Желательно изменить имя файла в соответствии с названием библиотеки, например openssl.til. Теперь .til=файл доступен для использования через операцию вставки (горячая клавиша INSERT) в окне Loaded Type Libraries (View ► Open Subviews ► Type Libraries).

Теперь можно закрыть базу данных и по желанию сохранить (если заголовочные файлы не имели отношения к конкретной базе данных, то вы можете просто проигнорировать изменения).

Все это хорошо и даже отлично, но только в случае если у вас есть доступ к исходному коду, который можно использовать для парсинга. К сожалению, чаще всего у вас не будет доступа к исходному коду, но вам все равно будет необходимо высококачественное дизассемблирование. Как же можно обучить IDA, в случае если у вас нет исходного кода? Это и является точным назначением утилит IDS (idsutils). Утилиты IDS представляют собой набор трех программ использующихся для создания .ids-файлов. Давайте сначала обсудим,что представляет собой .ids-файл, и затем рассмотрим, как же их создавать.

3.1.1. Файлы IDS

IDA использует .ids-файлы для того чтобы «укрепить» свои знания о библиотечных функциях. В .ids-файле описывается содержимое разделяемой библиотеки, путем создания списка всех экспортированных функций содержащихся в библиотеке. Информация о каждой функции включает в себя имя функции, связанный с ней порядковый номер, использует ли функция stdcall, и если да, то, какое

количество байт функция удаляет из стека при завершении, также хранятся дополнительные комментарии, которые отображаются при ссылке на функцию в дизассемблированном коде. На практике .ids-файлы представляют собой сжатые .idt-файлы, содержащие текстовые описания каждой библиотечной функции.

При первой загрузке исполняемого файла в базу данных, IDA определяет, с какими разделяемыми библиотеками он связан. Для каждой разделяемой библиотеки IDA производит поиск соответствующего .ids-файла в папке <IDADIR>/ids для того, чтобы получить описания любых библиотечных функций, к которым данный исполняемый файл может обращаться. Важно понимать, что .ids-файлы не обязательно хранят сигнатуры функций. Поэтому IDA не всегда может предоставить анализ параметров функций, базируясь только на информации из .ids-файлов. Однако IDA может произвести точный подсчет указателя стека, в случае если .ids-файл содержит корректную информацию о вызове функций и количестве байт, которые функции удаляют из стека. В ситуациях, где из DLL экспортируются искаженные имена (mangled names), IDA может вывести сигнатуру параметров функции из искаженного имени, в данном случае информация становится доступной при загрузке .ids-файла. Мы описываем синтаксис .idt-файлов в «Creating IDS Files” в разделе 8.1.2. В этом отношении в .til-файлах содержится гораздо больше информации полезной для дизассемблирования вызовов функций, однако не стоит забывать, что для их генерации необходим исходный код.

Библиотечные функции, которые используют вызов, подобный stdcall могут нанести ущерб анализу указателя стека. В случае отсутствия библиотек типов или файлов информации .ids, IDA не может определить использует импортированная функция вызов stdcall или нет. Это важно, так как IDA может потерять возможность корректно отслеживать поведение указателя стека для функций, у которых неизвестна информация об их вызове. Кроме информации об использовании вызова stdcall, IDA также необходимо точно знать какое количество байт функция удаляет из стека после завершения. В случае отстствия информации о вызове, IDA пытается автоматически определить исплользуется ли вызов подобный stdcall при помощи метода математического анализа известного как симплексный метод (simplex method). Второй метод основывается на ручной правке пользователем. На рис.3. отображена специальная форма диалога редактирования функций.

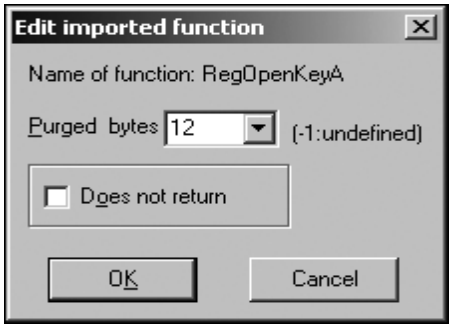


Рис.3.1. Редактирования импортированной функции

Вы можете открыть этот диалог, выбрав ячейку нужной функции в таблице импорта и затем вызвав меню Edit ► Functions ► Edit Function (горячая клавиша ALT-P). Отметим некоторую ограниченную функциональность данного диалога. Из-за того, что это импортированная функция у IDA нет доступа к скомпилированному телу функции, и поэтому нет информации относительно структуры кадра стека функции, и нет прямого доказательства использования вызова подобного stdcall. Из-за отсутствия такой информации IDA устанавливает поле Purged bytes в -1, показывая, что неизвестно какое количество байт функция удаляет из стека при завершении. Для того чтобы переопределить это значение, вам нужно самостоятельно его ввести, тогда IDA будет пользоваться этой информацией при анализе указателя стека. В случае если у IDA есть информация о поведении функции, это поле может быть уже заполнено. Заметим, что это поле никогда не заполняется в результате симплексного метода анализа.

3.1.2. Создание файлов IDS

Набор утилит idsutils используется для создания .ids-файлов. В него входит два парсера, dll2idt.exe для извлечения информации из Windows DLL и ar2idt.exe для извлечения информации из ar-библиотек. В обоих случаях результатом является текстовый .idt-файл, содержащий строки с экспортированными функциями, в которых осуществляется привязка порядкового номера к имени функции. Синтаксис .idt-файлов достаточно прост и описан в файле readme.txt поставляемом с idsutils. Большинство строк в .idt-файле используются для описания экспортированных функций по следующей схеме:

Запись начинается с положительного числа представляющего собой порядковый номер экспортированной функции.
После порядкового номера через пробел следует директива имени в форме Name=function, например, Name=RegOpenKeyA. Если использован специальный нулевой порядковый номер, то директива имени используется для указания имени библиотеки описаннываемой в текущем .idt-файле, например:

```
0 Name=advapi32.dll
```

Может быть использована дополнительная директива Pascal, используемая для того, чтобы указать что функция использует вызов подобный stdcall и отразить число байт, которое функция убирает из стека при завершении. Пример:

```
483 Name=RegOpenKeyA Pascal=12
```

Также может присутствовать дополнительная директива для указания комментария, который должен отображаться с функцией с каждой ссылкой на нее в дизассемблированном коде. Полностью запись может выглядеть вот так:

```
483 Name=RegOpenKeyAPascal=12 Comment=Openaregistrykey
```

Дополнительные директивы описаны в файле readme.txt. Назначение утилит парсинга idsutils заключается в максимальном автоматизировании создания .idt-файлов. Первый шаг в создании .idt-файла заключается в получении файла библиотеки, которую вы хотите разобрать, а далее осуществить парсинг, воспользовавшись соответствующей утилитой. Например, для того чтобы создать .idt-файл для OpenSSL библиотеки ssleay32.dll, то нужно воспользоваться следующей командой:

```
$ ./dll2idt.exe ssleay32.dll
Convert DLL to IDT file. Copyright 1997 by Yury Haron. Version 1.5
File: ssleay32.dll   ... ok
```

В случае если парсинг прошел успешно, то будет создан файл SSLEAY32.DLL. Отличие в регистре букв в имени файлов заключается в том, что выходной файл именуется в соответствии с названием библиотеки, хранимом в самом файле библиотеки. Первые несколько строк результирующего .idt-файла:

```
ALIGNMENT 4
;DECLARATION
;
0 Name=SSLEAY32.dll
;
121 Name=BIO_f_ssl
173 Name=BIO_new_buffer_ssl_connect
122 Name=BIO_new_ssl
174 Name=BIO_new_ssl_connect
124 Name=BIO_ssl_copy_session_id
```

Заметим, что парсеры не могут определить, использует ли функция вызов, подобный stdcall, и сколько байт убирается из стека по завершении. Дополнения для директив Pascal или Comment должны быть выполнены вручную при помощи текстового редактора до создания .ids-файла. Заключительным шагом для создания .ids-файла является использование утилиты zipids.exe для того, чтобы сжать .idt-файл, и затем скопировать результирующий .ids-файл в папку <IDADIR>/ids.

```
$ ./zipids.exe SSLEAY32.idt
File: SSLEAY32.idt   ... {219 entries [0/0/0]}         packed
$ cpSSLEAY32.ids ../Ida/ids
```

Здесь показано, что IDS загружает SSLEAY32.ids каждый раз, когда загружается бинарный файл, связанный с ssleay32.dll. В случае если вы не скопировали .ids-файл в папку, то вы можете просто загрузить его через меню File ► Load File ► IDS File.

Дополнительный шаг в использовании .ids-файлов заключается в том, что вы можете привязать их к конкретным .sig или .til файлам. Когда вы выбираете .ids-файл, IDA использует конфигурационный файл .ids <IDADIR>/idsnames. Этот текстовый файл хранит строки, позволяющие сделать следующее:

Привязать имя разделяемой библиотеки к соответствующему имени .ids-файла. Это позволяет IDA корректно определять .ids-файл, когда имя разделяемой библиотеки некорректно приводится к имени в формате MS-DOS 8.3. Пример:

```
libc.so.6     libc.ids        +
```

Привязать .ids-файл к .til-файле. В данных случаях IDA автоматически загружает указанный .til файл каждый раз при загрузке данного .ids-файла. Следующий пример показывает, как сделать так чтобы openssl.til загружался каждый раз при загрузке SSLEAY32.ids:

```
SSLEAY32.ids    SSLEAY32.ids    +   openssl.til
```

Привязать .sig-файл к соответствующему .ids-файлу. В подобном случае IDA автоматически загружает указанный .ids-файл каждый раз, когда данный .sig-файл применяется к дизассемблированному коду. Строка ниже дает указание IDA загружать SSLEAY32.ids каждый раз, когда пользователь применяет сигнатуру FLIRT libssl.sig:

```
libssl.sig      SSLEAY32.ids    +
```

В Главе 5 мы рассмотрим альтернативные способы использования парсеров библиотек, основанные на скриптах, а также воспользуемся анализом функций IDA для генерации более информативных .idt-файлов.

3.2. Изменение предопределенных комментариев при помощи loadint

Мы знакомы с автокомментариями IDA, которые отображаются рядом с инструкциями языка ассемблера. Вот два примера подобных комментариев:

```
.text:08048654    leaecx, [esp+arg_0] ;LoadEffectiveAddress
.text:08048658    and   esp, 0FFFFFFF0h ; Logical AND
```

Источником для этих комментариев является файл <IDADIR>/ida.int, в котором они хранятся отсортированными по типу процессора, и по типу инструкции. В случае если автокомментирование включено, IDA будет производить поиск комментариев для каждой инструкции, и будет отображать найденные справа от дизассемблированного кода.

Утилиты loadint предоставляют возможность, как модифицировать существующие комментарии, так и добавлять новые в файл ida.int. Также как и в случае с другими дополнительными утилитами, документацию по утилите loadint вы можете найти в файле

readme.txt, поставляемом с loadint. Также вы можете найти предопределенные комментарии для всех процессорных модулей IDA в виде .cmt-файлов. Изменение существующих комментариев заключается в простом поиске файла комментариев для нужного процессора (например, pc.cmt для x86), внесении желаемых изменений в текст, запуске loadint.exe для пересоздания файла комментариев ida.int, и копировании этого файла в основную папку IDA, откуда он будет загружен в следующий раз при запуске IDA. Простой запуск утилиты для пересоздания базы данных комментариев выглядит так:

```
$ ./loadint.exe comment.cmt ida.int
Comment base loader. Version 2.04. Copyright (c) 1991-2007 by Ilfak Guilfanov
Output database is not found. Creating...

15958 cases, 15498 strings, total length: 512811
```

Примерами, для чего вам может потребоваться редактировать базу комментариев, являются изменение существующих комментариев, или включение комментариев для инструкций, у которых назначенные комментарии отсутствуют. В файле pc.cmt, например, многие из стандартных инструкций закомментированы, чтобы при включении автокомментирования не отображалось слишком много комментрариев. Следующие строки, извлеченные из pc.cmt, показывают, что для инструкции x86 mov по умолчанию комментарии не отображаются:

```
NN_ltr:          "Load Task Register"
//NN_mov:        "Move Data"
NN_movsp:        "Move to/from Special Registers"
```

Для того, чтобы включить комментирование инструкции move, необходимо раскомментировать среднюю строку, и пересоздать базу данных комментариев как было описано ранее.

Существует одно маленькое замечание, спрятанное глубоко в документации loadint - у loadint.exe должен быть доступ к ida.hlp, который поставляется вместе с IDA. Если вы получите следующее сообщение, то вам стоит скопировать ida.hlp в папку loadint и затем заново запустить loadint.exe.

```
$ ./loadint.exe comment.cmt ida.int
Comment base loader. Version 2.04. Copyright (c) 1991-2007 by Ilfak Guilfanov
```

Can't initialize help system.
File name: 'ida.hlp', Reason: can't find file (take it from IDA distributive).

Также вы можете использовать опцию -n для того, чтобы указать расположение папки IDADIR, как показано в следующей командной строке:

$ ./loadint.exe -n <IDADIR> comment.cmt ida.int

Файл comment.cmt служит в качестве главного входного файла для loadint. Синтаксис этого файла описывается в документации loadint. Грубо говоря, comment.cmt создает привязку типов процессора соответствующих файлам комментариев. Файлы комментариев для определенных процессоров в свою очередь определяют привязку конкретных инструкций к соответствующим текстовым комментариям. Весь процесс проходит в зависимости от нескольких констант (C-подобные enums), которые определяют все типы процессоров (найденные в comment.cmt), и все возможные инструкции для каждого процесса (найденные в allins.hpp).

Если вы хотите добавить предопределенные комментарии для нового типа процессора, то придется сделать больше, чем просто изменить существующие комментарии, так как данный процесс тесно связан с процессом создания новых процессорных модулей. Если не погружаться слишком глубоко в процесс создания новых модулей, то для создания комментариев для нового типа процессора, требуется сначала создать набор численных констант (разделяемых с данным процессорным модулем) в файле allins.hpp, который определяет по константе для каждой инструкции в наборе. Во-вторых, вы должны будете создать файл комментариев, в котором осуществляется привязка перечисленных инструкций к соответствующим им комментариям. В-третьих, вам необходимо определить новую константу для типа процессора, и создать запись в файле comment.cmt, в котором привязывается тип процессора к соответствующему файлу комментариев. После того как вы выполните эти три шага, вам необходимо запустить loadint.exe для того, чтобы пересоздать базу данных комментариев.

ГЛАВА 4. ПАТЧИНГ БИНАРНЫХ ФАЙЛОВ И ДРУГИЕ ОГРАНИЧЕНИЯ IDA

Одним из самых часто задаваемых вопросов у новых пользователей IDA, является вопрос “Как я могу пропатчить бинарный файл при помощи IDA?”. Самым простой ответ на этот вопрос - “Никак”. Основная цель IDA, это помочь вам в понимании поведения бинарного файла, предоставив вам возможности для наилучшего дизассемблирования из всех существующих. IDA не предназначена для того чтобы облегчить модификацию исследуемых вами файлов. Самые упрямые не желают принимать “Никак” в качестве ответа, и продолжают задавать вопросы, такие как: “Что по поводу менюEdit ► Patch Program?” и “Какоетогдапредназначениеуменю File ► Produce File ► Create EXE File?”. В данной главе мы обсудим эти аномалии и посмотрим, можем ли мы заставить IDA помогать нам в разработке патчей для бинарных файлов хотя бы немного.

4.1. Малоизвестное ьеню Patch Program

Меню Edit ► Patch Program впервые было упомянуто в Главе 6 как одна из скрытых возможностей версии IDA с графическим интерфейсом, которую можно включить в файле конфигурации idagui.cfg (в консольных версиях IDA меню Patch доступно по умолчанию). На рисунке 4. показаны опции, доступные в подменю Edit ► Patch Program.

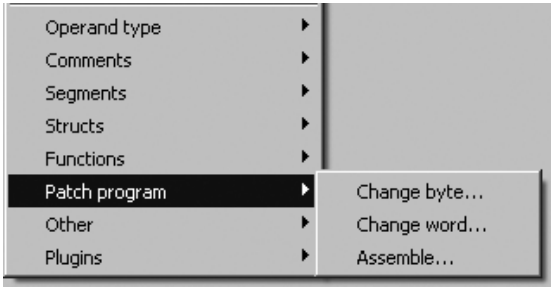


Рис.4. Подменю Patch Program

Кажется, что каждый пункт подменю так и дразнит возможностью модификации бинарного файла различными путями. Но

на самом деле, это опции предлагают три различных пути для модификации базы данных и бинарного файла, который из нее создается. После создания базы данных IDA больше не обращается к оригинальному бинарному файлу. Исходя из вышесказанного, данное меню стоило назвать Patch Database, для более точного отражения его возможностей.

Но не все потеряно, так как опции, находящиеся в меню, показанном на рис.4. предлагают самый простой путь понаблюдать за эффектами от изменений, которые впоследствии можно сделать с оригинальым бинарным файлом. Далее в этой главе мы расскажем как эспортировать сделанные вами изменения, и использовать их для того, чтобы пропатчить оригинальный бинарный файл.

4.1.1. Изменение отдельных байт в базе данных

Опция меню Edit ► Patch Program ► Change Byte используется для того, чтобы изменить значение одного или нескольких байт в базе данных IDA. На рис.4.1 показан диалог байт-патчинга.

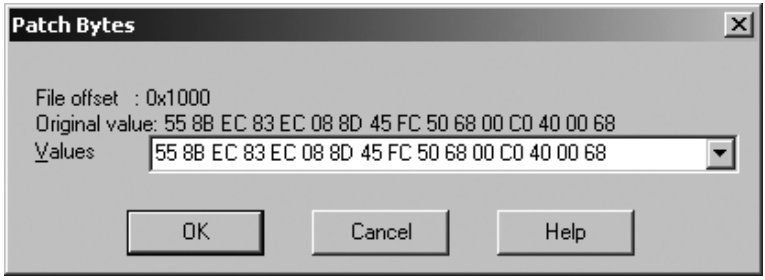


Рис.4.1. Диалог байт-патчинга

В диалоге отражаются 16 байт, начинающихся с текущей позиции курсора. Вы можете изменить несколько или все эти байты, но вы не можете изменять байты, следующие за шестнадцатым, для этого вам надо закрыть диалог, переместить курсор на нужную позицию и открыть его снова. В диалоге также отображается смещение для отображаемых байт (File offset). Это значение представлено в виде шестнадцатеричной величины, показывающей смещение в бинарном файле. Данное значение не отображает виртуальный адрес, по которому байты расположены в базе данных. То, что IDA сохраняет информацию

о смещении в оригинальном файле, пригодится в будущем при создании патча для оригинального бинарного файла. К тому же, несмотря на количество внесенных изменений, вы всегда сможете посмотреть оригинальное значение байта в поле Original Value. Стандартное средство для автоматической отмены изменений отсутствует, но можно создать скрипт для выполнения данной задачи.

4.1.2. Изменение слова в базе данных

Существует также еще и возможность для патчинга слов, пусть и не такая полезная как байт-патчинг. На рис.4.2. показан диалог, в котором можно изменить только одно слово из двух байт за раз.

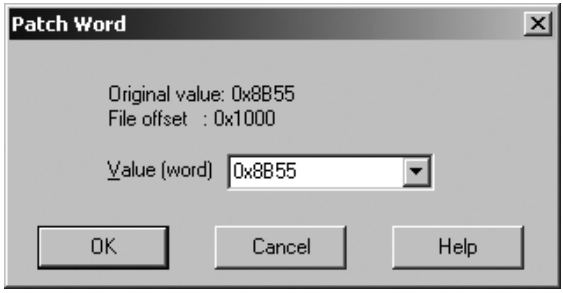


Рис.4.2. Диалог Patch Word

Также как и в диалоге байт-патчинга, в окне отображается смещение в оригинальном бинарном файле, а не виртуальный адрес изменяемого слова. Очень важно помнить, что байты слова отображаются в порядке принятом для работы процессора. Например, при дизассемблировании кода для x86, слова воспринимаются в остроконечном формате, в то время как в MIPS они воспринимаются в тупоконечном формате. Обязательно учитывайте это при введении нового значения для слова. Также как и в предыдущем диалоге отображается оригинальное значение слова из бинарного файла, независимо от того сколько раз вы изменяли данное слово.

4.1.3. Использование диалога Assemble

Наверное, одной из самых интересных возможностей IDA доступных через меню Patch Program является опция Assemble (Edit ►

Patch Program ► Assemble). К сожалению, данная опция доступна не для всех типов процессоров, так как она зависит от внутренней возможности дизассемблирования в данном процессорном модуле. Например, процессорный модуль x86 поддерживает дизассемблирование, в то время как в процессорном модуле MIPS нет такой возможности. В случае если возможность дизассемблирования недоступна, то вы увидите сообщение "Sorry, this processor module doesn't support the assembler.".

Опция Assemble позволяет вам вводить утверждения на языке ассемблера, которые обрабатываются внутренним ассемблером. Результирующие байты инструкции записывается в текущую позицию на экране. На рисунке 4.3. отображен диалог Assemble Instruction используемый для введения инструкции.

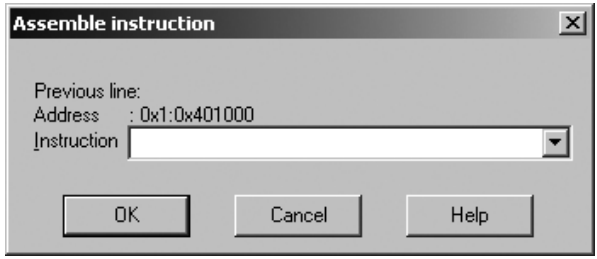


Рис.4.3. Диалог Assemble Instruction

Вы можете ввести за раз только одну инструкцию в поле Instruction. При вводе используется тот же синтаксис что и в дизассемблированном коде x86. После нажатия кнопки OK (или ENTER), введенная вами инструкция ассемблируется, и соответствующие ей байты заносятся в базу данных начиная с виртуального адреса, отображаемого в поле Address. Внутренний ассемблер IDA позволяет использовать символьные имена внутри инструкций, присутствующие внутри программы. Синтаксис подобный mov [ebp+var_4], eax и call sub_401896 является абсолютно верным, и ассемблер будет корректно обрабатывать символьные ссылки.

После ввода инструкции диалог остается открытым и готовым к вводу новой инструкции по виртуальному адресу, следующему за адресом введенной ранее инструкции. Во время ввода новых инструкций в поле Previous line отображается предыдущая введенная инструкция.

При вводе новых инструкций нужно обязательно обращать внимание на их расположение, особенно если вводимая инструкция длиннее замещаемой. В случае если вводимая инструкция короче замещаемой, вам необходимо решить, что делать с оставшимся местом (введение инструкции NOP является одним из возможных вариантов). Если же вводимая инструкция длиннее замещаемой, то IDA заменит необходимое количество байт в последующей инструкции для того, чтобы вместить вводимую. Это может быть не тем поведением, которое вы ожидали, поэтому важно планировать вводимые изменения с осторожностью перед использованием ассемблера для изменения байт программы. Простого пути для создания дополнительного места для вводимой инструкции без замещения следующей не существует.

Важно помнить, что возможности IDA для патчинга базы данных ограничиваются маленькими, простыми патчами, которые легко умещаются в пространство внутри базы данных. Если у вас есть патч, которому требуется дополнительное место, то вам придется найти такое неиспользуемое место внутри оригинального бинарного файла. Подобные пустые места часто добавляются компиляторами для отделения секций внутри бинарного файла. Например, в бинарных файлах Windows PE каждая отдельная часть программы должна начинаться с адреса кратного 512 байтам. В случае если части программы не требуется 512 байт, то оставшиеся байты заполняются для того чтобы сохранить границы между частями. Следующие строки из дизассемблированного кода файла PE отражают подобную ситуацию:

```
.text:0040963E          ; [00000006 BYTES: COLLAPSED FUNCTION RtlUnwind. PRESS KEYPAD "+" TO EXPAND]
.text:00409644                  align 200h
.text:00409644  _text           ends
.text:00409644
.idata:0040A000  ; Section 2. (virtual address 0000A000)
```

В данном случае IDA использует директиву align для отображения того, что часть программы дополнена до 512-байтной (200h) границы, начинающейся с адреса .text:00409644. Верхней границей являются следующие 512 байт или .text:00409800. Дополненная область обычно заполняется нулями и вполне заметна в шестнадцатеричном виде. В конкретном примере есть свободное место для вставки до 444 байт (0x1BC = 409800h - 409644h), измененных программных данных, которые могут переписать несколько или все нули в конце части .text . Вы можете пропатчить функцию для

того,чтобы она перемещалась к данной области в бинарном файле, выполняла добавленные инструкции, и затем возвращалась обратно к оригинальной функции.

Стоит отметить что секция .idata в бинарном файле на самом деле не начинается до адреса .idata:0040A000. Это результат ограничения расположения памяти (а не файла), в котором требуется, чтобы секции начинались в рамках 4Кб (одна страница памяти). В теории можно вставить дополнительные 2,048 байт модифицированных данных в диапазоне памяти 00409800-0040A000. Сложность заключается в том, что фактически байты из данной области отсутствуют в бинарном файле, хранимом на диске. Для того, чтобы использовать это пространство, придется выполнить нечто большее, чем простую вставку необходимых байтов. Для начала необходимо вставить блок данных размером 2048 байт между окончанием секции .text и началом секции idata . Далее необходимо исправить размер секции .text в заголовке PE файла. И наконец, необходимо изменить расположение .idata и всех послудующих секций внутри заголовка PE файла для того, чтобы указать, что все следующие секции расположены на 2048 байт дальше. Подобные изменения не звучат слишком ужсложными, но для их выполнения требуется внимание к деталям и хорошее знание формата PE файлов.

4.2. Выходные файлы IDA и создание патчей

Еще одной интересной опцией меню, является File ► Produce File. Судя по опциям в данном меню, IDA может создавать файлы форматов MAP, ASM, INC, LST, EXE, DIF, и HTML. Многие из этих форматов представляют большой интерес, поэтому мы опишем их все в следующих разделах.

4.2.1. Файлы MAP

В файлах формата .map описывается общее расположение внутри бинарного файла, включающее в себя информацию о секциях, из которых состоит файл, и расположение символов внутри каждой части. При создании .map-файла вам необходимо ввести его имя и типы символов, которые вы хотите в нем сохранить. На рис.4.4 показан диалог создания .map-файла, в котором можно выбрать какую информацию следует сохранить.

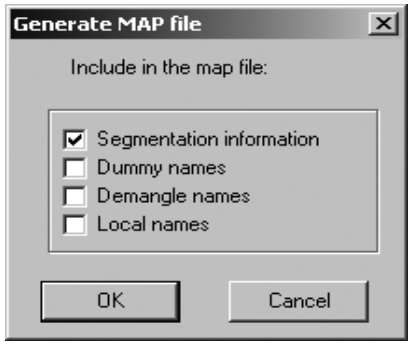


Рис.4.4. Создание map-файла

Адресная информация в .map-файле представлена с использование логических адресов. Логический адрес описывает расположение символа с использованием номера сегмента и смещения внутри сегмента. Первые несколько строк простого .map-файла отображены ниже. В данном коде присутствуют три сегмента и первые два символа из множества. Логический адрес _fprintf указывает на то, что она расположена со смещением 69h внутри первого (.text) сегмента.

```
Start         Length     Name                   Class
0001:00000000 000008644H .text                  CODE
0002:00000000 000001DD6H .rdata                 DATA
0003:00000000 000002B84H .data                  DATA

 Address         Publics by Value
 0001:00000000       _main
 0001:00000069       _fprintf
```

Файлыформата .map, генерируемые IDA совместимыс Borland Turbo Debugger. Назначением данных файлов, является помощь в восстановлении символьных имен из очищенных бинарных файлов.

4.2.2. Файлы ASM

IDA может создать .asm-фалйа из текущей базы данных. Основная идея заключается в создании файла, который можно пропустить через ассемблер для пересоздания использованного

бинарного файла. IDA пытается сохранить достаточное количество информации, включая такие вещи как расположение структур, для того чтобы сделать использование ассемблера максимально возможным. Удастся ли вам воспользоваться сгенерированным .asm-файлов или нет, зависит от множества факторов, причем не на последнем месте стоит фактор того, что используемый вами ассемблер должен понимать синтаксис, используемый в IDA.

Используемый синтаксис языка ассемблера определяется настройкой Target assembler, которую можно найти во вкладке Analysis в меню Options ► General. По умолчанию IDA создает файл ассемблера, в котором отражается вся база данных. Однако вы можете ограничить диапазон листинга, при помощи мыши или, используя SHIFT-стрелка вверх и SHIFT-стрелка вниз для прокрутки и выбора области которую вы хотите использовать. В консольной версии IDA можно использовать команду Anchor (ALT-L) для того, чтобы установить начало области, и затем использовать стрелки для того чтобы ее расширить.

4.2.3. Файлы INC

Файл формата INC содержит определения структур данных и пронумерованных типов данных. По сути, он представляет собой дамп содержимого окна Structures в форме, пригодной для использования ассемблером.

4.2.4. Файлы LST

LST-файл это не более чем простой текстовый файл с дампом содержимого окна дизассемблирования IDA. Также как и в случае с ASM-файлами, вы можете выделить определенную область.

4.2.5. Файлы EXE

Несмотря на то, что данная опция является самой многообещающей, на деле, к сожалению, именно в ней больше всего "костылей". В общем, данная опция не работает с большинством типов файлов, поэтому, скорее всего вы увидите сообщение "This type of output file is not supported.".

Хоть это и была бы идеальная возможность для патчера, фактически пересоздать исполняемый файл из базы данных IDA очень сложно. Информация, которая представлена вам в базе данных

IDA,составлена из содержимого секций, которые в свою очередь составляют вместе оригинальный файл. Однако есть множество случаев, когда IDA обрабатывает не все секции исходного файла, и некоторая информация теряется при загрузке файла в базу данных, делая пересоздание файла из базы данных невозможным. Простейшим примером такой потери является то, что IDA по умолчанию не загружает секцию ресурсов (.rsrc) из PE файлов, делая восстановление секции ресурсов из базы данных невозможным.

В других случаях IDA обрабатывает всю информацию из оригинального файла, но делает ее труднодоступной в ее оригинальной форме. Подобные примеры включают в себя таблицы символов, таблицы импорта и таблицы экспорта, поэтому потребуется немало усилий для того, чтобы полноценно восстановить их для пересоздания исполняемого файла.

Алти Мар Гудмундсон создал свой вариант добавления возможности создания EXE-файлов в IDA - pe_scripts. Это набор скриптов IDA для работы с PE файлами. Один из файлов называется pe_write.idc, и он предназначен для создания дампа рабочего образа PE из существующей базы данных. Если вы хотите пропатчить PE-файл, вам следует придерживаться подобной последовательности действий:

- Загрузите необходимый PE-файл в IDA. Убедитесь, что бы отключили опцию Make imports в диалоге загрузки.
- Запустите скрипт pe_sections.idc для того, чтобы поместить все секции из оригинального файла в новую базу данных.
- Примените необходимые изменения к базе данных.
- Запустите скрипт Execute the pe_write.idc для того, чтобы создать дамп из базы данных и поместить в новый PE-файл.

Скрипты IDA рассматриваются в Главе 10.

4.2.6. Файлы DIF

DIF-файл содержит в себе список всех измененных в базе данных IDA байт. Это самый полезный формат файла для патчинга оригинального бинарного файла на основе внесенных в базу данных изменений. Форматдовольнопрост, вотпример .dif-файла:

This difference file is created by The Interactive Disassembler

dif_example.exe

000002F8: 83 FF
000002F9: EC 75
000002FA: 04 EC
000002FB: FF 68

В файл включен однострочный заглавный комментарий, после него следует название оригинального бинарного файла, и затем список изменений. Каждая строка с изменениями указывает на смещение в оригинальном файле, где был изменен байт, далее указывается исходное значение байта, и, наконец, новое значение. В данном примере база данных для файла dif_example.exe была изменена в четырех местах соответствующих смещению 0x2F8–0x2FB в оригинальном файле. Дело за малым — написать программу, которая путем парсинга .dif-файла будет модифицировать оригинальный бинарный файл. Вы можете найти подобную утилиту в интернете.

4.2.7. Файлы HTML

IDA может использовать язык гипертекстовой разметки для создания листинга дизассемблированного кода с цветовой разметкой. HTML-файл,создаваемый IDA, представляет из себя LST-файл с добавленными тегами HTML, для создания цветовой разметки аналогичной окну дизассемблирования IDA. К сожалению, всгенерированных HTML-файлах не содержится никаких гипертекстовых ссылок, которые могли бы упростить навигацию по файлу. Например, было бы полезно превратить именные ссылки в гиперссылки, для того, чтобы кликнув по ним, сразу же перемещаться к соответствующему имени.

ГЛАВА 5. ОТЛАДЧИК IDA

IDA широко известна как дизассемблер, и является одним из наилучших инструментов для выполнения статического анализа бинарных файлов. Из-за высокой сложности современных методов антистатического анализа, довольно часто можно встретить сочетание использования, как статического анализа, так и динамического, для использования преимуществ обоих. В идеальном случае все инструменты должны быть собраны в одном месте. В версии 4.5 IDA появился встроенный отладчик Windows PE, еще более укрепивший

роль IDA как инструмента для всестороннего реверс-инжиниринга. В последующих версиях возможности отладчика были расширены — появилась удаленная отладка для Windows, Linux и Mac OS X.

В нескольких следующих главах мы раскроем базовые возможности отладчика IDA, расскажем, как его можно использовать для анализа обфусцированного кода, и как выполнять удаленную отладку для бинарных файлов Windows, Linux и Mac OS X. Несмотря на то, что читатель знаком с использованием отладчиков, мы все равно рассмотрим множество базовых возможностей отладчиков во время разбора функций отладчика IDA.

5.1 Launching the debugger

Отладчики в основном используются для выполнения двух типов заданий: исследования образов памяти (дампов ядра) связанных с некорректно завершившимися процессами и для контролируемого запуска процессов. Сессия отладки начинается с выбора процесса для отладки. Есть два способа сделать это. Первый заключается в том, что большинство отладчиков умеют присоединяться к запущенному процессу (при наличии у пользователя прав на это). В зависимости от используемого отладчика, он может иметь возможность самостоятельно отображать список доступных процессов для выбора. В случае отсутствия такой возможности, пользователь должен самостоятельно определить ID процесса, и затем указать отладчику присоединиться к нему. Как именно происходит присоединение отладчика к процессу, зависит от используемой операционной системы, и не входит в рамки обсуждения в данной книге. При присоединении к уже запущенному процессу, наблюдать или контролировать загрузочную последовательность процесса невозможно, так как весь код загрузки и инициализации будет уже завершен к моменту присоединения.

Как именно происходит присоединение отладчика IDA к процессу, зависит от того, открыта ли в данный момент база данных. В случае если не база данных не открыта будет доступно меню Debugger ► Attach показанное на рис.5.

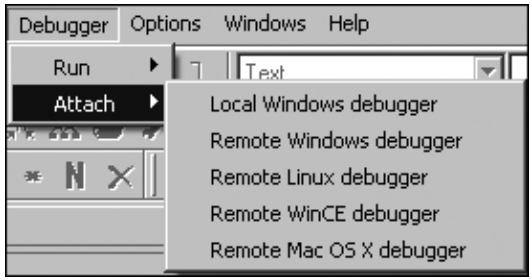


Рис.5. Присоединение к процессу

Из доступных опций меню можно выбрать различные отладчики. При выборе Local Windows Debugger, IDA отобразит список запущенных процессов, к которым можно присоединиться. На рис.5.1. показан пример подобного списка.

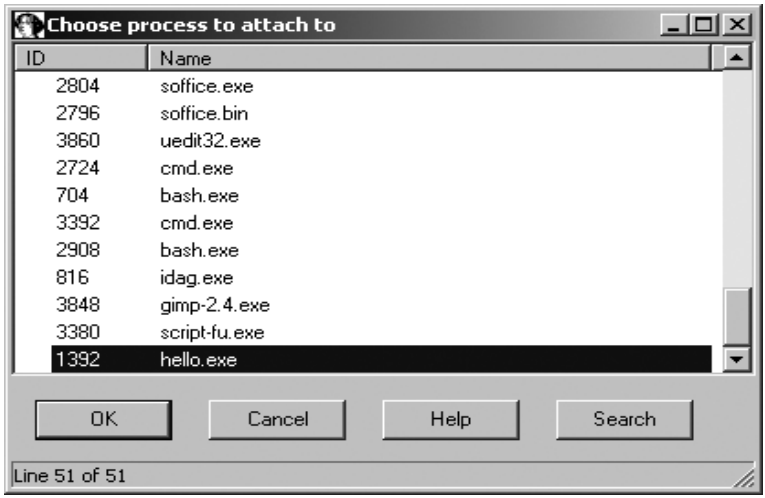


Рис.5.1. Диалог выбора процесса для отладки

После выбора процесса, отладчик создает временную базу данных, путем снятия образа памяти запущенного процесса. В дополнение к образу запущенного процесса, во временной базе данных

содержатся секции для всех разделяемых библиотек, загруженных процессом, что приводит к несколько более запутанной базе данных, чем вы привыкли видеть. В случае подобного присоединения к процессу у IDA будет мало доступной информации для дизассемблирования процесса, потому что загрузчик IDA никогда не обрабатывает соответствующий образ исполняемого файла, к тому же не был выполнен автоматизированный анализ бинарного файла. Фактически, после присоединения отладчика к процессу, будет дизассемблирвана только одна инструкция — та, на которую установлен указатель инструкций. Присоединение к процессу незамедлительно устанавливает его выполнение на паузу, позволяя вам установить точки остановки до повторного запуска процесса.

Альтернативным вариантом присоединения к процессу, является открытие соответствующего бинарного файла в IDA до присоединения к запущенному процессу. Если в данный момент присутствует открытая база данных, то меню Debugger принимает совершенно другой вид, показанный на рис.5.2.

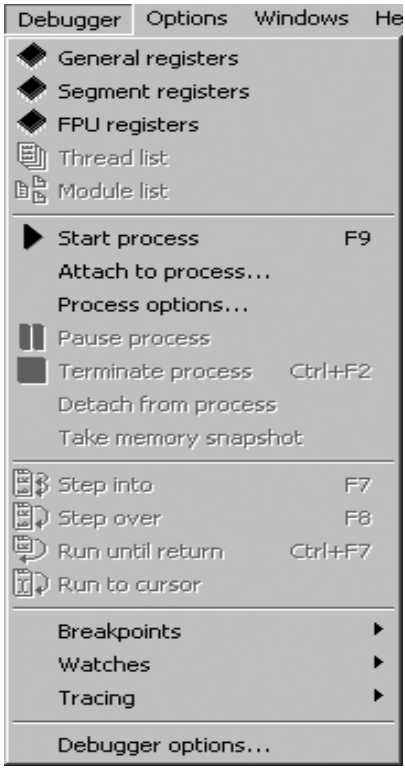


Рис.5.2. Меню Debugger при открытой базе данных

Для того чтобы можно было воспользоваться опцией меню Debugger ► Attach to Process, необходимо дать IDA возможность найти процесс с тем же именем, что и у загруженного на данный момент исполняемого файла. При выборе Attach to Process отображается список всех процессов с совпадающими именами. Вы можете выбрать присоединение к любому из этих процессов, но IDA не гарантирует, что этот процесс был запущен с тем же образом бинарного файла, что открыт в текущей базе данных IDA.

Вторым вариантом для отладки, является запуск процесса под контролем отладчика. В случае отсутствия открытой базы данных, новый процесс может быть запущен через меню Debugger ► Run. Если

же база данных открыта, то новый процесс запускается через Debugger ► Start Process или Debugger ► Run to Cursor. При использовании отладки с открытой базой данных, запущенный процесс будет выполняться до первой точки остановки (вам нужно установить ее до запуска Debugger ► Start Process), или пока вы самостоятельно не установите процесс на паузу при помощи меню Debugger ► Pause Process. При использовании пункта меню Debugger ► Run to Cursor точка остановки будет установлена автоматически в текущей позиции курсора. В таком случае, запущенный процесс будет выполняться до этой точки, или до точки установленной ранее. В случае если процесс не доходит до какой-либо точки остановки, он будет продолжать выполняться, пока не будет вручную установлен на паузу или завершен (Debugger ► Terminate Process).

Запуск процесса под контролем отладчика (в сравнении с присоединением к уже запущенному процессу) является единственным способом следить за всеми действиями производимыми процессом. Если установить точки остановки до запуска процесса, то становится возможным следить за стартовой последовательностью процесса. Контроль запуска процесса может быть полезен в случае работы с обфусцированным кодом, так как вы сможете поставить процесс на паузу сразу же после завершения функций деобфускации и до начала нормальной работы процесса.

Другим преимуществом данного способа является то, что IDA выполняет начальный автоматический анализ образа процесса до запуска самого процесса. Это повышает качество дизассемблированного кода по сравнение с присоединением к существующему процессу.

Есть несколько важных замечаний по поводу отладчика IDA, которые стоит упомянуть. Во-первых, в случае использования локальной отладки, вы сможете отлаживать только бинарные файлы, которые запускаются на вашей платформе (в данной главе подразумеваются только бинарные файлы Windows). Не существует никаких дополнительных слоев эмуляции, позволяющих выполнять отладку для других платформ или других типов процессоров при помощи локального отладчика IDA. Удаленная отладка в IDA позволяет вам работать с бинарными файлами платформ, для которых Hex-Rays выпускает сервер отладки (на данный момент это Windows, Windows CE/ARM, Mac OS X x86, и Linux).

Также важно отметить что, как и в случае с любым другим отладчиком, вам требуется оригинальный бинарный файл, и он должен быть запущен с полными привилегиями пользователя запустившего IDA. Это очень важно понимать, в случае если вы планируете использовать

отладчик IDA для анализа вредоносного программного обеспечения. Вы можете легко заразить компьютер, на котором производится отладка, в случае если вы не можете жестко контролировать данный экземпляр вредоносного ПО. Каждый раз, когда вы выбираете пункт меню Debugger ►Start Process, в IDA отображается следующее сообщение об опасности:

You are going to launch the debugger.
Debugging a program means that its code will be executed on your system.
Be careful with malicious programs, viruses and trojans!
REMARK: if you select 'No', the debugger will be automatically disabled.
Are you sure you want to continue?

Если выбрать ответ 'No', то меню Debugger будет убрано из основного окна IDA. Это меню не появится снова, пока вы не закроете текущую базу данных.

Настоятельно рекомендуется производить отладку вредоносного ПО только внутри песочницы.

5.2. Основные экраны отладчика

Независимо от того, каким образом вы запустили отладчик, после того как процесс будет поставлен на паузу, под его контролем IDA перейдет в режим отладки, где вам будет предоставлено несколько экранов. В отличие от стандартного режима дизассемблирования, в режиме отладки отсутствует привычный для Windows интерфейс с внутренними окнами. Вместо этого различные экраны отладчика представлены в виде отдельных окон с одним главным окном, которое используется для управления всеми остальными. Пять стандартных окон отладчика представлены на рис.5.3.

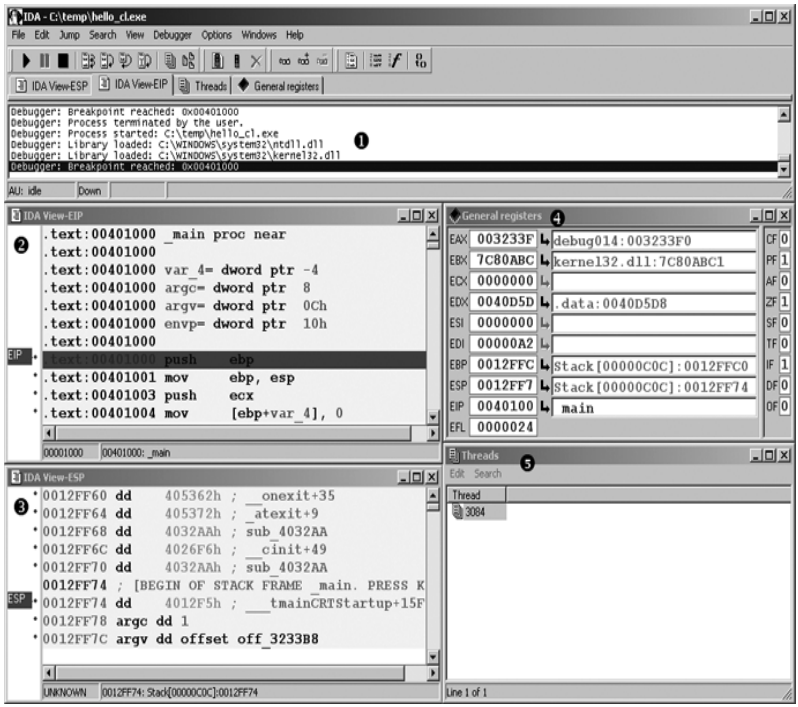


Рис.5.3. Окна отладчика IDA

Как показано на рисунке, главное управляющее окно обычно ❶ отображается в верхней части экрана, растянутое на всю его ширину. Также по умолчанию на панели инструментов отображается гораздо меньше кнопок, чем в режиме дизассемблирования. Несмотря на это, все функции дизассемблирования остаются доступными, и можно вернуть любые панели инструментов дизассемблирования, кликнув правой кнопкой мыши по области панели инструментов, и выбрав для отображения необходимые инструменты. В главном управляющем окне расположено стандартное окно сообщений IDA. Если свернуть главное окно, то остальные окна отладчика тоже будут свернуты.

Окно IDA View-EIPdisassembly представляет собой стандартное дизассемблирования, которое синхронизировано с текущим значением регистра указателя инструкций. В случае если IDA удается определить, что регистр указывает на область памяти внутри

окна дизассемблирования, то имя этого регистра отображается в левой колонке, иначе выводится адрес, на который регистр указывает. На рисунке 5.3. область, на которую указывает EIP, отмечена в View-EIP. По умолчанию IDA подсвечивает точки остановки красным, а следующую для исполнения инструкцию — синим. Дизассемблерный код связанный с отладкой создается при помощи того же процесса, что и в стандартном режиме дизассемблирования. Таким образом, скорее всего именно в отладчике IDA имеются наилучшие возможности для дизассемблирования среди всех отладчиков.

Окно View-ESP представляет собой еще одно окно для дизассемблирования, которое в основном используется для отображения содержимого стека выполняемого процесса. Все регистры, которые указывают на области стека (такие как ESP и EBP) подсвечиваются в левой колонке экрана. Используя информацию, полученную путем анализа текущей функции, IDA размещает комментарии на экране стека, которые отмечают рамки стекового кадра функции. Далее приведен пример разметки стека в IDA:

```
0012FF6C ; [BEGINOFSTACKFRAME _main. PRESS KEYPAD "-" TO
COLLAPSE]
0012FF6C var_4 dd 0
0012FF70 saved_fp dd    12FFC0h ; Stack[00000C0C]:saved_fp
0012FF74 retaddr dd    4012F5h ; ___tmainCRTStartup+15F
0012FF78 argc dd 1
0012FF7C argv dd offset off_3233B8
0012FF80 envp dd offset off_3233F0
0012FF80 ; [END OF STACK FRAME _main. PRESS KEYPAD "-" TO
COLLAPSE]
```

Первая и последняя строки в листинге отображают границы стекового кадра. Как отмечено в комментариях — стековые кадры можно сворачивать. Внутри стекового кадра, IDA делает пометки, используя имена, взятые из соответствующего экрана стекового кадра функций, позволяя легко определять область, в которой расположены некоторые переменные и соответствующие им значения внутри стека. IDA пытается предоставить как можно больше информации для каждого значения в стеке при помощи комментариев. В случае если в стеке находится адрес памяти, IDA пытается найти область функции (это помогает указать область, из которой эта функция была вызвана). Если же в стеке находится указатель на строку, то IDA отображает содержимое этой строки в виде комментария.

Окно General Registers отображает текущее содержимое регистров общего назначения процессора. Дополнительные окна для отображения сегментных регистров и регистров для данных с плавающей точкой могут быть открыты из меню Debugger.

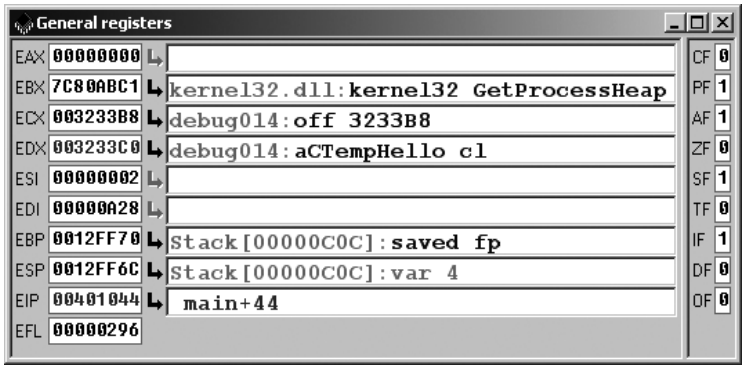


Рис.5.4. Окно General Registers

Внутри окна General Registers отображается: содержимое регистров в самом левом столбце, описание содержимого регистров в среднем столбце, и биты флагов процессора в правом столбце. Кликнув правой кнопкой мыши по значению регистра или биту флага, вы откроете меню Modify item, которое позволяет вам изменять содержимое любого регистра процессора или бита флага. Также клик правой кнопкой мыши по содержимому регистра предоставляет доступ к пункту меню Open Register Window. Если выбррать этот пункт, то откроется новое окно дизассемблирования с центром в области памяти, содержащейся в выбранном регистре. Если вы нечаянно закроете окно View-EIP или View-ESP, то вы можете воспользоваться командой Open Register Window, выбрав соответствующий регистр, и заново открыть окно. Если регистр указывает на действительную область памяти, то справа от регистра будет подсвечена черным загнутая вправо стрелка. Кликнув по этой стрелке можно переместиться к соответствующей области памяти в окне дизассемблера.

Последним экраном отладчика является окно Threads. В данном окне отображается список всех потоков текущего процесса. Дважды кликнув по любому потоку из списка в окне дизассемблирования View-EIP, будет совершен переход к текущей инструкции выбранного потока.

Дополнительные экраны отладчика доступны при помощи различных меню. Окно Modules (Debugger ► Module List) отображает список всех модулей (исполняемых файлов и разделяемых библиотек), загруженных в область памяти процесса. Двойной клик по любому модулю открывает список символов, экспортируемых этим модулем. На рис.5.5. показывается пример содержимого kernel32.dll.

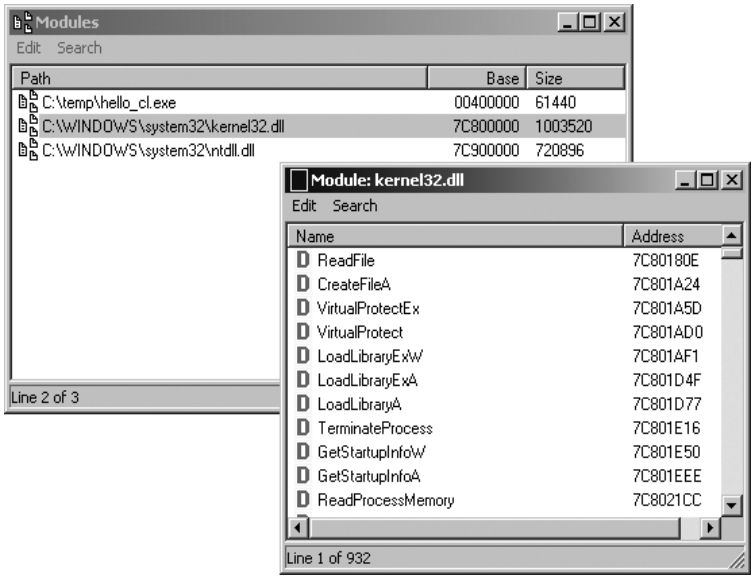


Рис.5.5. Окно Modules и связанное с ним содержимое модуля

Дополнительные экраны,относящиеся к работе отладчика, будут рассмотрены в "Управлении процессами" в части 5.3. В дополнение к экранам отладчика доступны все стандартные подвиды IDA, такие как Hex Dump и Segments (их можно вызвать при помощи меню Views ► Open Subviews).

5.3. Управление процессом

Наверное, одной из самых важных возможностей любого отладчика является способность контролировать, и при желании изменять поведение отлаживаемого процесса. Для этого в большинстве

отладчиков существуют команда, позволяющие выполнить одну или несколько инструкций до возвращения контроля отладчику. Подобные команды часто используются в сочетании с точками остановки, которые позволяют пользователю указать, что выполнение должно быть прервано, когда достигнута определенная инструкция или некоторое состояние.

Запуск процесса под контролем отладчика в основном достигается при использовании команд Step, Continue и Run. Так как именно они используются чаще всего, то будет полезным узнать какие кнопки панели инструментов и горячие клавиши связаны с этими командами. На рис.5.6. показаны кнопки панели инструментов, связанные с процессом запуска.

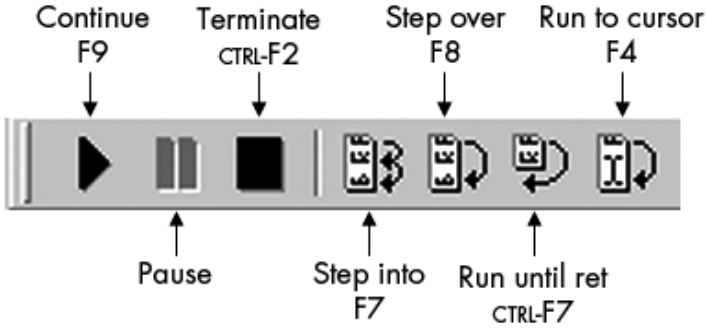


Рис.5.6. Инструменты отладчика, контролирующие выполнение процесса

Рассмотрим, что же делает каждая команда:

Continue - запускает заново выполнение процесса, поставленного на паузу. Выполнение продолжается до встречи точки остановки, установки паузы пользователем, или самостоятельного завершения процесса.

Pause устанавливает выполнение процесса на паузу.

Terminate - завершает выполняемый процесс.

Step Into - выполняет только следующую инструкцию. Если следующая инструкция является вызовом функции, то выполнение останавливается на первой инструкции этой функции. Название обусловлено тем, что отладчик входит в функцию.

Step Over - выполняет только следующую инструкцию. Если следующая инструкция является вызовом функции, то она воспринимается как одиночная инструкция, и остановка происходит после возврата из функции. Название обусловлено тем, что через функцию «перешагивают». Выполнение может быть остановлено до завершения функции, в случае если встречена точка остановки. Если вас не интересует поведение функци, то это очень полезный инструмент.

Run Until Return - вернуться к выполнению текущей функции и не останавливаться до выхода из нее. Полезно в случае если вы уже получили достаточно информации о функции, или если вы нечаянно нажали Step Into вместо Step Over.

Run to Cursor - возвращение к выполнению функции, остановка при достижении текущей позиции курсора (или встречи точки остановки). Удобно в случае выполнения больших блоков кода и отсутствии необходимости устанавливать точки остановки в каждом месте, где вы хотите остановить выполнение. Учтите, что пауза может не всегда сработать, так как текущая позиция курсора не может быть достигнута.

В дополнение к панели инструментов и горячим клавишам, все команды управления выполнением доступны через меню Debugger. Независимо от того, каким образом процесс был установлен на паузу, содержимое экранов отладчика обновляется для отображения текущего состояния процесса (регистры, флаги, память) на момент паузы.

5.3.1. Точки остановки

Точки остановки являются одной из основных возможностей отладки, которая удачно сочетается с возможностью установки выполнения процесса на паузу. Точки остановки устанавливаются в определенных областях программы для того, чтобы прервать ее выполнение. Также они являются прекрасным расширением идеи функции ”Run to Cursor”, в том плане, что процесс будет прерываться каждый раз, когда будет натыкаться на точку остановки, независимо от того, установлен в этом месте курсор или нет. К тому же, в то время как у вас только один курсор, точек остановки может быть несколько. Для того, чтобы установить точку остановки вам необходимо перейти к области, в которой вы ее хотите установить и воспользоваться горячей клавишей F2 (или кликнуть правой кнопкой мыши и выбрать Add Breakpoint). В тех местах, где установлена точка остановки, вся строка подсвечивается красным цветом. Для того, чтобы убрать точку остановки необходимо снова нажать F2 на той же строке. Полный

список установленных точек остановки вы можете посмотреть при помощи меню Debugger ► Breakpoints ► Breakpoint List.

По умолчанию в IDA используются программные точки остановки, которые выполняются путем замены байта опкода на инструкции остановки. Для бинарных файлов x86 такой инструкцией является int 3, ей соответствует значение опкода 0xCC. При нормальных обстоятельствах операционная система при встрече точки остановки передает управление отладчику, который следит за процессом. Как обсуждалось, обфусцированный код может изменить поведения программных точек остановки для того, чтобы попытаться создать преграды для работы любого присоединенного отладчика.

В некоторых процессорах (таких как x86, начиная с версии 386) присутствует поддержка аппаратных точек остановки. Такие точки обычно настраиваются при помощи специальных регистров процессора. В процессорах x86 данные регистры называются DR0–7 (debug registers от 0 до 7). Максимально можно установить до четырех точек остановки используя регистры DR0–3. Остальные регистры отладки используются для указания дополнительных ограничений для каждой точки остановки. При использовании аппаратных точек остановки нет необходимости добавлять в отлаживаемую программу специальную инструкцию. Вместо этого процессор сам решает, когда прервать выполнение программы в зависимости от значений, содержащихся в регистрах отладки.

После установки точки остановки можно изменить различные аспекты ее работы. Кроме того что можно просто прервать процесс, в отладчиках также часто поддерживаются точки остановки с условиями, позволяющие пользователю назначить некоторый набор условий, которые должны быть достигнуты для того, чтобы точка остановки сработала. При встрече такой точки, отладчик проверяет эти условия и если они не выполнены, то просто продолжает выполнение программы. Подобные точки с условиями нужны в случаях, когда вам необходимо чтобы остановка была совершена не сразу, а через некоторое время, когда будут достигнуты интересующие вас условия.

Отладчик IDA поддерживает как аппаратные точки остановки, так и точки с условиями. Для того, чтобы изменить поведение точки остановки (безусловной, программной) необходимо делать это после ее установки. Открыть диалог редактирования точки остановки можно при помощи клика правой кнопкой мыши по ней, и выбора в меню пункта Edit Breakpoint. На рис.5.7. показан диалог настройки точки остановки.

В поле Address отображается адрес редактируемой точки остановки, в то время как чекбокс Enabled показывает, активна эта точка или нет. Неактивные точки остановки пропускаются независимо

от достижения условий назначенных для остановки в этой точке. При помощи чекбокса Hardware можно установить, чтобы точка выполнялась в аппаратном режиме вместо программного.

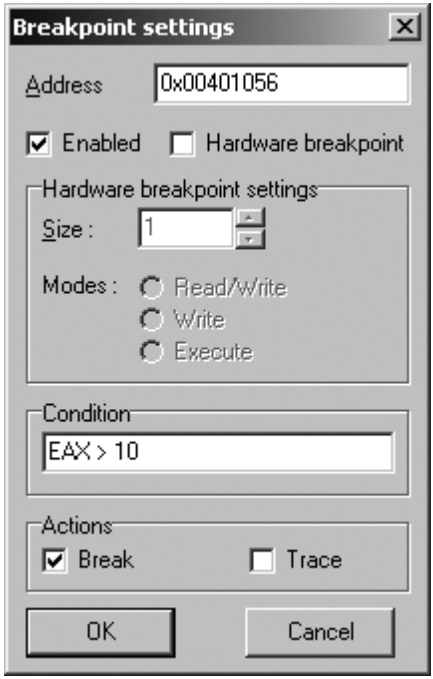


Рис.5.7. Диалог Breakpoint Settings

Предупреждаем, несмотря на то, что в текущей версии IDA можно установить более четырех аппаратных точек остановки (IDA 5.2.911), только четыре из них будут работать, остальные будут игнорироваться.

При указании аппаратной точки вам необходимо воспользоваться переключателем Modes для того, чтобы установить когда данная точка должна срабатывать — при выполнении, при записи, при записи\чтении. Последние два типа позволяют вам создавать точки остановки, которые срабатывают когда производится доступ к определенной области памяти, независимо от того, какая в

данный момент выполняется инструкция. Это может быть очень полезным, если вы больше заинтересованы в том, чтобы узнать, когда производится доступ к данным, чем, откуда он производится.

В дополнение к указанию режима аппаратной точки, необходимо указать размер. Для точек остановки, при выполнении, размер должен быть 1 байт. При записи или записи\чтении размер должен быть 1, 2 или 4 байта. Если размер равен 2, то адрес точки остановки должен быть словом. При размере в 4 байта адрес должен быть двойным словом. Размер аппаратной точки остановки сочетается с адресом и формирует диапазон байт, при котором точка остановки может сработать. Давайте попробуем это объяснить на примере. Допустим, что у нас есть аппаратная точка остановки, размер 4 байта, срабатывает при записи, адрес — 0804C834h. Данная точка сработает при записи 1 байта по адресу 0804C837h, 2 байт по адресу 0804C836h, и 4 байт по адресу 0804C832h. Во всех этих случаях в диапазоне 0804C834h-0804C837h записывается как минимум один байт. Больше информации о работе аппаратных точек остановки на процессорах x86 вы можете найти в Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide.

Точки остановки с условиями могут быть установлены путем введения выражения в поле Condition. Подобные точки, это возможность отладчика, а не набора инструкции или процессора. При срабатывании точки остановки, отладчик должен самостоятельно определить выполняются ли указанные условия, и указать должно ли выполнение программы быть установлено на паузу (при удовлетворении условий) или просто продолжено (условия не удовлетворены). Условия могут быть указаны как для программных, так и для аппаратных точек остановки.

Условия точек остановки IDA указываются при помощи выражений IDC. Выражения, которые определяются, как ненулевые считаются истинными и удовлетворяющими условия. Те выражения, что являются нулевыми, считаются ложными, не удовлетворяющими условия точки остановки, и точка остановки пропускается. Для того, чтобы помочь вам в написании выражений IDA предоставляет специальные переменные, через которые можно получить прямой доступ к содержимому регистров. Переменные именуются также как и соответствующие им регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, AX, BX, CX, DX, SI, DI, BP, SP, AL, AH, BL, BH, CL, CH, DL и DH.

К сожалению, не существует переменных, через которые можно получить доступ к регистрам флагов, ни к каждому по отдельности, ни ко всем вместе. Для того чтобы узнать флаги процессора, вам

необходимо вызывать функцию GeTRegValue . Если вам небоходимо вспомнить имена регистров или флагов, то вы можете подсмотреть их в окне General Registers. Несколько примеров выражений:

```
EAX == 100            // break if eax holds the value 100
ESI > EDI             // break if esi is greater than edi
Dword(EBP - 20) == 10  // Read current stack frame (var_20) and compare to 10
GetFlagsReg("ZF")      // break if zero flag is set
EAX = 1               // Set EAX to 1, this also evaluates to true (non-zero)
EIP = 0x0804186C       // Change EIP, perhaps to bypass code
```

Есть две вещи, которые стоит упомянуть относительно выражений для точек остановки — то, что функции IDC могут быть вызваны для получения информации о процессе и то, что назначение может использоваться как средство изменения значений регистров в определенных областях во время выполнения процесса. Ильфак собственноручно продемонстрировал этот метод в качестве примера переопределения возвращаемого функцией значения.

Оставшиеся опции, которые можно сконфигурировать в диалоге Breakpoint Settings, сгруппированы в области Actions в нижней части диалога. Чекбокс Break указывает, должно ли выполнение процесса устанавливаться на паузу при достижении точки остановки. Это может показаться странным — зачем создавать точку остановки, которая не останавливает процесс? На самом деле это полезно, в случае если вы просто хотите изменить значения какого-либо регистра при достижении необходимой инструкции, без необходимости ставить процесс на паузу. При помощи чекбокса Trace можно включить лог, в котором будет записываться каждый раз, когда точка остановки была достигнута.

5.3.2. Трассировка

Под трассировкой понимается запись в лог определенных событий, произошедших во время выполнения процесса. При трассировке события записываются в трассировочный буфер ограниченного размера, а также опционально в файл трассировки. Есть два вида трассировки: трассировка инструкций и трассировка функций. При трассировке инструкций (Debugger ► Tracing ► Instruction Tracing) IDA записывает адрес, инструкцию и значения всех регистров (кроме EIP), которые были изменены этой инструкцией. Данный вид трассировки может заметно замедлить выполнение процесса, из-за того

что отладчик пошагово выполняет процесс с необходимостью следить за состоянием всех регистров. Трассировка функций (Debugger ► Tracing ► Function tracing) является подвидом трассировки инструкций, в котором записываются только вызовы функций (опционально — возвращения из функций). При данной трассировке не записываются никакие значения регистров.

Доступны три вида отдельных событий для трассировки: запись, чтение\запись, выполнение. Как понятно из названий — можно вести лог при выполнении какого-либо из этих действий по заданному адресу. Каждый из этих видов трассировки выполняется с использованием особых точек остановки. Они не останавливают выполнение, и содержат установленную опцию trace . Трассировка при записи и при чтении\записи выполняется с использованием аппаратных точек остановки, и тем самым попадает под некоторые ограничения, упомянутые ранее, одним из самых важных которым является то, что нельзя использоваться одновременно больше четырех точек остановки. По умолчанию трассировка при выполнении реализуется при помощи программных точек остановки, и благодаря этому, ограничений на их количество нет.

Рис.5.8. отображает диалог опций трассировки (Debugger ► Tracing ► Tracing Options), который используется для конфигурации трассировочных операций в отладчике.

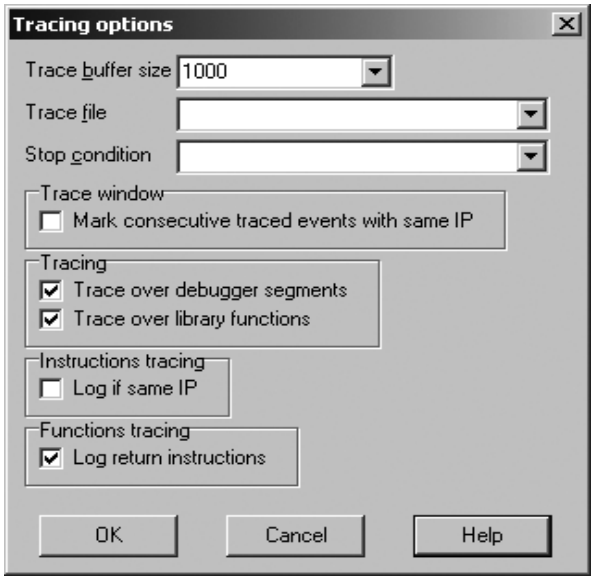


Рис.5.8. Диалог Tracing Options

Указанные здесь опции используются только для трассировки инструкций и функций. Эти опции не имеют никакого эффекта на события индивидуальных трассировок. Опция Trace buffer size определяет максимальное число событий трассировки для отображения. Если у буфера размер n, то будут отображаться только n последних событий. В случае указания лог-файла, все события будут записываться в него. Выбрать лог-файл при помощи файлового диалога нельзя, поэтому вам необходимо указать полный путь самостоятельно. В качестве условия для остановки может быть введено выражение IDC. Условие проверяется до трассировки каждой инструкции. Если условие истинно, то выполнение мгновенно останавливается. Такие выражения могут использоваться в качестве точек остановки с условиями, которые не привязаны к какой-то определенной области.

Если включить опцию Mark consecutive traced events with same IP, то все последовательные события, вызванные одной и той же инструкцией (IP обозначает указатель инструкции) будут отмечаться знаком равенства. Последовательные события, вызванные одной инструкцией, чаще всего связаны с инструкциями, которые используют

префикс REP в x86. Для того, чтобы отображался каждый повтор по одинаковому адресу, должна быть включена опция Log if same IP. Если не выбрать эту опцию, то инструкция с префиксом REP будет записываться только один раз, когда будет обнаружена. В нижеследующем листинге показана частичная трассировка со стандартными настройками:

```
00000F5D .text:080483DFrepmovsbECX=0 ESI=804963CEDI=8049648
00000F5D .text:080483E1 push    0    ESP=BFA23128
```

Отметим, что инструкция movsb перечислена только один раз.

В следующем листинге включена опция Log if same IP, поэтому rep записывается при каждой итерации:

```
00000F31 .text:080483DF rep movsb          ECX=B ESI=8049631
EDI=804963D EFL=210246 RF=1
00000F31 .text:080483DF rep movsb   ECX=A ESI=8049632 EDI=804963E
00000F31 .text:080483DF rep movsb   ECX=9 ESI=8049633 EDI=804963F
00000F31 .text:080483DF rep movsb   ECX=8 ESI=8049634 EDI=8049640
00000F31 .text:080483DF rep movsb   ECX=7 ESI=8049635 EDI=8049641
00000F31 .text:080483DF rep movsb   ECX=6 ESI=8049636 EDI=8049642
00000F31 .text:080483DF rep movsb   ECX=5 ESI=8049637 EDI=8049643
00000F31 .text:080483DF rep movsb   ECX=4 ESI=8049638 EDI=8049644
00000F31 .text:080483DF rep movsb   ECX=3 ESI=8049639 EDI=8049645
00000F31 .text:080483DF rep movsb   ECX=2 ESI=804963A EDI=8049646
00000F31 .text:080483DF rep movsb   ECX=1 ESI=804963B EDI=8049647
00000F31 .text:080483DF rep movsb   ECX=0 ESI=804963C EDI=8049648
EFL=200246 RF=0
00000F31 .text:080483E1 push    0   ESP=BFB16A18
```

В следующем случае была включена опция Mark consecutive traced events with same IP, поэтому отображаются специальные пометки, показывающие, что указатель инструкций не изменялся:

```
00000F34 .text:080483DF rep movsb          ECX=B ESI=8049631
EDI=804963D EFL=210246 RF=1
=       =       =       ECX=A ESI=8049632 EDI=804963E
=       =       =       ECX=9 ESI=8049633 EDI=804963F
=       =       =       ECX=8 ESI=8049634 EDI=8049640
=       =       =       ECX=7 ESI=8049635 EDI=8049641
=       =       =       ECX=6 ESI=8049636 EDI=8049642
=       =       =       ECX=5 ESI=8049637 EDI=8049643
```

```
=     =     =     ECX=4 ESI=8049638 EDI=8049644
=     =     =     ECX=3 ESI=8049639 EDI=8049645
=     =     =     ECX=2 ESI=804963A EDI=8049646
=     =     =     ECX=1 ESI=804963B EDI=8049647
=     =     =      ECX=0 ESI=804963C EDI=8049648 EFL=200246
RF=0
00000F34 .text:080483E1 push    0   ESP=BFFCFBD8
```

Мыупомянемещедвеопции - *Trace over debugger segments*и*Trace over library functions*. При включении первой их них трассировка инструкций и функций временно приостанавливается каждый раз, когда выполнение переходит к сегменту программы, который не находится внутри какого-либо сегмента загруженного в IDA. Самым простым примером подобной ситуации является вызов функции разделяемой библиотеки. При выборе опции Trace over library functions трассировка временно приостанавливается каждый раз, когда IDA определяет что выполняется библиотечная функция. Не путайте библиотечные функции, связанные с бинарным файлом и функции разделяемых библиотек, таких как DLL. Обе эти опции включены по умолчанию, что дает лучшую производительность при трассировке (так как отладчику не требуется следить за библиотечным кодом), и также уменьшает количество записываемых в лог событий, что не позволит библиотечному коду заполнить буфер событий.

5.3.3. Трассировка стека

Трассировка стека отображает все содержимое стека вызовов или последовательностей вызовов функций, сделанных для того,чтобы достичь определенную область в бинарном файле. На рис.5.9 показан пример трассировки стека сгенерированной при помощи команды Debugger ► Stack Trace.

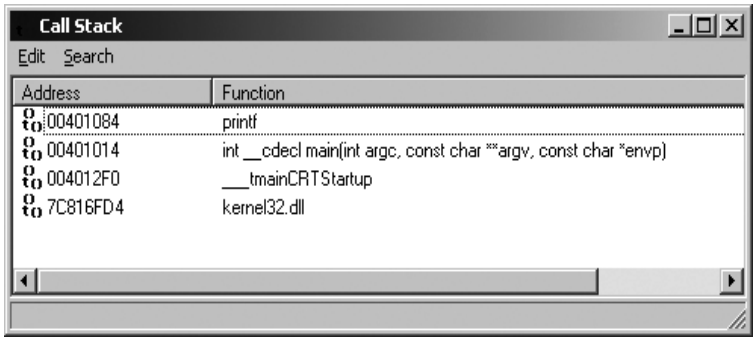


Рис.5.9. Пример трассировки стека

В верхней строке отображается имя функции, которая выполняется в данный момент. Во второй строке отображается функция, которая произвела вызов текущей функции и адрес, с которого она это сделала. Последующие строки отображают, откуда была вызвана каждая функция. Отладчик может производить трассировку стека путем просмотра встреченных им стековых кадров. Отладчик IDA использует содержимое регистра указателей кадров (EBP) для поиска каждого стекового кадра. После обнаружения стекового кадра, отладчик может извлечь указатель на следующий стековый кадр, также как и сохраненный адрес возврата, который используется для поиска инструкции, вызовшую текущую функцию. Отладчик IDA не может производить трассировку через стековые кадры, которые не используют EBP в качестве указателя кадров. На уровне работы с функциями, трассировка стека может быть полезна для ответа на вопрос «Как я сюда попал?», или более корректно «Какая последовательность функций привела к данной области?».

5.3.4. Watches

Во время отладки процесса у вас может появиться необходимость следить за значениями, содержащимися в одной или нескольких переменных. Вместо того, чтобы просматривать каждый раз необходимые области памяти на паузе, многие отладчики предоставляют возможность указывать список областей памяти, значения которых должны отображаться каждый раз при остановке процесса. Такие списки называются watch-листами, так как они позволяют вам следить за содержимым необходимых областей памяти

и его изменением во время выполнения программы. Такие списки удобны для навигации, они не устанавливают паузу в отличие от точек остановки.

Так как основной целью watch-точек являются данные, то они чаще всего устанавливаются в стеке, куче или секциях данных бинарного файла. Watches устанавливаются в отладчике IDA при помощи клика правой кнопкой мыши по интересующей вас области в памяти и выбора Ass Watch. Список всех watches может быть открыт через меню Debugger ► Watches ► Watch List. Отдельные watch можно удалять при помощи выбора их в списке и нажатия кнопки DELETE.

5.4. Автоматизация задач отладчика

Вы рассмотрели базовые скриптовые возможности IDA и IDA SDK, надеемся, что нам удалось показать полезность данных способностей при статическом анализе бинарных файлов. Скрипты и дополнения IDA являются не менее полезными, и при динамическом анализе с помощи отладчика. Вот несколько интересных применений: анализ данных во время выполнения отлаживаемого процесса, реализация точек остановки с комплексными условиями, а также использование мер противодействия антиотладочным методам.

5.4.1. Создание скриптов действия при помощи IDC

Все скриптовые возможности IDC, ранее рассмотренные в Главе 5, доступны и при использовании отладчика IDA. Скрипты могут быть запущены из меню File, при помощи горячих клавиш, а также они могут быть вызваны из командной строки IDA. В дополнение, пользовательские функции IDC могут использоваться в условиях точки остановки и в выражениях трассировки.

До версии 5.2 IDC предоставлялась только ограниченная поддержка взаимодействия скриптов и отладчика. В возможности IDC входили: установка, модификация и нумерация точек остановки; чтение и запись регистром и значений памяти. Доступ к памяти осуществляется при помощи функций Byte, PatchByte, Word, PatchWord, Dword и PatchDword описанных в Главе 5. Манипуляции с регистрами и точками остановки осуществляются при помощи следующих функций (полный список можно посмотреть в справочном файле IDA):

- long GetRegValue(string reg)

Возвращается значение указанного регистра (например, EAX). Значения регистров могут также быть получены при использовании имени необходимого регистра в выражении IDC.

- bool SetRegValue(number val, string name)

Устанавливает значение указанного регистра. Значения регистром можно также изменять напрямую, путем использования имени необходимого регистра в левой части выражения присвоения.

- bool AddBpt(long addr)

Добавляет программную точку остановки по указанному адресу.

- bool AddBptEx(long addr, long size, long type)

Добавляет точку остановки указанного размера и типа по заданному адресу. Типом точки должна быть одна из констант BPT_xxx, описанных в idc.idc или справочном файле IDA.

- bool DelBpt(long addr)

Удаляет точку остановки по указанному адресу.

- long GetBptQty()

Возвращает число точек остановки установленных в программе.

- long GetBptEA(long bpt_num)

Возвращает адрес, в котором установлена указанная точка остановки.

- long/string GetBptAttr(long addr, number attr)

Возвращает атрибут точки остановки, связанный с точкой установки расположенной по указанному адресу. Возвращаемое значение может быть числом или строкой, в зависимости от того, значение какого атрибута было запрошено.

- bool SetBptAttr(long addr, number attr, long value)

Устанавливает значение указанного атрибута заданной точки остановки. Не используйте эту функцию для установки выражения условий точки остановки (вместо этого используйте SetBptCnd).

- bool SetBptCnd(long addr, string cond)

Задает условия точки остановки при помощи указанного выражения IDC.

Следующий скрипт отображает, как можно установить произвольную функцию обработки точки остановки в области курсора:

```
#include <idc.idc>
/*
 * The following should return 1 to break, and 0 to continue execution.
 */
static my_breakpoint_condition() {
return AskYN(1, "my_breakpoint_condition activated, break now?") == 1;
}
/*
 * This function is required to register my_breakpoint_condition
 * as a breakpoint conditional expression
 */
static main() {
auto addr;
addr = ScreenEA();
AddBpt(addr);
SetBptCnd(addr, "my_breakpoint_condition()");
}
```

Насколько сложным будет выражение my_breakpoint_condition, зависит полностью от вас. В данном примере каждый раз при встрече новой точки остановки открывается диалог, в котором пользователь должен указать хочет ли он продолжить выполнение или поставить процесс на паузу в текущей области. Возвращаемое функцией my_breakpoint_condition значение, используется отладчиком для определения необходимости пропустить точку.

Начиная с версии 5.2 в IDA стали доступны возможности синхронного взаимодействия скриптов IDC с отладчиком. Таким образом, вместе с появлением IDA 5.2 появилось и значительное число новых функций IDC, предназначенных специально для взаимодействия с отладчиком, таких как возможность запускать отладчик, и следить за отладкой из скрипта IDC. Базовый подход при управлении отладчиком заключается в инициализации некоторого действия отладчика, и затем в ожидании кода соответствующего события. Не забывайте, что вызов синхронной функции отладчика блокирует все остальные операции IDA до завершения этого вызова. Вотнескольконовыхрасширений IDC:

long GetDebuggerEvent(long wait_evt, long timeout)
Ожидает, что в течение некоторого указанного времени (-1 означает бесконечность) произойдет определенное событие (wait_evt). Возвращает код типа события, который обозначает, какое событие произошло. Указать wait_evt можно с использованием флагов WFNE_xxx. Возможные возвращаемые значения можно узнать в справочном файле IDA.

- bool RunTo(long addr)

Запускает выполнение процесса до указанной области или до встречи точки остановки.

- bool StepInto()

Пошагово выполняет одну инструкцию, с возможностью входа в вызванные функции.

- bool StepOver()

Пошагово выполняет одну инструкцию, пропуская вызванные функции. Данной вызов может завершиться преждевременно при встрече точки остановки.

- bool StepUntilRet()

Выполняет код функции до ее завершения или до встречи точки остановки.

- bool EnableTracing(long trace_level, long enable)

Включает (или отключает) создание событий трассировки. Параметр trace_level должен быть равен одной из констант trACE_xxx определенных к idc.idc.

- long GetEventXXX()

Существует несколько функций при помощи, которых можно получить информацию касательно текущего события отладки. Некоторые из этих функций доступны для использования только с определенными типами событий. Вам необходимо проверить возвращаемое GeTDebuggerEvent значение для того, чтобы убедиться, что GetEventXXX доступно.

GetdebuggerEvent должна вызываться после каждой функции, заставляющей процесс выполняться для того, чтобы получить код

события отладчика. Если этого не сделать, то последующие попытки выполнения процесса могут быть невыполнимы. Например, в следующем фрагменте кода будет выполнен только один шаг из-за того что GeTDebuggerEvent не была вызвана для очистки последнего типа события между вызовами StepOver.

```
StepOver();
StepOver();    //this and the following calls will fail
StepOver();
StepOver();
```

В следующем примере показано, как правильно производить пошаговое выполнение процесса с помощью GetdebuggerEvent:

```
StepOver();
Get DebuggerEvent(WFNE_SUSP, -1);
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
StepOver();
GetDebuggerEvent(WFNE_SUSP, -1);
```

Вызовы функции GeTDebuggerEvent позволяют продолжить выполнение, даже если вам не требуется возвращаемое от них значение. Тип события WFNE_SUSP обозначает, что мы ожидаем события остановки отлаживаемого процесса из-за исключения или точки остановки. Вы могли заметить, что функции простого запуска остановленного процесса не существует. Однако, запустить процесс можно при помощи флага WFNE_CONT в вызове GetdebuggerEvent, как показано в примере:

```
GetDebuggerEvent(WFNE_SUSP | WFNE_CONT, -1);
```

В данном случае вызов ожидает первого события остановки после запуска процесса с текущей области.

Существуют дополнительные функции IDC, которые используются для автоматического запуска отладчика, и присоединения к выполняемому процессу. Для получения более подробной информации об этих функциях смотрите справочный файл IDA.

Ниже приведен пример скрипта для сбора статистики по адресам, с которых извлекаются инструкции (предполагается, что отладчик запущен):

```
static main() {
auto ca, code, addr, count, idx;
  ca  (1) = GetArrayId("stats");
if (ca      != -1) {
DeleteArray(ca);
  }
  ca       = CreateArray("stats");
      (2)                EnableTracing(TRACE_STEP, 1);
for       (code   = (3) GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -
1); code > 0;
code       = GetDebuggerEvent(WFNE_ANY | WFNE_CONT, -1)) {
addr = (4)      GetEventEa();
count       (5) = GetArrayElement(AR_LONG, ca, addr) + 1;
SetArrayLong(ca,       (6) addr, count);
  }
EnableTracing(TRACE_STEP, 0);
    (7) for (idx = GetFirstIndex(AR_LONG, ca);
        idx != BADADDR;
idx = GetNextIndex(AR_LONG, ca, idx)) {
count = GetArrayElement(AR_LONG, ca, idx);
Message("%x: %d\n", idx, count);
}
   (8)  DeleteArray(ca);
       }
```

В начале скрипта производится проверка существования глобального массива с именем stats. Если данный массив обнаружен, то он удаляется и создается заново, для того чтобы при запуске он был пуст. Следующим шагом является включение пошаговой трассировки до входа в цикл совершения шагов. Каждый раз при создании события отладчика извлекается адрес связанного события, текущий счет для связанного адреса извлекается из глобального массива и инкрементируется, после чего в массив записывается новое значение. Отметим, что указатель инструкций используется в качестве индекса для глобального массива, что экономит время в сравнении с поиском адреса в какой-либо структуре. После завершения процесса запускается второй цикл дляизвлечение и вывода всех значений массива, которые являются действительными. В данном случае только

действительные значения в массиве отображают адреса, из которых были вызваны инструкции. Пример вывода данного скрипта:

```
401028: 1
40102b: 1
40102e: 2
401031: 2
401034: 2
401036: 1
40103b: 1
```

Немного изменив предыдущий скрипт, его можно будет использовать для сбора статистики типов функций, выполненных во время существования процесса. В следующем примере показано, какие изменения требуется произвести в первом цикле для того, чтобы собирать данные о типах инструкций вместо данных об адресах.

```
for (code = Get DebuggerEvent(WFNE_ANY | WFNE_CONT, -1); code >
0;
code = Get DebuggerEvent(WFNE_ANY | WFNE_CONT, -1)) {
Message("code = %d\n", code);
addr = GetEventEa();

mnem = GetMnem(addr);

count = GetHashLong(ht, mnem) + 1;

SetHashLong(ht, mnem, count);
}
```

Вместо того чтобы пробовать произвести классификацию опкодов, мы предпочитаем группировать инструкции по мнемоникам. Из-за того что мнемоники являются строками, мы используем хэш-таблицы глобальных массивов для извлечения текущего счета, связанного с данной мнемоникой и записи измененного значения обратно в корректную запись в хэш-таблице. Вывод измененного скрипта:

```
add:  18
and:  2
call:  46
cmp:  16
```

```
dec:  1
imul: 2
jge:  2
jmp:  5
jnz:  7
js:   1
jz:   5
lea:  4
mov:  56
pop:  25
push: 59
retn: 19
sar:  2
setnz: 3
test: 3
xor:  7
```

В Главе 6 мы снова обратимся к скриптовым возможностях IDC при взаимодействии с отладчиком для помощи при деобфускации бинарных файлов.

5.4.2 Автоматизация Действий Отладчика при помощи Дополнений IDA

IDA SDK предоставляет огромные возможности для разработки различных скомпилированных расширений, которые могут быть интегрированы в IDA и иметь полный доступ к API IDA. В IDA API присутствует набор возможностей доступных в IDC, и отладочные расширения не являются исключением. Отладочные расширения к API описаны в <SDKDIR>/dbg.hpp, и включают C++, копии всех функций IDC описанных ранее, также присутствует полная поддержка асинхронной отладки.

API отладчика появились в IDA версии 4.6, и это было единственным средством взаимодействия с отладчиком вплоть до IDA версии 5.2. Для асинхронного взаимодействия дополнения получают доступ к уведомлениям отладчика при помощи типа уведомлений HT_DBG (смотрите loader.hpp). Уведомления отладчика описываются в перечислении dbg_notification_t, находящемся в dbg.hpp.

В API отладчика команды, взаимодействия с отладчиком обычно описываются парами, где одна функция используется для синхронного взаимодействия, а вторая для асинхронного. Обычно функции для синхронного взаимодействия именуются в виде

COMMAND(), в то время как функции для асинхронного взаимодействия именуются request_COMMAND(). Версии вида request_XXX используются для помещения в очередь отладчика действий для дальнейшей обработки. После того как вы закончите формировать очередь асинхронных запросов, вы должны вызвать функцию для инициации процесса обработки вашей очереди запросов. Во время обработки очереди запросов, уведомления отладчика будут отправляться всем callback-функциям, зарегистрированным при помощи hook_to_notification_point.

Мы можем разработать асинхронную версию скрипта подсчета адресов из предыдущего раздела при помощи асинхронных уведомлений. Первым заданием является настройка hooking и unhooking уведомлений отладчика. Мы сделаем это при помощи методов init и term как показано далее:

```
//A netnode to gather stats into
netnode stats("$ stats", 0, true);
intidaapi init(void) {
  hook_to_notification_point(HT_DBG, dbg_hook, NULL);
return PLUGIN_KEEP;
}
void idaapi term(void) {
  unhook_from_notification_point(HT_DBG, dbg_hook, NULL);
}
```

Отметим, что мы также решили описать глобальную netnode, которую мы будем использовать для сбора данных. Далее мы хотим, чтобы дополнение активировалось при помощи назначенной горячей клавиши. Функция run нашего дополнения-примера:

```
voididaapirun(intarg) {
stats.altdel();   //clear any existing stats
   request_enable_step_trace();
   request_step_until_ret();
run_requests();
}
```

Так как мы используем асинхронный метод в данном примере, то мы должны сначала подать запрос для включения пошаговой трассировки, и после этого подать запрос для повторного запуска выполнения процесса. Из соображений простоты мы будем собирать статистику только по текущей функции, поэтому мы подадим запрос

для запуска до завершения текущей функции. После формирования очереди, мы вызываем run_requests для ее обработки:

Все что осталось сделать, это обработать уведомления, которые мы ожидаем получить создав callback-функцию HT_DBG. Простая функция обрабатывающая только два сообщения показана ниже:

```
int idaapi dbg_hook(void *user_data, int notification_code, va_list va) {
switch (notification_code) {
case dbg_trace:  //notification arguments are detailed in dbg.hpp
        va_arg(va, thid_t);
ea_t ea = va_arg(va, ea_t);
        //increment the count for this address
stats.altset(ea, stats.altval(ea) + 1);
return 0;
case dbg_step_until_ret:
      //print results
for (nodeidx_t i = stats.alt1st(); i != BADNODE; i = stats.altnxt(i)) {
msg("%x: %d\n", i, stats.altval(i));
        }
        //delete the netnode and stop tracing
stats.kill();
     request_disable_step_trace();
run_requests();
        break;
   }
}
```

Уведомление dbg_trace будет получено для каждой инструкции, которая выполняется до того как мы отключим трассировку. При получении уведомления трассировки, адрес точки трассировки извлекается из списка args, и затем используется для обновления соответствующего индекса в массиве netnode. Уведомление dbg_step_until_ret отправляется после того, как процесс получит утверждение return и выйдет из функции. Данное уведомление является сигналом к тому, чтобы прекратить трассировку и вывести собранную статистику. Далее используется цикл для просмотра всех действительных значений stats netnode до уничтожения netnode и запроса остановки пошаговой трассировки. Так как в данном примере используются асинхронные команды, то в очередь добавляется запрос на остановку трассировки, и подразумевает, что мы должны выполнить run_requests для того, чтобы очередь была обработана. Предупреждаем,

нельзя использовать синхронную версию функции, в то время как процесс обрабатывает асинхронное уведомление.

Синхронное взаимодействие SDK с отладчиком выполняется аналогично скриптовому взаимодействию с IDC. Как и в случае с функциями SDK, упомянутыми в предыдущих главах, имена многих функций SDK не совпадают с именами функций IDC, поэтому вам может потребоваться некоторое время для того, чтобы найти необходимые функции в dbg.hpp.

Главным несовпадением имен является функция GeTDebuggerEvent IDC, которая в SDK называется wait_for_next_event. Существует также еще одно существенное отличие между SDK и IDC — переменные соответствующие регистрам процессора не показываются автоматически. Для того, чтобы получить значения регистров процессора из SDK, вам необходимо использовать get_reg_val и set_reg_val для чтения и записи соответственно.

ГЛАВА 6. СОЧЕТАНИЕ ОТЛАДЧИКА И ДИЗАССЕМБЛЕРА

Сочетание дизассемблера и отладчика может быть довольно мощным инструментом для проведения различных манипуляций с бинарными файлами, и применения различных статических и динамических методов при проведении реверс-инжиниринга. Однако необходимо знать способности и ограничения каждого инструмента как отдельно, так и вместе.

В данной главе мы обсудим несколько важных вещей, касающихся того, как именно статическая сторона IDA взаимодействует с ее динамической стороной. Мы рассмотрим методы, которые могут использоваться при помощи отладчика IDA для обхода антиотладочной (и антидизассемблерной) защиты, которая может применяться во вредоносном ПО. Важно помнить, что основная цель при анализе вредоносного ПО, это не его запуск, а получение дизассемблерного кода, достаточного для использования инструментов статического анализа. Как вы можете помнить из обсуждения, существует много методов разработанных специально для того, чтобы помешать дизассемблеру, полноценно выполнять свою задачу. При работе с подобными методами, отладчик является просто одним из средств для достижения цели. Мы запустим обсуфцированную программу под контролем отладчика, и попробуем получить деобфусцированную версию этой программы, которую намного более предпочтительно анализировать при помощи дизассемблера.

6.1. Обфускация

Для начала нам следует немного рассказать о деобфускации. Для того чтобы обфусцированная программа могла выполнять свою задачу, ей необходимо деобфусцировать себя. Далее приведена последовательность пунктов, которая представляет из себя упрощенную инструкцию по динамическому деобфусцированию бинарных файлов.

- Откройте обфусцированную программу при помощи отладчика.
- Найдите процедуру деобфускации и устновите точку остановки в ее конце.
- Запустите программу из отладчика, и подождите, пока сработает точка остановки.
- Воспользуйтесь средствми отладчика для создания дампа памяти текущего состояния процесса и получения его файла.
- Завершите процесс до того как он сможет сделать что-нибудь вредоносное.
- Выполните статический анализ полученного образа процесса.

Большинство современных отладчиков имеют достаточно возможностей для выполнения подобного задания. В таких случаях часто используется популярный отладчик OllyDbg, предназначенный для работы под Windows. Пункт 2 на самом деле не всегда является таким простым, каким он кажется. Для его выполнения могут потребоваться различные инструменты, также некоторое время для разбора в дизассемблере, или даже долгое пошаговое выполнение — все для того, чтобы правильно определить конец процедуры деобфускации. Во многих случаях конец процедуры деобфускации определяется не по какой-то конкретной инструкции, а по особому поведению. Таким поведением может быть большое изменение значения указателя инструкций, показывающее переход к области, находящейся далеко от кода деобфускации. Например, при работе с бинарными файлами, упакованными UPX, все, что вам требуется это наблюдать за указателем инструкций, когда он примет значение меньшее, чем адрес точки входа программы, то значит, что процедура деобфускации завершена, и осуществлен переход к только что деобфусцированному коду. Данный процесс называется процессом поиска оригинальной точки входа (original entry point, OEP), которая является адресом, с которого программа начала бы свое выполнение, в случае если бы не была обфусцирована.

Нужно быть очень осторожным при выполнении 3 пункта. Вам необходимо сначала дважды подумать, перед тем как позволить беспрепятственно выполниться части вредоносного ПО, и надеяться, что все точки остановки установлены и настроены правильно. Если программа пропустит вашу точку остановки, то вредоносный код может быть выполнен до того как вы поймете что это произошло. Исходя из соображений безопасности, попытки деобфусцировать вредоносное ПОпод контролем отладчика, должны всегда проводиться в песочнице, которую будет не жалко полностью вычистить, в случае если что-то пойдет не так.

Для выполнения 4 пункта может потребоваться определенный уровень усилий, так как, несмотря на то, что создание дампа памяти поддерживается в большинстве отладчиков, создание дампа всего образа процесса чаще всего нет. Дополнение OllyDump (автор Gigapede) ждя OllyDbg позволяет создавать дамп всего процесса. Не забывайте учитывать то, что дамп образа процесса из памяти может содержать совсем не то, что хранится в оригинальном бинарном файле на диске. Однако, при анализе вредоносного ПО, главной целью не является создание рабочего деобфусцированного выполняемого файла, главная цель — получение файла с правильной структурой, которым можно воспользоваться при дальнейшем анализе в дизассемблере.

Одной из самых сложных частей воссоздания бинарного образа из обфусцированного процесса, является восстановление таблицы импортированных функций. Таблица импортированных функций также проходит процесс обфускации. В результате получается, что необходимо также осуществить привязку деобфусцированного процесса ко всем разделяемым библиотекам и функциям, которые требуются процессу для полноценной работы. Единственной зацепкой в данном случае является таблица адресов импортированных функций, которая находится где-то в памяти процесса. При дампе деобфусцированного процесса обычно предпринимаются шаги для восстановления действительной таблицы импорта в образе процесса. Для этого требуется, чтобы заголовки сохраненного образа были изменены для указания на новую структуру таблицы импорта, которая должным образом отражает все зависимости разделяемых библиотек в оригинальной программе. Для автоматизации данного процесса можно воспользоваться утилитой ImpREC (Import REConstruction) от MackT.

6.2. Базы данных IDA и отладчик

Для начала важно понимать, как отладчик воспринимает вашу базу данных при инициации (и завершении) сессии отладки. Отладчику необходим образ, с которым он может работать. Данный образ отладчик может получить либо присоединившись к существующему процессу, либо создав новый процесс из исполняемого файла. В базе данных IDA не содержится образ процесса, и в большинстве случаев он не может быть воссоздан из базы данных (если может, то можно легко выполнить File ►4 Produce File ► 4 Create EXE File). При запуске сессии отладки из IDA, дизассемблер сообщает отладчику имя оригинального файла, который отладчик использует для создания и присоединения к новому процессу. Также передается информация, касающаяся формата дизассемблирования и символьных имен. Любые комментарии, которые вы вводили в режиме дизассемблера, будут недоступны и, что более важно, все патчи (изменения содержимого байт), которые вы применили к базе данных - не будут отражены в отлаживаемом процессе. Другими словами, невозможно пропатчить базу данных, и наблюдать эффект от данных изменений при запуске отладчика.

Верно также и противоположное. После завершения отладки процесса и возврата в режим дизассемблера, будут отражены только косметические изменения (такие как переименованные переменные или функции). Любые изменения в памяти, такие как самоизменяющийся код, не возвращаются в базу данных для анализа. Вам может потребоваться добавить новый деобфусцированный код обратно в базу данных для его анализа без повторного запуска. Для этого предназначена команда File ►4 Produce File ► 4 Create EXE File. Диалог подтверждения отображен на рис.6.

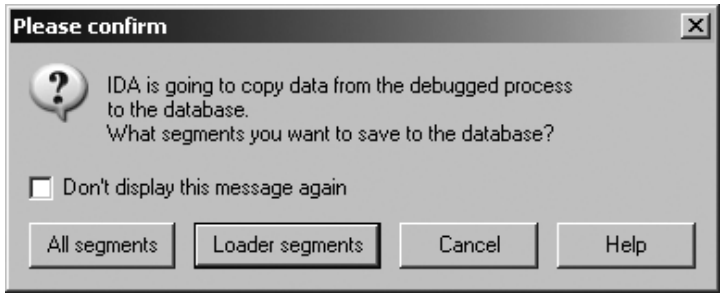


Рис.6. Диалог подтверждения создания снимка

При настройках по умолчанию в базу данных будут скопированы сегменты загрузчика из выполняемого процесса. Сегменты загрузчика - это такие сегменты, которые загружаются в базу данных одним из загрузочных модулей IDA. В случае работы с обфусцированной программой, один или несколько таких сегментов, скорее всего, содержат обфусцированные данные, и поэтому анализ в дизассемблере практически не представляется возможным. Это именно те сегменты, которые вы захотите скопировать обратно из выполняемого процесса для того, чтобы использоваться преимущества деобфускации выполненный процессом.

Если выбрать All segments, то все созданные отладчиком сегменты будут скопированы обратно в базу данных. Это включает содержимое всех разделяемых библиотек загруженных процессом, а также дополнительные сегменты отладчика, такие как содержимое стека.

В случае если отладчик присоединился к уже существующему процессу без соответствующей базы данных, тогда не будет отмечен ни один сегмент отладчика как сегмент загрузчика, так как файл не был загружен одним из загрузчиков IDA. В данном случае, вы можете выбрать загрузку всех доступных сегментов в базу данных. Или вы можете изменить атрибуты сегментов, и отметить необходимые как сегменты загрузчика. Атрибуты сегментов могут быть отредактированы в окне Segments (View ► 4 Open Subviews ► 4 Segments), необходимо кликнуть правой кнопкой мыши по интересующему вас предмету, и выбрать Edit Segment. Данный диалог показан на рис.6.1. Если выбрать чекбокс Loader segment, то данный сегмент будет восприниматься, как сегмент загрузчика, и будет скопирован в базу данных вместе с остальными сегментами загрузчика.

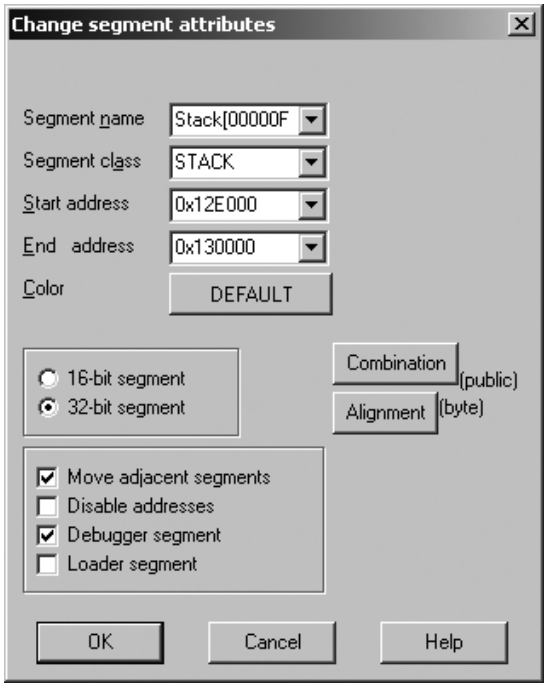


Рис.6.1. Диалог редактирования сегмента

6.3. Отладка обфусцированного кода

Мы множество раз упоминали, что хорошей стратегией получения деобфусцированной версии программы, является загрузка обфусцированной программы в отладчик, выполнение до завершения процедуры деобфускации, и затем снятие снимка деобфусцированной программы. В данном случае, скорее стоит называть это контролируемым выполнением, а не отладкой, так как то, что мы на самом деле делаем, это наблюдаем за кодом, и затем делаем снимок памяти в подходящий момент. А отладчик просто является инструментом, при помощи которого мы можем это сделать. Ну, или как минимум мы надеемся, что он таковым является. Мы изучили множество антидизассемблерных и антиотладочных методов, которые

используют обфускаторы для того, чтобы помешать нам получить четкий снимок программы. Так давайте же теперь посмотрим, как отладчик IDA может нам помочь в обходе данных методов.

В этой главе мы предполагаем, что обфусцированные программы, с которыми мы имеем дело, применяют какую-нибудь форму шифрования, или компрессии самых интересных частей бинарного файла. Уровень сложности получения четкого представления кода зависит целиком от сложности методов антианализа, использованных при обфускации процесса и средств необходимых для их преодоления. Перед тем как мы начнем, хотелось бы привести несколько важных правил для работы с вредоносным ПО:

- Обеспечьте защиту сети. Всегда работайте в окружении песочницы.
- Вначале постарайтесь использовать пошаговое выполнение. Это может быть довольно утомительным, но это наилучший способ держать программу под контролем.
- Дважды подумайте, перед тем как запускать команду отладчика, которая позволит выполнить сразу несколько инструкций. Если не быть осторожным, то можно позволить программе выполнить вредоносную часть кода.
- Старайтесь пользоваться аппаратными точками остановки. Программные точки установки довольно тяжело устанавливать в обфусцированном коде, т.к. алгоритмы деобфускации могут изменить инструкции, вставленной вами точки остановки.
- При первоначальном изучении программы лучше всего позволить отладчику обрабатывать все исключения, сгенерированные программой для того, чтобы вы смогли принимать обдуманное решение, какие исключения позволить пропустить, а какие отладчик должен будет продолжать обрабатывать.
- Будьте готовы к тому, что нужно будет часто перезапускать отладку, так как один неверный шаг может все испортить (например, если вы позволите процессу определить отладчик). Отмечайте безопасные для выполнения адреса для того, чтобы проще было вернуться после перезапуска.

В общем, постарайтесь всегда осознанно подходить к работе с обфусцированной программой первый раз. В большинстве

случаев,вашей основной целью будет получение деобфусцированной версии программы. Ускорение процесса деобфускации, основанное на знании того, как далеко нужно зайти в программе до установки точки остановки - является второстепенной целью, и этим можно заняться после того как вы сможете удачно деобфусцировать программу в первый раз.

6.3.1. Простые циклы расшифровки и декомпрессии

Под простыми циклами дешифровки и декомпрессии мы подразумеваем такие циклы, в которых не используются методы вложенной обфускации, и у которых можно точно определить все точки выхода. Когда вы встречаете такой цикл, то лучше всего установить точки остановки во всех возможных точках выхода, и затем выполнить цикл. Попробуйте сначала выполнить такой цикл один-два раза в пошаговом режиме для того, чтобы лучше понять его, и затем устанавливайте точки остановки. При установке точки остановки сразу после цикла убедитесь, что по адресу установки точке не будет происходить никаких изменений во время цикла, иначе программная точка остановки может не сработать. Если вы сомневаетесь, то используйте аппаратную точку остановки.

Если вашей основной задачей является полная автоматизация процесса деобфускации, то вам необходимо разработать алгоритм распознавания окончания процедуры деобфускации. При выполнении этого условия ваш алгоритм может остановить процесс и сделать снимок памяти текущего процесса. В случае работы с простыми алгоритмами деобфускации, распознать конец процедуры может быть очень просто по большому изменению значения указателя инструкций, или по запуску особой инструкции. Например, начало и конец процедуры декомпрессии UPX показаны далее:

```
        UPX1:00410370 start proc near
    ❶   UPX1:00410370 pusha
        UPX1:00410371 mov     esi, offset off_40A000
UPX1:00410376 lea     edi, [esi-9000h]
UPX1:0041037C push    edi
...
        UPX1:004104EC pop     eax
    ❷   UPX1:004104ED popa                    ; opcode 0x53
        UPX1:004104EE lea     eax, [esp-80h]
UPX1:004104F2
UPX1:004104F2 loc_4104F2:                 ; CODE XREF: start+186ïfj
```

```
UPX1:004104F2 push    0
UPX1:004104F4 cmp     esp, eax
UPX1:004104F6 jnz     short loc_4104F2
        UPX1:004104F8 sub     esp, 0FFFFFF80h
    (3) UPX1:004104FB jmp     loc_40134C
```

Различные характеристики данной процедуры могут использоваться для распознавания ее завершения. Во-первых, происходит сохранение в стек всех регистров в начальной точке программы (1). Операция выгрузки значений регистров из стека производится ближе к концу процедуры (2), после того как программа была распакована. Следовательно, первоначальная стратегия для автоматизации процесса — пошаговое выполнение программы до встречи инструкции popa. Из-за того что пошаговое выполнение довольно медленное, то в примере IDC, показанном на рис.6.2, используется немного иной подход к сканированию инструкции popa, и дальнешему выполнению программы до ее адреса:

Листинг 6.2. Простой скрипт распаковщика UPX

```
#include <idc.idc>

 #define POPA 0x53

 static main() {
   auto addr, seg;
   addr = BeginEA();   //Obtain the entry point address
   seg = SegName(addr);
    (2)          while (addr != BADADDR && SegName(addr) == seg) {
          (3)       if (Byte(addr) == POPA) {
              RunTo(addr);
                        (4) GetDebuggerEvent(WFNE_SUSP, -1);
Warning(
"Program  (5)
is
unpacked!");
              TakeMemorySnapshot(1);
              return;
      }
          (1) addr = FindCode(addr, SEARCH_NEXT | SEARCH_DOWN);
   }
Warning("Failed to locate popa!");
```

```
}
```

Скрипт, приведенный в примере 6.2, предназначен для запуска в базе данных IDA до запуска отладчика. Скрипт самостоятельно запускает отладчик, и получает управление созданным процессом. Скрипт использует некоторые особенности UPX, и поэтому не является хорошим вариантом для использования в качестве общего скрипта для деобфускации. Скрипт полагается на то, что процедура декомпрессии располагается в конце одного их сегментов, (обычно называемого UPX1) и на то, что UPX не использует методы десинхронизации для предотвращения нормального дизассеблирования.

На сегодняшний день UPX является одной из самых популярных утилит для обфускации (наверное, потому что он бесплатен). Однако такая популярность не обозначает высокую эффективность. Одним из основных его недостатков является то, что UPX предоставляет опцию командной строки, позволяющую привести запакованный бинарный файл к оригинальному виду. После этого, конечно, появились кустарные разработки предотвращающие самораспаковку UPX. Из-за тогочто UPX проверяет целостность файла перед распаковкой, то достаточно помешать этой проверке, и функция самораспаковки будет недоступна. Помешать проверке можно изменив имя стандартных секций UPX на что-либо отличное от UPX0, UPX1 и UPX2. По этой причине старайтесь не задавать в своих скриптах эти имена напрямую.

Скрипт полагается на эти факты при сканировании, по одной инструкции за раз, начиная с точки входа программы, до тех пор, пока следующая инструкция находится в том же сегменте программы и до встречи инструкции рора. После того, как будет найдена инструкция рора, отладчик продолжит выполнение процесса до адреса этой инструкции, где программа будет распакована. Остается только последний шаг — сделать снимок памяти для того, чтобы вернуть деобфусцированную программу обратно в базу данных для дальнейшего анализа.

Существует также более широкое решение для автоматической распаковки — необходимо использовать тот факт, что большинство процедур деобфускации добавляются в конце бинарного файла, и после завершения процедуры деобфускации выполняют переход к оригинальной точке входа, которая расположена намного ближе к началу. В некоторых случаях оригинальная точка входа может располагаться в абсолютно ином сегменте программы, а в некоторых, она расположена перед процедурой деобфускации. Скрипт, приведенный в примере 11.2, представляет собой базовое средство для запуска простого алгоритма деобфускации до перехода к оригинальной точке входа программы:

Листинг 6.3. Выполнение до встречи OEP

```
static main() {
    auto start, code;
    start = BeginEA();
    RunTo(start);
    GetDebuggerEvent(WFNE_SUSP, -1);
    EnableTracing(TRACE_STEP, 1);
for (code = GetDebuggerEvent(WFNE_ANY | WFNE_CONT,
-1); code > 0;
         code    =    GetDebuggerEvent(WFNE_ANY    |
WFNE_CONT, -1)) {
       if (GetEventEa() < start) break;
    }
    PauseProcess();
    GetDebuggerEvent(WFNE_SUSP, -1);
    EnableTracing(TRACE_STEP, 0);
MakeCode(EIP);
    TakeMemorySnapshot(1);
 }
```

Аналогично скрипту, приведенному в листинге 6.2, данный скрипт необходимо запускать из дизассемблера, а не из отладчика. Скрипт самостоятельно выполняет запуск отладчика, и присоединяется к созданному процессу. Работа скрипта основывается на двух предположениях: на том, что весь код до точки входа является обфусцированным и на том, что не производится никаких вредоносных действий до точки входа. Скрипт запускает отладчик, и останавливает выполнение процесса на точке входа программы. Далее программа запускает пошаговую трассировку, и зацикливается для проверки каждого сгенерированного события. В случае если адрес события предшествует адресу точки входа программы, то считается что деобфускация завершена, и процесс останавливается. И в завершение скрипт проверяет, как байты по текущему адресу указателя инструкций представлены в формате кода.

Во время выполнения скрипта может быть отображено предупреждение, показанное на рис.6.4.

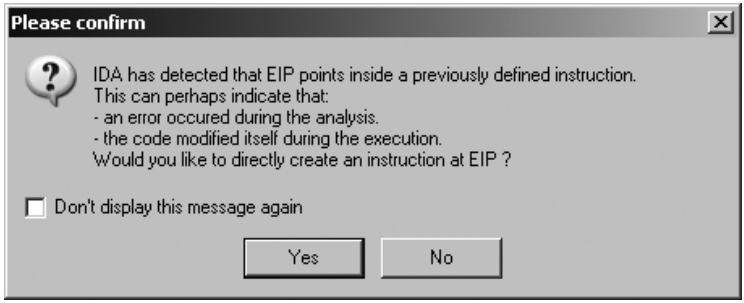


Рис.6.4. Предупреждение отладчика об указателе инструкций

Данное предупреждение показывает то, что указатель инструкций указывает на данные, или что он указывает в середину ранее дизассемблированной интрукции. Это предупреждение часто встречается при пошаговом выполнении кода, в котором был применен метод десинхронизации дизассемблирования. Такое часто встрчеается, когда программа совершает переход к области, в которой ранее находились данные, которые теперь стали кодом после деобфускации программы. При положительном ответе IDA переформатирует байты как код, что будет вполне логично и правильно, так как указатель инструкций указывает, что именно они должны быть выплонены следующими.

Заметим, что из-за того что используется пошаговая трассировка, скрипт будет работать гораздо медленнее чем приведенный в примере 6. Однако ценой подобного замедления мы получаем пару преимуществ. Во-первых, теперь мы можем ставить условие завершения, которое не привязано к конкретному адресу. Такое невозможно только с использованием точек остановки. Во-вторых, скрипт обладает «иммунитетом» к любым попыткам десинхронизации ассемблра, так как рамки инструкций определяются только за счет runtime-значений указателя инструкций, а не при помощи статического анализа. В своем объявлении, касающемся скриптовых возможностей отладчика, Ильфак представил гораздо более надежный скрипт для выполнения задач универсального распаковщика.

6.3.2. Восстановление таблицы импорта

После того как завершена деобфускация бинарного файла, можно приступать к его анализу. Может быть, у вас и нет намерения запускать деобфусцированную программу (на самом деле мы не смогли бы запустить программу, если снимок был загружен обратно в базу данных), но таблица импорта программы всегда является ценным ресурсом для понимания поведения программы.

При нормальных обстоятельствах IDA может провести парсинг таблицы импорта программы в качестве части процесса загрузки файла перед созданием базы данных. К сожалению, при работе с обфусцированной программой, единственная таблица импорта, которую обнаружит IDA, принадлежит к компоненту деобфускации этой программы. В большинстве случаев, таблица импорта также является обфусцированной, и в дальнейшем восстанавливается в процессе деобфускации. Чаще всего в процессе восстановления после получения деобфусцированных данных выполняется самостоятельная загрузка библиотек и разрешение адресов функций. При этом в программах для Windows почти всегда происходят вызовы функции LoadLibrary совместно с GetProcAddress для разрешения требуемых адресов функций.

Более продвинутая процедура восстановления таблицы импорта может использовать функции поиска вместо GetProcAddress для того, чтобы избежать срабатывания точек остановки, установленных на саму функцию GetProcAddress. Подобные процедуры также могут использовать значения хеша вместо строк для определения функции с запрашиваемым адресом. В редких случаях реконструкторы таблицы импорта могут обходить даже вызовы LoadLibrary , для этого в

процедуре восстановления должна быть реализована собственная версия этой функции.

Чистым результатом процесса восстановления таблицы импорта является таблица адресов функций, от которой не так уж и много толку при статическом анализе. Если сделать снимок памяти, то, скорее всего мы увидим что-то подобное следующему листингу:

```
UPX1:0040A000 dword_40A000     dd 7C812F1Dh          ; DATA XREF:
start+1o
UPX1:0040A004 dword_40A004     dd 7C91043Dh          ; DATA XREF:
sub_403BF3+68
UPX1:0040A004                          ; sub_405F0B+2B4r ...
UPX1:0040A008             dd 7C812ADEh
UPX1:0040A00C dword_40A00C     dd 7C9105D4h          ; DATA XREF:
sub_40621F+5Dr
UPX1:0040A00C                          ; sub_4070E8+Fr ...
UPX1:0040A010             dd 7C80ABC1h
UPX1:0040A014 dword_40A014     dd 7C901005h          ; DATA XREF:
sub_401564+34r
UPX1:0040A014                          ; sub_4015A0+27r ...
```

Данный блок данных отображает некоторое число 4-байтных значений, находящихся в непосредственной близости друг к другу, и к которым ведут ссылки с различных мест программы. Проблема в том, что данные значения, такие как 7C812F1Dh, отображают адреса библиотечных функций такими, какими они были размечены в отлаживаемом нами процессе. В сегменте кода самой программы вызовы функций выглядели бы приблизительно так:

```
UPX0:00403C5B          call    ds:dword_40A004
UPX0:00403C61          test    eax, eax
UPX0:00403C63          jnz     short loc_403C7B
UPX0:00403C65          call    sub_40230F
UPX0:00403C6A          mov     esi, eax
UPX0:00403C6C          call    ds:dword_40A058
```

Заметим, что два вызова функции ссылаются на содержимое восстановленной таблицы импорта, в то время как третий вызов функции ссылается на функцию, которая находится в базе данных. В идеальном случае каждая запись в восстановленной таблице импорта должна быть названа в соответствии с функцией, которую она содержит.

Данную проблему лучше всего решать до создания снимка памяти деобфусцированного процесса. Как показано далее, если мы просмотрим ту же область памяти из отладчика, то мы увидим совершенно иную картину. Так как у отладчика есть доступ к регионам памяти, в которых лежат функции, на которые происходит ссылка, то отладчик может отображать адреса (такие как 7C812F1Dh) в соответствии с символьными именами (в данном случае kernel32_GetCommandLineA).

```
UPX1:0040A000 off_40A000 dd offset kernel32_GetCommandLineA ; DATA XREF:UPX0:loc_40128Fr
UPX1:0040A000                                   ; start+1o
UPX1:0040A004 off_40A004 dd offset ntdll_RtlFreeHeap  ; DATA XREF: UPX0:004011E4r
UPX1:0040A004                                   ; UPX0:0040120Ar ...
UPX1:0040A008 off_40A008 dd offset kernel32_GetVersionExA ; DATA XREF: UPX0:004011D4r
UPX1:0040A00C dd offset ntdll_RtlAllocateHeap          ; DATA XREF: UPX0:004011B3r
UPX1:0040A00C                                   ; sub_405E98+Dr...
UPX1:0040A010 off_40A010 dd offset kernel32_GetProcessHeap; DATA XREF: UPX0:004011AAr
UPX1:0040A014 dd offset ntdll_RtlEnterCriticalSection; DATA XREF: sub_401564+34r
UPX1:0040A014; sub_4015A0+27r ...
```

На данный момент не важно, что отладчик использует немного иную схему именования, чем мы привыкли. Отладчик добавляет к каждой функции префикс, показывающий, из какой библиотеки эта функция была экспортирована. Например, функция GetCommandLineA из kernel32.dll будет иметь имя kernel32_GetCommandLineA.

Нам нужно обойти две проблемы с таблицей импорта, показанной в предыдущем листинге. Первое, для того чтобы вызовы функций были более читабельными, необходимо задать имя каждой записи в таблице импорта в соответствии с вызываемыми ими функциями. Если записям даны нормальные имена, то IDA сможет автоматически отображать сигнатуры функций из библиотек типов. Дать имена записям в таблице не так уж и сложно, особенно если известно какие давать. Это ведет ко второй проблеме — получение надлежащих имен. В первом варианте для решения этой проблемы необходимо взять имена, сгенерированные отладчиком, отбросить имя

библиотеки, и оставшееся назначить соответствующей записи в таблице. Есть маленькая неприятность в данном варианте — как имена функций, так и имена библиотек могут содержать символы подчеркивания, а это может усложнить определения точного имени функции в случае с длинной строкой. Однако, несмотря на подобную сложность, именно этот подход используется скриптом именования таблицы импорта renimp.idc, поставляемым вместе с IDA (находится в <IDADIR>/idc). Для того чтобы этот скрипт работал корректно, его необходимо запускать когда отладчик активен (чтобы у него был доступ к именам загруженных библиотек). Найдите таблицу импорта при помощи View ► Open Subview ► Imports. Далее выделите содержимое таблицы при помощи мыши от начала до конца таблицы. Скрипт renimp.idc выполняет итерации внутри выделения, выбирает имя функции, отбрасывает префикс с именем библиотеки, и соответственно именует записи в таблице импорта. После запуска скрипта, ранее приведенная таблица импорта принимает следующий вид:

```
UPX1:0040A000; LPSTR __stdcall GetCommandLineA ()
UPX1:0040A000 GetCommandLineA dd offset kernel32_GetCommandLineA
UPX1:0040A000 ; DATA XREF: UPX0:loc_40128Fr
UPX1:0040A000 ; start+1o
UPX1:0040A004 RtlFreeHeap dd offset ntdll_RtlFreeHeap; DATA XREF: UPX0:004011E4r
UPX1:0040A004 ; UPX0:0040120Ar...
UPX1:0040A008; BOOL __stdcall GetVersionExA (LPOSVERSIONINFOA lpVersionInformation)
UPX1:0040A008 GetVersionExA dd offset kernel32_GetVersionExA; DATA XREF: UPX0:004011D4r
UPX1:0040A00C RtlAllocateHeap dd offset ntdll_RtlAllocateHeap; DATA XREF: UPX0:004011B3r
UPX1:0040A00C ; sub_405E98+Dr...
UPX1:0040A010; HANDLE __stdcall GetProcessHeap ()
UPX1:0040A010 GetProcessHeap dd offset kernel32_GetProcessHeap; DATA XREF: UPX0:004011AAr
UPX1:0040A014 RtlEnterCriticalSection dd offset ntdll_RtlEnterCriticalSection
UPX1:0040A014 ; DATA XREF: sub_401564+34r
UPX1:0040A014; sub_4015A0+27r ...
```

В приведенном листинге скрипт выполнил работу по переименованию каждой записи в таблице импорта, но IDA еще добавила прототипы функций к каждой функции, о которой ей было известно. Никакой информации о типах не будет отображаться, в случае если префикс библиотеки не был отброшен. Скрипт renimp.idc может совершить ошибку при извлечении имени функции, если ее имя содержит подчеркивание. Библиотека ws2_32 является известным примером модуля, содержащего подчеркивание в имени. В скрипте renimp.idc происходит дополнительная обработка ws2_32. Однако, любой другой модуль с подчеркиванием в имени помешает корректному парсингу функций имен.

Альтернативный подход для получения информации о функциях требует поиска файловых заголовков в памяти, связанных с адресами функций, и затем парсинга таблицы экспорта, описанной в этих заголовках, для обнаружения имен необходимых функций. По сути, это является обратным поиском имени функции, зная ее адрес. Скрипт IDC (RebuildImports.idc), основанный на данном подходе, доступен на сайте книги. RebuildImports.idc не имеет недостатка renimp.idc связанного с символами подчеркивания.

Эффект от переименования каждой записи в таблице импорта сохраняется вплоть до дизассемблирования, далее показан автоматически обновленный дизассемблерный листинг:

```
UPX0:00403C5B call    ds: RtlFreeHeap
UPX0:00403C61 test    eax, eax
UPX0:00403C63 jnz     short loc_403C7B
UPX0:00403C65 call    sub_40230F
UPX0:00403C6A mov     esi, eax
UPX0:00403C6C call    ds:RtlGetLastWin32Error
```

Имя каждой переименованной записи таблицы импорта записывается во все места, откуда вызываются импортированные функции, что делает дизассемблерный код более читабельным. После того как таблица импорта была приведена в необходимый вид, можно сделать снимок памяти, и вернуть все изменения в базу данных.

6.3.3. Прячем отладчик

Популярным методом защиты от отладчика является определения присутствия его в системе. Авторы инструментов обфускации прекрасно понимают, что при помощи отладчика вы

можете разрушить всю сделанную ими работу. В ответ они часто принимают меры, предотвращающие запуск их инструментов, в случае обнаружения присутствия отладчика. Мы упомянули статью Николаса Фальере (Nicolas Falliere) “Антиотладка в Windows” (Windows Anti-Debug Reference), которая содержит прекрасное описание нескольких специфичных для Windows методов определения присутствия отладчика. Вы можете познакомиться с некоторыми из этих методов при помощи простого скрипта IDC для запуска сессии отладки, и автоматической конфигурации нескольких точек остановки.

Для того, чтобы запустить сессию отладки из IDC, следует начать скрипт с двух следующих строк:

```
RunTo (BeginEA ());
GetDebuggerEvent (WFNE_SUSP, -1);
```

Данные операторы запускают отладчик, требуют остановку по адресу точки входа, и затем ожидают пока операция завершиться (коротко говоря, нам необходимо проверить значение, возвращаемое GeTDebuggerEvent). После того как наш скрипт снова получит контроль у нас будет активная сессия отладки, а процесс, который мы хотим отлаживать, отображен в памяти вместе со всеми библиотеками, от которых он зависит.

Мы будем обходить первое обнаружение отладчика, совершаемое при помощи поля IsDebugged расположенного в блоке среды окружения процесса (process environment block, PEB). Это поле длиной 1 байт, которое принимает значение единицы, в случае если процесс подвергается отладке, и 0 если нет. Поле расположено после первых двух байт PEB, поэтому все, что нам нужно, это найти PEB и пропатчить соответствующий байт, чтобы установить нулевое значение. По счастливому совпадению, именно это поле также проверяется функцией Windows API IsDebuggerPresent, поэтому мы убьем двух зайцев за раз. Обнаружить PEB проще простого, так как регистр EBX содержит указатель на него при входе в процесс. Следующая часть скрипта выполняет нужный нам патч:

```
PatchByte(EBX + 2, 0);   //Set PEB.IsDebuggerPresent to zero
```

Другим методом, упомянутым в статье Фальере, является проверка нескольких бит в другом поле PEB называемом NtGlobalFlags. Эти биты относятся к работе кучи процесса и устанавливаются в единицу при отладке процесса. Следующий код IDC

получает поле NtGlobalFlags из PEB, сбрасывает необходимые биты и затем сохраняет обратно в PEB:

```
GlobalFlags = Dword (EBX + 0x68) & ~0x70;      //read and mask PEB.NtGlobalFlags
PatchDword (EBX + 0x68, globalFlags);      //patch PEB.NtGlobalFlags
```

Несколько методов в статье Фальери полагаются на то что информация, возвращаемая некоторыми системными функциями при отладке отличается от возвращаемой ими обычно. Первой такой функцией является NtQueryInformationProcess из ntdll.dll. При помощи подобной функции процесс может запросить информацию оего ProcessDebugPort. Если производится отладка этого процесса, то результат будет отличен от нуля, иначе ноль. Одним из путей обхода этого метода является установка точки остановки на выход из функции NtQueryInformationProcess. Для того чтобы автоматически обнаруживать эту инструкцию, нам необходимо предпринять следующие шаги:

- Посмотреть адрес функции NtQueryInformationProcess .
- Запросить IDA создать функцию по этому адресу. Заметим, что обычно функция не дизассемблируется, так как ее вызовы могут отсутствовать.
- После того как IDA создаст функцию, необходимо запросить ее атрибуты, для того чтобы определить ее конечный адрес.
- Вычесть 3 из конечного адреса для того, чтобы получить начало инструкции возвращения, и установить по этому адресу точку остановки. Мы вычитаем 3, потому что функция завершается инструкцией ret 14h длиной 3 байта, а не однобайтовой ret.
- Изменить атрибуты точки остановки для предотвращения остановки выполнения в точке остановки, и добавить в условия, чтобы запускалась некоторая функция каждый раз, когда встречается точка остановки.

Первый шаг может оказаться довольно запутанным. Во-первых, нужнопомнить, что необходимая нам функция будет называться в отладчике ntdll_NtQueryInformationProcess . Во-вторых, в IDA есть ошибка, которая мешает нам автоматически искать это символьное имя. Хоть эту ошибку и обещали исправить в будущей версии IDA, но нам необходимо работать уже сейчас. Проблема заключается в том, что это функция также обладает и именем ZwQueryInformationProcess (ntdll_ZwQueryInformationProcess в

отладчике). IDA применяет оба имени к адресу данной функции, но только имя ZwQueryInformationProcess может быть обнаружено. С учетом данной ошибки, все эти шаги можно реализовать так:

```
func = LocByName("ntdll_ZwQueryInformationProcess");
MakeFunction(func, BADADDR);                //can't find end until a function exists
end = GetFunctionAttr(func, FUNCATTR_END) - 3;  //compute address of ret
AddBpt(end);
SetBptAttr(end, BPT_BRK, 0);  //don't stop
SetBptCnd(end, "bpt_NtQueryInformationProcess()"); //run this function on break
```

Нам остается только реализовать функцию для точки остановки, которая будет держать отладчик спрятанным от обнаружения. Прототипфункции NtQueryInformationProcess показандалее:

```
 NTSTATUS WINAPI NtQueryInformationProcess(
   __in       HANDLE ProcessHandle,
   __in       PROCESSINFOCLASS ProcessInformationClass,
   __out      PVOID ProcessInformation,
   __in       ULONG ProcessInformationLength,
   __out_opt  PULONG ReturnLength

);
```

Информация о процессе запрашивается путем предоставления целочисленного идентификатора запроса в параметре ProcessInformationClass. Информация возвращается через пользовательский буфер, указанный через параметр ProcessInformation. Также можно передать константу ProcessDebugPort (значение 7) для того, чтобы запросить статус отладки данного процесса. Если процесс отлаживается в отладчике пользовательского пространства, то возвращаемое значение будет отличным от нуля. Если процесс не отлаживается, то возвращаемое значение будет равно нулю. Функция точки остановки, которая всегда устанавливает возвращаемое значение, ProcessDebugPort в ноль, показана далее:

```
#define ProcessDebugPort 7
static bpt_NtQueryInformationProcess() {
auto p_ret;
if (Dword(ESP + 8) == ProcessDebugPort) {//test ProcessInformationClass
      p_ret = Dword(ESP + 12);  // ProcessInformation, this is a pointer
```

```
if (p_ret) {
PatchDword(p_ret, 0);  //fake no debugger present
}
   }
 }
```

Напоминаем, что эта функция запускается каждый раз, когда выполнение доходит до инструкции ret функции NtQueryInformationProcess. В текущий момент указатель стека указывает на адрес возврата функции, который расположен наверху пяти аргументов NtQueryInformationProcess. Функция точки остановки начинается с того, что проверяет значение ProcessInformationClass для того, чтобы определить запрашивается ли информация ProcessDebugPort. Если да, то функция продолжает выполняться и получает возвращаемое значение, проверяя что оно не является нулевым, далее сохраняет нулевое возвращаемое значение, для того чтобы скрыть отладчик.

Другая функция, которая была упомянута в статье Фальере, называется NtSetInformationThread (которая является оберткой для ZwSetInformationThread), и также находится в ntdll.dll. Прототип этой функции приведен далее:

```
NTSTATUS NtSetInformationThread(
  IN HANDLE  ThreadHandle,
  IN THREADINFOCLASS  ThreadInformationClass,
  IN PVOID  ThreadInformation,
  IN ULONG  ThreadInformationLength
);
```

АнтиотладочныйметодзаключаетсявпередачезначенияTHRead HideFromDebuggerвпараметреTHReadInformationClass, чтозаставляетпотокотсоединитьсяототладчика. Для того, чтобы обойти этот метод требуется примерно то же, что и в предущем примере, кроме того мы устанавливаем точку остановки на первую инструкцию функции NtSetInformationThread, так как она в отличие от NtQueryInformationProcess, которая выполняет простой запрос NtSetInformationThread, не должна выполняться вообще, иначе она может отсоединить поток от отладчика. Код для данного решения приведен далее:

```
func = LocByName("ntdll_ZwSetInformationThread");
AddBpt(func);                  //break at function entry
MakeFunction(func, BADADDR);
```

```
SetBptAttr(func, BPT_BRK, 0);  //don't stop
SetBptCnd(func, "bpt_NtSetInformationThread()");
```
Соответствующая функция для точки остановки приведена далее:
```
#define ThreadHideFromDebugger 0x11
static bpt_NtSetInformationThread() {
if (Dword(ESP + 8) == ThreadHideFromDebugger) {//test ThreadInformationClass
    EAX = 0;                          //STATUS_SUCCESS
    EIP = GetFunctionAttr(EIP, FUNCATTR_END) - 3; //jump to end of function

}
 }
```

В данном случае, функция срабатывает сразу после входа. Так как мы хотим предотвратить выполнение функции, чтобы помешать отсоединению от отладчика, то мы проверяем значение параметра THReadInformationClass, и обходим тело функции, если пользователь указал THReadHideFromDebugger. Обход тела функции выполняется при помощи установки необходимого возвращаемого значения и изменения указателя инструкций таким образом, чтобы он указывал на инструкцию выхода из функции.

Последняя функция, которую нам необходимо обсудить и которая также упоминалась в статье Фальери это OutputDebugStringA из kernel32.dll. Прототип данной функции показан далее:

```
void WINAPI OutputDebugStringA(
  __in_opt  LPCTSTR lpOutputString
);
```

В данном примере WINAPI является синонимом _stdcall и используется для того, чтобы указать метод вызовы функции, используемый OutputDebugStringA. Коротко говоря, у функции нет возвращаемого значения, на что указывает тип void в прототипе; однако, в соответствии со статьей эта функция возвращает 1, в случае если отладчик не присоединен к процессу, и также она «возвращает» адрес строки переданной в качестве параметра, в случае если отладчик присоединен. В нормальных условиях функции _stdcall возвращающие значение создаются при помощи регистра EAX. Так как в регистре EAX должно находится некоторое значение при возврате из OutputDebugStringA, то можно поспорить, что это значение является возвращаемым; но в любом случае, так как указан тип void, то нет никакой документации или гарантии о том, что должен содержать EAX

в данном случае. Этот метод антиотладки основан на наблюдении поведения функции. Решением в данной ситуации является установки 1 в EAX каждый раз при возврате OutputDebugStringA . Следующий код IDC реализует данное решение:

```
func = LocByName("kernel32_OutputDebugStringA");
MakeFunction(func, BADADDR);
end = GetFunctionAttr(func, FUNCATTR_END) - 3;
AddBpt(end);
SetBptAttr(end, BPT_BRK, 0);  //don't stop
 //fix the return value as expected in non-debugged processes
SetBptCnd(end, "EAX = 1");
```

В данном примере используется такой же метод для автоматического определения конца функции OutputDebugStringA, который использовался в предыдущих примерах. Однако, в отличие от предыдущего примера, в данном случае нам достаточно одного выражения IDC (а не специальной функции). В данной ситуации условие точки остановки изменяет содержимое EAX, чтобы убедиться, что в нем находится 1 при возврате функции.

Скрипт (HideDebugger.idc), который сочетает все элементы, приведенные в данном разделе, и является полезным инструментом для инициации сессии отладки и применении мер для борьбы с антиотладочными методами, доступен на веб сайте книги. Для получения большей информации по поводу того, как спрятать отладчик советуем посмотреть блог Ильфака, где он приводит различные методы.

6.3.4. Разбираемся с исключениями

Иногда программы ожидают, что они самостоятельно будут обрабатывать все исключения во время их выполнения. Обфусцированные программы могут пойти еще дальше, и намеренно генерировать исключения в качестве антиотладочного и антиконтрольного метода. К сожалению, исключения часто являются индикаторами проблемы, а назначение отладчика это помочь вам в решении проблем. Таким образом, отладчики обычно желают обрабатывать все исключения, которые происходят во время выполнения программы для того, чтобы помочь вам найти ошибки.

В случае, когда программа ожидает, что будет обрабатывать свои исключения самостоятельно, то нам необходимо предотвратить перехват исключений отладчиком или, как минимум, в случае

перехвата исключения, передать его обратно процессу. К счастью, у отладчика IDA есть возможность передавать отдельные исключения при их появлении или автоматически передавать все исключения указанного типа.

Автоматическая обработка исключений настраивается через команду меню Debugger ► Debugger Options; диалог показан на рис.6.7.

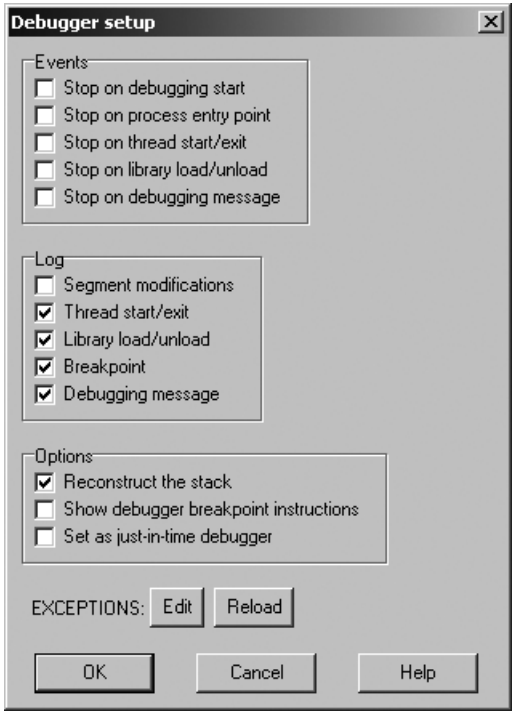


Рис. 6.7. Диалог настройки отладчика

В дополнение к нескольким событиям, которые могут быть настроены, чтобы автоматически останавливать отладчик, и некоторому количеству событий способных автоматически записываться в лог окна сообщений IDA - диалог настройки отладчика используется для того, чтобы настроить поведение отладчика при

перехвате исключений. Кнопка Edit открывает диалог конфигурации исключений, показанный на рис.6.8.

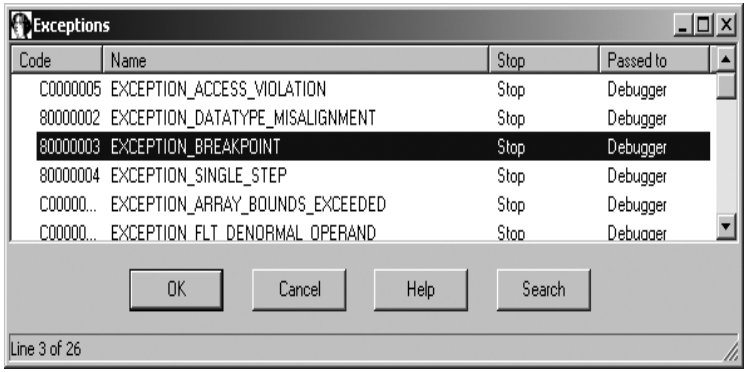


Рис.6.8. Диалог конфигурации исключений

Для каждого исключения, тип которого известен отладчику, в диалоге приводится список специфичного для операционной системы кода исключения, имя исключения, необходимо ли останавливать процесс (Stop/No), а также отладчик может перехватывать это исключение или автоматически передавать его приложению (Debugger/Application). Основной список всех исключений и настроек по умолчанию содержится в <IDADIR>/cfg/exceptions.cfg. Дополнительно конфигурационный файл содержит сообщения, которые необходимо отображать каждый раз, когда исключение данного типа происходит в ходе запуска процесса отладчиком. Изменение значений по умолчанию можно выполнить редактированием файла exceptions.cfg при помощи текстового редактора. В файле exceptions.cfg значения stop и nostop используются для того, чтобы указать, должен ли отладчик останавливать процесс или нет при подобом исключении.

Обработка исключений также может быть настроена отдельно для каждого процесса при помощи редактирования индивидуальных исключений через диалог Exceptions configuration. Для того чтобы изменить поведение отладчика для данного типа исключения, кликните правой кнопкой мыши по необходимому исключению в диалоге, и выберите Edit. На рис. 6.9. показан диалог обработки исключений.

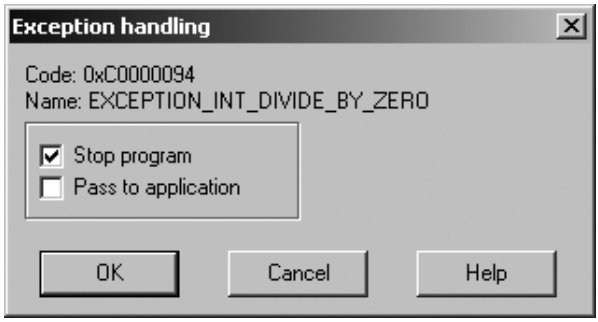


Рис.6.9. Диалог обработки исключений

Две опции, соответствующие двум настраиваемым опциям из exceptions.cfg, могут быть настроены для любого исключения. Во-первых, можно указать, должен ли отладчик останавливать процесс при встрече исключения указанного типа, или он должен обработать исключение и продолжить выполнение. Важно понимать, что отладчик может только определенным образом обрабатывать исключения, которые относятся к отладке, такие как точки остановки или одношаговые исключения. Для большинства других типов исключений при выборе их пропуска процесс, скорее всего, войдет в бесконечный цикл создания исключения, так как отладчик не знает, как его обрабатывать должным образом.

Вторая опция позволяет вам решать, должен ли данный тип исключения передаваться приложению, для того чтобы оно могло самостоятельно его обработать. В случае, если в приложении есть операции, которые полагаются на обработку исключений, то вам стоит выбрать данную опцию, чтобы передавать данный тип исключения приложению. Это может потребоваться при анализе обфусцированного кода, сгенерированного утилитой tElock (которая использует собственный обработчик исключений). Заметим, что включение опции Pass to application для определенного типа исключения переопределяет опцию Stop Program для этого типа исключения, в случае если это исключение было должным образом обработано приложением. Другими словами, если приложение обрабатывает должным образом исключение, переданное ему отладчиком, то отладчик не будет останавливать приложение независимо от того, была ли установлена опция Stop Program.

Недостатком текущей версии отладчика (5.2 на момент написания учебного пособия) является то, что он не предоставляет

никаких средств передачи отдельных исключений во время отладки. Для примера возьмем исключение целочисленного деления на ноль. При использовании опции Pass to the application, мы можем передавать все появления этого исключения приложению. Если установлена опция Stop program, то при каждом появлении этого исключения, программа будет останавливаться. Иного не дано. Когда мы выбираем Stop program в качестве основной стратегии для обработки исключений, то нет никакой возможности передать это исключение обратно программе, и продолжить выполнение. Такая возможность есть в OllyDbg и gdb, а Ильфак сказал, что эта опция может быть включена в следующих версиях IDA. Эта возможность крайне полезна для трассировки распаковщиков, которые намеренно генерируют исключения в качестве метода препятствия реверс-инжинирингу. В таких случаях, нежелательно давать распаковщику полную свободу действий при помощи опции Pass to application, но мы могли бы передавать исключение приложению после его остановки, для того чтобы понаблюдать как оно его обрабатывает.

Для того чтобы определить как приложение будет обрабатывать исключение, необходимо знать путь выполнения трассировки обработчиков исключений, что в свою очередь, требует знания об обнаружении обработчиков исключений. Ильфак обсуждает трассировку обработчиков Windows SEH в своем блоге в посте с названием "Tracing exception handlers.". Основная идея заключается в том, чтобы обнаружить интересующие нас обработчики исключений, путем просмотра списка установленных обработчиков приложения. При работе с исключениями Windows SEH, указатель на начало этого списка может быть обнаружен в первом dword, в блоке среды окружения потока (thread environment block, TEB). Список обработчиков исключений представляет собой стандартную структуру данных в виде связного списка, который содержит указатель о следующем обработчике исключений в цепочке, и указывает на функцию, которую необходимо вызвать для обработки всех генерируемых исключений. Исключения передаются вниз по списку от одного обработчика к другому до тех пор, пока обработчик не согласится обрабатывать данное исключение, и не уведомит операционную систему, что процесс может продолжить нормальное выполнение. Если ни один обработчик не станет обрабатывать текущее исключение, то ОС завершит процесс или, в случае если процесс отлаживается, сообщит отладчику, что в процессе произошло исключение.

При отладке в IDA, TEB отображаются в базу данных в секцию с именем TIB[NNNNNNNN], где NNNNNNNN это

восьмизначное шестнадцатеричное представление идентификационного номера потока. В следующем листинге показывается пример первого dword в такой секции:

```
TIB[000009E0]:7FFDF000 TIB_000009E0_ segment byte public 'DATA' use32
TIB[000009E0]:7FFDF000 assume cs:TIB_000009E0_
TIB[000009E0]:7FFDF000 ;org 7FFDF000h
TIB[000009E0]:7FFDF000 dd offset dword_22FFE0
```

Первые три строки отображают суммарную информацию о сегменте, а в четвертой содержатся первые dword секции, указывающие, что первый обработчик исключений, может быть найден по адресу 22FFE0h (offset dword_22FFE0). Если для данного потока не было назначено обработчиков исключений, то первое dword будет содержать значение 0FFFFFFFFh, указывающее о том, что был достигнут конец цепочки обработчиков. В данном примере отображены два dwords по адресу 22FFE0h:

```
Stack[000009E0]:0022FFE0 dword_22FFE0 dd 0FFFFFFFFh ;
DATAXREF: TIB[000009E0]:7FFDF000o
Stack[000009E0]:0022FFE4 dd offset loc_7C839AA8
```

Первое dword содержит значение 0FFFFFFFFh, указывает, что это последняя запись обработчка в цепи. Второе dword содержит адрес 7C839AA8h (offset loc_7C839AA8), указывающий, что функция по данному адресу должна вызываться для обработки всех возникающих во время выполнения исключений. Если нам необходимо проследить, как обрабатываются исключения, то стоит установить точку остановки по этому адресу.

Так как просмотреть цепочку SEH довольно просто, то было бы неплохо иметь в отладчике возможность просмотреть ее. В случае наличия такой возможности,мы довольно просто перешли бы к любому из обработчиков, и расставили точки остановки перед ними. К сожалению, это еще одна функция, которая доступна в OllyDbg, но недоступна в отладчике IDA. Для того, чтобы устранить этот недостаток, мы разработали дополнение для IDA, которое при вызове из отладчика отобразит список всех обработчиков исключений заданных для данного потока. Пример показан на рис.6.10.

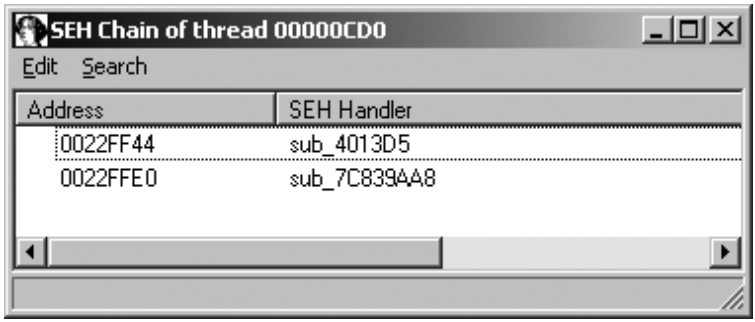


Рис.6.10. Дополнение SEH Chain

Дополнение использует функцию SDK choose2, для отображения диалога со списком текущей цепи обработчиков. Для каждого установленного обработчика исключений отображаются адрес записи и адрес соответствующего обработчика. При помощи двойного клика по записи, выполняется переход в дизассемблерный режим (IDA View-EIP или IDA View-ESP) по адресу обработчика. Назначением данного выполнения, является упрощение процесса поиска обработчиков исключений. Исходный код доступен на сайте книги.

На обратной стороне процесса обработки исключений находится способ, следуя которому обработчик исключений возвращает контроль (если решает сделать это) приложению при встрече исключения. Когда обработчик исключений вызывается операционной системой, ему предоставляется доступ к содержимому всех регистров процессора, актуальному на момент появления исключения. Во время процесса обработки исключений, функция может изменить значения регистров до возврата контроля приложению. Это сделано для того, чтобы обработчик исключений мог исправить ситуацию, чтобы программа могло продолжить нормальное выполнение. Если обработчик определит, что процесс может продолжить выполнение, то он сообщает об этом ОС и значения регистров восстанавливаются. Некоторые утилиты препятствуют реверс-инжинирингу при помощи изменения хода процесса, путем модификации сохраненного значения указателя инструкций во время фазы обработки исключений. Когда ОС возвращает контроль процессу, то выполнение продолжается с адреса, указанного в измененном указателе инструкций.

В своем блоге в посте по поводу отслеживания исключений, Ильфак обсуждает тот факт, что обработчики исключений Windows SEH возвращают контроль процессу при помощи функции NtContinue (также известной как ZwContinue) из ntdll.dll. У NtContinue есть доступ к сохраненным значениям всех регистров, и можно точно определить, откуда процесс продолжит выполнение. Это осуществляетсяпутем изучения значения, содержащегося в указателе инструкций изнутри NtContinue. После того, как мы узнаем, откуда процесс начнет выполнение, мы сможем установить точку остановки, чтобы избежать вхождения в код операционной системы,и остановить процесс как можно раньше до начала выполнения. Следующие шаги отображают процесс, который необходимо выполнить:

- Найдите NtContinue и установите точку остановки на ее первой инструкции с флагом «не останавливать процесс».
- Добавьте условие к этой точке остановки.
- При срабатывании точки остановки, получите адрес сохраненных значений регистров, путем чтения указателя CONTEXT из стека.
- Извлеките сохраненный указатель инструкций из записи CONTEXT.
- Установите точку остановки по извлеченному адресу, и продолжите выполнение процесса.

Используя скрипт, аналогичный скрипту прячущему отладчик, мы можем автоматизировать эти задачи и связать их с началом сессии отладки. Следующий код демонстрирует запуск процесса в отладчике, и установку точки остановки на NtContinue:

```
static main() {
auto func;
RunTo(BeginEA());
GetDebuggerEvent(WFNE_SUSP, -1);
//   func = LocByName("ntdll_NtContinue");
func = LocByName("ntdll_ZwContinue");
AddBpt(func);
SetBptAttr(func, BPT_BRK, 0);  //don't stop
SetBptCnd(func, "bpt_NtContinue()");}
```

Цель данного заключается в простой установке точки остановки на запись NtContinue. Поведение точки остановки определяется функцией IDC bpt_NtContinue, которая приведена далее:

```
static bpt_NtContinue() {
```

```
auto p_ctx, next_eip;
   p_ctx = Dword(ESP + 4);          //get CONTEXT pointer argument
   next_eip = Dword(p_ctx + 0xB8);   //retrieve eip from CONTEXT
AddBpt(next_eip);                   //set a breakpoint at the new eip
SetBptCnd(next_eip, "Warning(\"Exception return hit\") || 1");
}
```

Функция - определяет указатель на сохраненную информацию о регистрах, получает сохраненное значение указателя инструкций по смещению 0xB8, внутри структуры CONTEXT, и устанавливает точку остановки по этому адресу. Для того чтобы пользователь знал почему произошла остановка процесса, отображается сообщение.

Вы решили сделать так, потому что точка остановки не была установлена напрямую пользователем, поэтому он может не сопоставить событие с возвратом от обработчика исключений.

Данный пример отражает простое средство обработки возвратов исключений. Более сложные решения, могут быть реализованы при помощи функции bpt_NtContinue.Например, если вы подозреваете, что обработчик исключений изменяет содержимое регистров, и мешает устанавливать аппаратные точки остановки, то вы можете восстановить значения регистров отладки для того, чтобы вернуть контроль над отлаживаемым процессом.

ЗАКЛЮЧЕНИЕ

Дизассемблеры и отладчики могут использоваться не только для нахождения ошибок в программном обеспечении, но также и как эффективный инструмент для реверс-инжиниринга. Вся сила IDA проявляется в интерактивном взаимодействии с пользователем. В начале исследования дизассемблер выполняет автоматический анализ программы, а затем пользователь с помощью интерактивных средств IDA начинает давать осмысленные имена, комментировать, создавать сложные структуры данных и другим образом добавлять информацию в листинг, генерируемый дизассемблером, пока не станет ясно, что именно и как делает исследуемая программа.

Дизассемблер имеет консольную и графическую версии. Поддерживает большое количество форматов исполняемых файлов. Одной из отличительных особенностей IDA Pro является возможность дизассемблирования байт-кода виртуальных машин Java и .NET. Также поддерживает макросы, плагины и скрипты, а последние версии содержат интегрированный отладчик.

Целью этого курса было представить тот минимум необходимых навыков, который понадобится вам для эффективной работы при дизассемблировании, отладке и создании собственных алгоритмов защиты приложений.

СПИСОК ЛИТЕРАТУРЫ

1. Касперски К., Рокко Е. Искусство дизассемблирования. – СПб: БХВ Петербург, 2008.
2. Касперски К. Техника отладки программ без исходных текстов – БХВ-Петербург, 2005.
3. Касперски К. Техника защиты компакт-дисков от копирования. – БХВ-Петербург, 2004.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого были определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

О кафедре

Кафедра ВТ СПбГУ ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России.

Первоначально кафедра называлась кафедрой математических и счетно-решающих приборов и устройств и занималась разработкой электромеханических вычислительных устройств и приборов управления. Свое нынешнее название кафедра получила в 1963 году.

Кафедра вычислительной техники является одной из крупнейших в университете, на которой работают высококвалифицированные специалисты, в том числе 8 профессоров и 15 доцентов, обучающие около 500 студентов и 30 аспирантов.

Кафедра имеет 4 компьютерных класса, объединяющих более 70 компьютеров в локальную вычислительную сеть кафедры и обеспечивающих доступ студентов ко всем информационным ресурсам кафедры и выход в Интернет. Кроме того, на кафедре имеются учебные и научно-исследовательские лаборатории по вычислительной технике, в которых работают студенты кафедры.

Чему мы учим

Традиционно на кафедре ВТ основной упор в подготовке специалистов делается на фундаментальную базовую подготовку в рамках общепрофессиональных и специальных дисциплин, охватывающих наиболее важные разделы вычислительной техники: функциональная схемотехника и микропроцессорная техника, алгоритмизация и программирование, информационные системы и базы данных, мультимедиа технологии, вычислительные сети и средства телекоммуникации, защита информации и информационная безопасность. В то же время, кафедра предоставляет студентам старших курсов возможность специализироваться в более узких профессиональных областях в соответствии с их интересами.

Специализации на выбор

Кафедра ВТ ИТМО предлагает в рамках инженерной и магистерской подготовки студентам на выбор по 3 специализации.

1. Специализация в области информационно-управляющих систем направлена на подготовку специалистов, умеющих проектировать и разрабатывать управляющие системы реального времени на основе средств микропроцессорной техники. При этом студентам, обучающимся по этой специализации, предоставляется уникальная возможность участвовать в конкретных разработках реального оборудования, изучая все этапы проектирования и производства, вплоть до получения конечного продукта. Дня этого на кафедре организована специальная учебно-производственная лаборатория, оснащенная самым современным оборудованием. Следует отметить, что в последнее время, в связи с подъемом отечественной промышленности, специалисты в области разработки и проектирования информационно-управляющих систем становятся все более востребованными, причем не только в России, но и за рубежом.
2. Кафедра вычислительной техники - одна из первых, начавшая в свое время подготовку специалистов в области открытых информационно-вычислительных систем. Сегодня студентам, специализирующимся в этой области, предоставляется уникальная возможность изучать и осваивать одно из самых мощных средств создания больших информационных систем - систему управления базами данных Oracle. При этом повышенные требования, предъявляемые к вычислительным ресурсам, с помощью которых реализуются базы данных в среде Oracle, удовлетворяются за счет организации на кафедре специализированного компьютерного класса, оснащенного мощными компьютерами фирмы SUN, связанными в локальную сеть кафедры. В то же время, студенты,

специализирующиеся в данной области, получают хорошую базовую подготовку в области информационных систем, что позволяет им по завершению обучения успешно разрабатывать базы данных и знаний не только в среде Oracle, но и на основе любых других систем управления базами данных.

3.И, конечно же, кафедра не могла остаться в стороне от бурного натиска вычислительных сетей и средств телекоммуникаций в сфере компьютерных технологий. Наличие высокопрофессиональных кадров в данной области и соответствующей технической базы на кафедре (две локальные вычислительные сети, объединяющие около 80 компьютеров и предоставляющие возможность работы в разных операционных средах - Windows, Unix, Solaris), позволило организовать подготовку специалистов по данному направлению, включая изучение вопросов компьютерной безопасности, администрирования, оптимизации и проектирования вычислительных сетей.

Оголюк Александр Александрович

Защита приложений от модификации. Дополнительные материалы

Учебное пособие

В авторской редакции
Редакционно-издательский отдел НИУ ИТМО
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ №
Тираж 50 экз.
Отпечатано на ризографе

Редакционно-издательский отдел
Санкт-Петербургского национального
исследовательского университета информационных
технологий, механики и оптики
197101, Санкт-Петербург, Кронверкский пр., 49