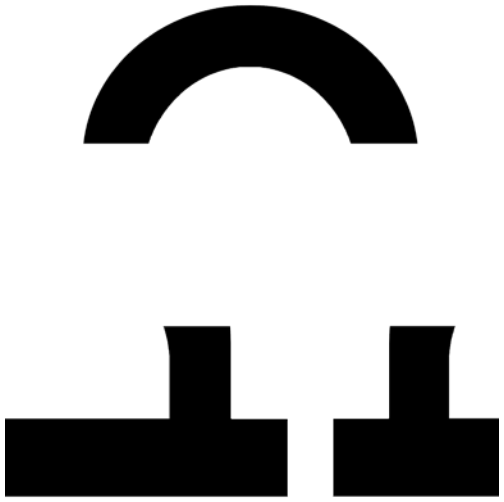
А.А. Оголюк

ЗАЩИТА ПРИЛОЖЕНИЙ ОТ МОДИФИКАЦИИ

Учебное пособие



Санкт-Петербург

2013

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

А.А. Оголюк

ЗАЩИТА ПРИЛОЖЕНИЙ ОТ МОДИФИКАЦИИ

Учебное пособие

Санкт-Петербург

2013

Оголюк А. А. Защита приложений от модификации: учебное пособие. – СПб: СПбГУ ИТМО, 2013. – 56 с.

Целью данного учебного пособия является ознакомление студентов с основами защиты приложений от модификации. В пособии приведены начальные сведения о дизассемблировании, а также показаны различные подходы к изучению и отладке приложений, реконструкции алгоритмов, практические приемы работы с популярным инструментом дизассемблирования – IDA. Материал пособия разбит на три раздела. Каждый раздел содержит краткий теоретический материал и вопросы.

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

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого 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

В рамках предмета «Защита приложений от модификации» мы рассмотрим вопросы выбора и использования средств дизассемблирования, отладки и защиты приложений, познакомится с внутренним устройством и алгоритмам работы основных инструментов (ПО) дизассемблирования и отладки.

Учебное пособие содержит практический материал, позволяющий развить навыки работы со средствами и инструментами изучения и защиты приложений от модификации.

Показаны различные подходы к изучению и отладке приложений, реконструкции алгоритмов, практические приемы работы с популярным инструментом дизассемблирования – IDA. Полученные в ходе изучения данного курса знания позволят эффективно защищать программы от модификации и несанкционированного копирования, а также создавать более оптимизирование приложения.

РАЗДЕЛ 1. ВВЕДЕНИЕ В ДИЗАССЕМБЛИРОВАНИЕ

1. ВВЕДЕНИЕ В ДИЗАССЕМБЛИРОВАНИЕ

1.1. Базовые понятия дизассемблирования

Что такое дизассемблирование. Дизассемблирование – преобразование программы на машинном языке к ее ассемблерному представлению. Декомпиляция – получение кода языка высокого уровня из программы на машинном языке или ассемблере.

Декомпиляция – достаточно сложный процесс. Это обусловлено следующими причинами:

– **Процесс компиляции происходит с потерями**. В машинном языке нет имен переменных и функций, и тип данных может быть определен только по производимым над ними операциям. Наблюдая пересылку 32-х бит данных, требуется значительная работа, чтобы определить, являются ли эти данные целым числом, дробью или указателем.

– **Компиляция это операция типа множество-множество**. Компиляция и декомпиляция могут быть выполнены множеством способов. Поэтому результат декомпиляции может значительно отличаться от исходного кода.

– **Декомпиляторы в значительной степени зависимы от конкретного языка и библиотек**. Обрабатывая исполняемый файл, созданный компилятором Delphi, декомпилятором, разработанным для C, можно получить фантастический результат.

– **Необходимо точное дизассемблирование исполняемого файла.** Любая ошибка или упущение на фазе дизассемблирования практически наверняка размножатся в результирующем коде.

Прогресс средств декомпиляции происходит медленно, но верно. Наиболее сложный на сегодняшний день декомпилятор IDA, будет рассмотрен ниже.

Зачем нужно дизассемблирование. Цель инструментов дизассемблирования заключается в содействии исследованию функционирования программ, когда их исходные коды не доступны. Наиболее распространенные цели дизассемблирования:

– анализ вредоносного программного обеспечения;

– анализ уязвимостей программного обеспечения с закрытым исходным кодом;

– анализ совместимости программного обеспечения с закрытым исходным кодом;

– валидация компилятора;

– отображение команд программы в процессе отладки.

Анализ вредоносного программного обеспечения. Разработчики вредоносного программного обеспечения вряд ли предоставят исходный код своего детища. А без доступа к исходному коду, будет довольно сложно

определить поведение вредоносной программы. Существуют два основных метода исследования вредоносных программ - динамический и статический анализ. *Динамический анализ* заключается в исполнении программы в тщательно контролируемом окружении (песочнице) и записи каждого ее действия. *Статический анализ* основан на разборе исходного кода, который, в случае вредоносной программы, в основном состоит из дизассемблированных листингов.

Анализ уязвимостей. Для простоты, можно разделить процесс аудита безопасности на три стадии: поиск уязвимостей, анализ уязвимостей, и разработка эксплойта. Одни и те же шаги предпринимаются вне зависимости от того, имеется ли у вас исходный код; однако, уровень трудоемкости резко возрастает, когда в вашем распоряжении есть лишь исполняемый файл. Первый шаг - исследование потенциально уязвимых условий в программе. Это зачастую достигается использованием динамических техник, таких как фаззинг, однако также может быть реализовано (обычно со значительно большими усилиями) посредством *статического анализа*. Как только проблема обнаружена, требуется определить, является ли она уязвимостью и если да, то при каких условиях.

Листинг дизассемблирования помогает понять, каким образом компилятор расположил переменные в памяти. Например, может быть полезно узнать, что объявленный программистом 70-байтный массив символов при распределении памяти компилятором был округлен в сторону 80 байт. Листинги дизассемблирования также предоставляют единственный способ понять, объявлены ли переменные глобально или внутри функций. Понимание реального расположения переменных в памяти жизненно важно при разработке эксплойтов.

Анализ совместимости. Когда программы доступны только в виде исполняемых файлов, сторонним разработчикам крайне сложно обеспечить совместимость с ними своих программ, а также расширить их функциональность. Например, если производитель не предоставил драйвер для аппаратного устройства, то реверс инжиниринг – практически единственное средство для разработки альтернативных драйверов.

Валидация компилятора. Дизассемблирование может быть средством для проверки соответствия работы компилятора его спецификации. Также исследователя может заинтересовать наличие дополнительных возможностей, оптимизирующих результат компиляции. С точки зрения безопасности важно быть уверенным, что код, генерируемый компилятором, не содержит черных ходов.

Отладка. К сожалению, дизассемблеры, встроенные в отладчики, зачастую малоэффективны (OllyDbg ─ исключение). Они неспособны к серийному дизассемблированию и иногда отказываются дизассемблировать, не будучи в состоянии определить границы функции. Поэтому, для лучшего контроля над процессом отладки, лучше использовать отладчик в сочетании с хорошим дизассемблером.

Как дизассемблировать. Типичные задачи, с которыми сталкивается дизассемблер: взять 100 КБ из исходного файла, отделить код от данных, преобразовать код к языку ассемблера, и главное ничего не потерять. В этот список можно добавить дополнительные пожелания, например, определение границ функций, распознавание таблиц переходов, выделение локальных переменных... Это значительно усложнит его работу. Качество результирующих листингов дизассемблирования определяется свойствами алгоритмов, а также уместностью их применения в конкретной ситуации.

1.2. Базовый алгоритм дизассемблирования

Шаг 1. Первым шагом в процессе дизассемблирования является идентификация кодового сегмента. Так как команды обычно смешаны с данными, то дизассемблеру необходимо их разграничить.

Шаг 2. Получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле) и выполнить табличное преобразование двоичного кода операции в соответствую ему мнемонику языка ассемблера.

Шаг 3. Как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу. После этого необходимо выбрать одну из разновидностей синтаксиса языка ассемблера.

Шаг 4. Далее необходимо перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

1.3. Алгоритм линейной развертки

Алгоритм линейной развертки использует крайне прямолинейный подход при выборе очередной команды для дизассемблирования: где завершается код одной команды - начинается код новой. В результате, наиболее сложной задачей становится определение первой команды. Дизассемблирование начинается с первого байта в сегменте кода и последовательно продвигается, обрабатывая команды одну за другой, пока не будет достигнут конец сегмента. При этом не производится попыток понять логику передач управления в программе посредством распознавания команд перехода таких, например, как условия.

Главное преимущество алгоритма линейной развертки состоит в полном покрытии кодового сегмента. Одним из основных недостатков является невозможность распознать данные, если они совмещены с кодом. Это очевидно из листинга 1.1, демонстрирующего результат дизассемблирования функции при помощи данного алгоритма. Эта функция содержит конструкцию switch. Компилятор принял решение реализовать switch как таблицу переходов. Более того, компилятор предпочел разместить таблицу переходов внутри самой функции. Конструкция jump по адресу ❶ 401250 ссылается на таблицу адресов

начиная с ❷410257. К сожалению, дизассемблер рассматривает ее как набор команд ❷ и неверно генерирует соответствующее представление на языке ассемблера.

Листинг 1.1
Алгоритм линейной развертки

```
40123f:    55                      push   ebp
 401240:    8b ec                   mov    ebp,esp
 401242:    33 c0                   xor    eax,eax
 401244:    8b 55 08                mov    edx,DWORD PTR [ebp+8]
 401247:    83 fa 0c                cmp    edx,0xc
 40124a:    0f 87 90 00 00 00       ja     0x4012e0
❶  401250:    ff 24 95 57 12 40 00     jmp    DWORD PTR [edx*4+0x401257]
❷ 401257:    e0 12                   loopne 0x40126b
 401259:    40                      inc    eax
 40125a:    00 8b 12 40 00 90       add    BYTE PTR [ebx-0x6fffbfee],cl
 401260:    12 40 00                adc    al,BYTE PTR [eax]
 401263:    95                      xchg   ebp,eax
 401264:    12 40 00                adc    al,BYTE PTR [eax]
 401267:    9a 12 40 00 a2 12 40    call   0x4012:0xa2004012
 40126e:    00 aa 12 40 00 b2       add    BYTE PTR [edx-0x4dffbfee],ch
 401274:    12 40 00                adc    al,BYTE PTR [eax]
 401277:    ba 12 40 00 c2          mov    edx,0xc2004012
 40127c:    12 40 00                adc    al,BYTE PTR [eax]
 40127f:    ca 12 40                lret   0x4012
 401282:    00 d2                   add    dl,dl
 401284:    12 40 00                adc    al,BYTE PTR [eax]
 401287:    da 12                   ficom  DWORD PTR [edx]
 401289:    40                      inc    eax
 40128a:    00 8b 45 0c eb 50       add    BYTE PTR [ebx+0x50eb0c45],cl
 401290:    8b 45 10                mov    eax,DWORD PTR [ebp+16]
401293:    eb 4b                   jmp    0x4012e0
```

1.4. Алгоритм рекурсивного спуска

Алгоритм рекурсивного спуска использует концепцию передачи управления, определяющую, должна ли команда быть дизассемблирована по наличию или отсутствию на нее ссылок от других команд. Для понимания алгоритма рекурсивного спуска, полезно классифицировать команды в зависимости от их влияния на счетчик команд CPU.

Команды, не влияющие на счетчик команд. После выполнения такой команды управление переходит непосредственно к следующей команде. Примерами такой команды может послужить арифметические, такие как `add`; Для подобных команд процесс дизассемблирования такой же как и при линейной развертке.

Команды условного перехода. Команды условного перехода, такие как x86 jnz, образуют две возможных ветви исполнения. Поскольку в статическом контексте обычно невозможно определить исход проверки условия, алгоритм рекурсивного спуска дизассемблирует обе ветви. Адрес целевой ветви добавляется в список адресов для последующего дизассемблирования. Дизассемблирование продолжается последовательно, так, как если бы условие было ложно.

Команды безусловного перехода. Безусловные переходы приводят к нарушению последовательного порядка исполнения команд. Команда,

получающая управление после выполнения безусловного перехода, может располагаться от него на значительном расстоянии. Кроме того, как видно из листинга 1.1, команды, следующие непосредственно за командой безусловного перехода не исполняются вообще. Таким образом, необходимость в их дизассемблировании отпадает.

Алгоритм рекурсивного спуска пытается определить адрес назначения безусловного перехода и занести его в список адресов для последующего дизассемблирования. К сожалению, не все безусловные переходы могут быть корректно обработаны данным алгоритмом. Когда адрес назначения перехода зависит от параметра, получаемого в процессе исполнения, его определение методами статического анализа становится невозможным.

Команды вызова функции. Команды вызова функции работают сходным образом с командами безусловных переходов (включая невозможность определить адрес назначения команды, такой как call eax), за исключением того, что после выполнения функции, управление обычно возвращается команде, следующей за ее вызовом. При этом, как и в случае с условными переходами, образуются две ветви исполнения. Адрес назначения команды call добавляется в список адресов для последующего дизассемблирования, в то время как команда, следующая за call, дизассемблируется с использованием алгоритма линейной развертки.

Алгоритм рекурсивного спуска может оказаться неэффективным в случае, если при возвращении из вызываемой функции поведение программы отклоняется от ожидаемого. Например, в коде функции может преднамеренно модифицироваться адрес возврата.

Команды возврата. В некоторых случаях, алгоритм рекурсивного спуска терпит неудачу. Команда возврата из функции (например, ret x86) не предоставляет информации о том, какая команда будет выполнена далее. Если бы программа была на самом деле запущена, управление было бы передано по адресу, расположенному на вершине стека. У дизассемблера нет возможности доступа к стеку. Вместо этого дизассемблирование внезапно останавливается. В этом случае алгоритм рекурсивного спуска обращается к списку отложенных адресов, и процесс дизассемблирования возобновляется. Этот рекурсивный процесс отражает смысл названия алгоритма.

Алгоритм рекурсивного спуска превосходно разделяет код и данные. Главным же недостатком такого подхода является неспособность распознавать ветви, образуемые таким командами как **jump** и **call**, использующими для адресации таблицы поиска. Несмотря на это, в сочетании с эвристиками для распознавания указателей на код, алгоритм рекурсивного спуска способен обеспечить хорошее покрытие кода в сочетании с прекрасным разделением кода и данных.

2. ИНСТРУМЕНТЫ ДИЗАССЕМБЛИРОВАНИЯ И РЕВЕРС ИНЖИНИРИНГА

2.1. Инструменты классификации

Впервые увидев незнакомый файл, часто полезно спросить себя: "Что это такое?". При этом первое правило - никогда не доверять расширению файла.

file - стандартная утилита, имеющаяся в большинстве UNIX-подобных операционных систем, а также в среде Cygwin для Windows. file пытается определить тип файла по характерным для него особенностям.

PE Tools - набор инструментов для систем Windows, полезный как для анализа запущенных процессов, так и исполняемых файлов.

Из списка процессов возможно снять дамп памяти определенного процесса в файл, использовать **PE Sniffer** для определения компилятора, использованного для его создания, а также узнать, был ли он обработан одной из известных утилит обфускации. Пользователи могут просматривать и изменять заголовки PE-файлов, используя утилиту **PE Editor**. Модификация заголовков PE-файлов часто необходима при попытке реконструировать исходный PE из его обфусцированной версии.

Обфускация - это попытка замаскировать поведение содержащейся в файлах программы. Программист может использовать обфускацию по многим причинам, наиболее распространенными из которых являются защита проприетарных алгоритмов и сокрытие вредоносного кода..

PEiD еще один инструмент для Windows, определяющий компилятор, использованный при создании PE-файла, а также программы, примененные для его обфускации.

2.2. Инструменты анализа

При необходимости реверс инжиниринга бинарных файлов, нам потребуются более сложные инструменты, детализирующие информацию о файле, полученную на этапе классификации.

nm. Целью утилиты nm является "извлечение таблицы имен из объектных файлов". Если использовать nm для анализа объектного файла (.o файл в отличие от исполняемого), то результатом работы программы станут имена всех функций и глобальных переменных, объявленных в нем.

Для отображения типа имен используются буквы:

U - неопределенное имя, обычно внешняя символьная ссылка.

T - имя, определенное в тексте программы, обычно имя функции.

T - локальное имя, объявленное в тексте программы. В программе на C обычно соответствует статической функции.

D - инициализированная переменная.

C - неинициализированная переменная.

Заглавные буквы обозначают глобальные имена, в то время как строчные - локальные. Полный список значений букв можно найти на страницах документации man для **nm.**

ldd. При создании исполняемого файла, все ссылки программы на библиотечные функций должны быть разрешены. У линкера есть два способа разрешения вызовов функций: *статическое* и *динамическое связывание.*

В случае статического связывания, линкер объединяет объектные файлы приложения с копией запрашиваемой библиотеки. Преимущества статического связывания заключаются (1) в более быстром осуществлении вызовов функций, и (2) более простом распространении приложений. Недостатки статического связывания заключаются в (1) большем размере результирующей программы и (2) большей трудоемкости ее обновления при изменении компонент библиотеки. Статическое связывание также усложняет процесс реверс инжиниринга.

Динамическое связывание отличается от статического тем, что линкер добавляет ссылки на запрашиваемые библиотеки в результирующий исполняемый файл. Это обычно положительно сказывается на его размере. Кроме того, обновление библиотек также значительно упрощается. Один из недостатков динамического связывания состоит в усложнении процесса загрузки. Еще одним недостатком динамического связывания, является необходимость для разработчиков распространять не только их приложение, но и все необходимые для его работы библиотеки.

Утилита **ldd** доступна для систем Linux и BSD. В системах OS X подобную информацию можно получить при помощи утилиты otool с флагом -L: otool -L*filename*. В системах Windows для отображения списка зависимых библиотек может быть использована утилита dumpbin (dumpbin /dependents *filename*), входящая в состав Visual Studio.

Objdump. Назначение objdump - "отображать информацию из объектных файлов." Для решения этой задачи objdump предоставляет огромное количество (30+) опций командной строки. Программа objdump может быть использована для отображения следующих типов данных (и еще многих), содержащихся в объектных файлах:

- **Заголовки сегментов.** Сводная информация по каждому сегменту программы.
- **Скрытые заголовки.** Информация о выделении памяти программе и другая необходимая загрузчику информация.
- **Отладочная информация.** Извлекает всю отладочную информацию, расположенную в файле программы.
- **Символьная информация.** Дамп таблицы имен, в виде сходном с результатом **nm.**

objdump реализует алгоритм линейной развертки для сегментов, помеченных как код.

objdump входит в состав набора инструментов **GNU binutils** и содержится в Linux, FreeBSD, и Windows (посредством Cygwin). Для работы с объектными файлами **objdump** использует библиотеку **libbfd**, компонент **binutils**, и таким образом способна анализировать файловые форматы, поддерживаемые этой библиотекой (среди прочих ELF и PE). Для анализа ELF-

файлов, также существует утилита **readelf**. **readelf** предлагает в основном те же возможности, что и **objdump**. Основное ее отличие заключается в том, что **readelf** не использует библиотеку **libbfd**.

otool полезна для анализа информации об исполняемых файлах OS X Mach-O. **otool** может пригодиться для отображения информации относительно заголовков файлов, таблиц имен, а также дизассемблирования их кодовых сегментов.

dumpbin утилита командной строки входящая в состав Microsoft Visual Studio. Подобно **otool** и **objdump**, **dumpbin** способна отображать различную информацию относительно PE-файлов Windows.

Кроме того **dumpbin** умеет извлекать различные секции PE-файлов включая имена, названия экспортируемых и импортируемых функций, а также дизассемблированный код.

c++filt. В объектном файле не может содержаться двух функций с одинаковым именем. Для осуществления перегрузки функций компилятор генерирует для них новые имена, используя информацию о типе их аргументов. Процесс создания уникальных имен для перегружаемых функций носит название *искажение имен*.

Стандарт C++ не регламентирует схему искажения имен, оставляя разработчикам компилятора создание их собственной. Длявосстановления искаженных имен и разработана утилита **c++filt**. **c++filt** рассматривает каждое входное слово так, как будто это искаженное имя, и далее пытается восстановить настоящее имя функции. Когда **c++filt** терпит неудачу, она попросту выводит слово без изменений.

Важно отметить, что искаженные имена содержат дополнительную информацию о функциях, которую **nm** обычно не предоставляет. Эта информация может быть крайне полезна в случае реверс инжиниринга. В ней также могут содержаться данные относительно имен классов и соглашения о связях функций.

2.3. Инструменты глубокого исследования

В этом параграфе мы обсудим инструменты, разработанные для извлечения данных независимо от типа исходных файлов.

Утилита **strings** разработана для извлечения строк из файлов, безотносительно формата этих файлов.

Наряду со строками, которые могут быть выведены программой, мы также видим строки, представляющие собой имена функций или библиотек.

Несколько последних замечаний относительно использования **strings**:

– Используя **strings** для анализа исполняемых файлов, важно помнить, что по умолчанию будут исследованы только загружаемые сегменты файла.

– **strings** не показывает где именно, внутри файла была обнаружена строка.

– Многие файлы используют различные кодировки.

Дизассемблеры. Однажды вы столкнетесь с бинарными файлами, которые не соответствуют широко распространенным стандартам. В таком случае вам понадобятся инструменты способные начать процесс дизассемблирования со смешения указанного пользователем.

Примерами таких *потоковых дизассемблеров* для системы команд x86 являются **ndisasm** и **diStorm**. **ndisasm** утилита, идущая в комплекте с Netwide Assembler (NASM).

Во многих случаях полезна гибкость потокового дизассемблирования. Например, при атаке на компьютерные сети, сетевые пакеты могут содержать шеллкод. Потоковые дизассемблеры могут быть использованы для дизассемблирования частей пакета, содержащих шеллкод, с целью последующего анализа их вредоносной содержимого.

Вопросы:

1. Для каких целей используется дизассемблирование?
2. Назовите и охарактеризуйте алгоритмы дизассемблирования.
3. Перечислите и опишите инструменты дизассемблирования.

РАЗДЕЛ 2. ОСНОВЫ ИСПОЛЬЗОВАНИЯ IDA

3. НАЧАЛО РАБОТЫ С IDA

3.1. Запуск IDA

В этом разделе мы начнем с описания опций, которые предоставляет IDA при запуске, а затем объясним, что именно происходит, когда вы открываете бинарный файл для анализа. Наконец, мы сделаем краткий обзор пользовательского интерфейса.

Для единообразия примеры будут представлены с GUI Windows, за исключением случаев, когда разговор идет о специфической версии IDA (например, в примере отладки под Linux).

Обратите внимание: При установке в Windows IDA создает ключ реестра HKEY_CURRENT_USER\Software\Hex-Rays\IDAC:\launching_ida.html - CHP-4-FNOTE-1. Многие опции, которые можно изменять из самого IDA (не редактируя файлы конфигурации) хранятся в этом ключе. На других платформах IDA хранит подобные значения в бинарном файле данных ($HOME/.idapro/ida.cfd), который непросто редактировать.

При запуске IDA встретит вас загрузочным экраном, отображающим информацию о лицензии. После него отображается диалог, предлагающий три способа попасть в рабочее окружение:

New запускает помощника, который будет направлять вас при выборе файла для анализа. Вначале пользователь должен определить тип файла, который он хочет открыть. После того, как указан тип файла, для выбора файла используется стандартный диалог "Открыть файл". Наконец, отображается один или более дополнительных диалогов, которые позволяют вам указать специфические опции для анализа перед тем, как файл будет загружен, проанализирован и отображён.

Трудность в работе с помощником New File кроется в том, что пользователь должен точно знать тип файла, который необходимо открыть. Если тип файла неизестен, лучше использовать другой метод загрузки в IDA. Кнопка New соответствует команде File ► New.

Go прерывает процесс загрузки и открывает IDA с пустой рабочей областью. Команда File ► New запускает помощник New File Wizard. Команда File ► Open запускает диалог "открыть файл" без запуска помощника.

По умолчанию IDA применяет фильтр **known extensions**. Убедитесь, что в фильтре установлено нужное вам значение или он вообще отключен (необходимо выбрать "All Files". Когда вы открываете файл этим способом, IDA пытается автоматически определить тип загружаемого файла.

Previous позволяет открыть ранее используемые файлы. Список этих файлов наполняется значениями из записи **History** ключа реестра IDA.

Загрузка файла в IDA. Если вы решите открыть файл, используя команду File ► Open. IDA генерирует список потенциальных типов файлов и отображает этот список вверху диалога. Список показывает, какие загрузчики

IDA лучше всего подходят для работы с выбранным файлом. Этот список создается в результате запуска каждого файлового загрузчика в директории loaders. Если не знаете, какой загрузчик выбрать, неплохим решением будет остановиться на выбранном по умолчанию варианте.

Иногда Binary File будет единственной записью, доступной в списке загрузчиков. В таком случае подразумевается, что ни один из загрузчиков не распознал выбранный файл.

Выпадающее меню Processor Type позволяет вам указать, какой процессорный модуль (из директории *procs* IDA) надо использовать при дизассемблировании.

Поля **Loading Segment** и **Loading Offset** активны, только если выбраны выходной формат Binary File и процессор семейства x86. Так как двоичный загрузчик не может извлечь из файла какую-либо информацию о размещении памяти, значения сегмента и смещения используются для получения начального адреса загруженного содержимого. Если вы забудете указать начальный адрес во время загрузки, его в любое время можно изменить с помощью команды Edit ► Segments ► Rebase Program command.

Кнопки **Kernel Options** предоставляют доступ к конфигурации специфических опций дизассемблирования, которые IDA будет использовать для улучшения процесса рекурсивного погружения.

Кнопки **Processor Options** предоставляют доступ к опциям конфигурации выбранного процессорного модуля.

Оставшиеся опции используются для тонкой настройки процесса загрузки файов, каждая опциям подробно описана в файле помощи IDA.

Файлы баз данных IDA. Когда вы будете довольны опциями загрузки, то нажмите ОК, чтобы закрыть диалог (или Finish). После этого IDA загрузит выбранный исполняемый файл в память и проанализирует необходимые части этого файла. В результате будет создан файл базы данных IDA, компоненты которого хранятся в четырех файлах. Их имена совпадают с именем выбранного исполняемого файла, а разрешения становятся *.id0, .id1, .nam*, и *.til*. В файле *.id0* находится содержимое базы данных B-дерева, тогда как файл *.id1* содержит флаги, описывающие каждый байт программы. Файл *.nam* содержит индексную информацию, относящуюся к именованным областям программы, которые отображаются в окне имён IDA. Наконец, файл *.til* используется для хранения информации о локальных определениях типов, специфических для конкретной базы данных. Формат каждого из этих файлов является проприетарным форматом IDA, и их непросто редактировать без помощи инструментария IDA.

Для удобства эти четыре файла архивируются (опционально – со сжатием) в единственный **IDB**-файл каждый раз, когда вы закрываете текущий проект. Когда говорят о базе данных IDA, обычно имеют в виду IDB-файл. Если база данных закрыта корректно, вы не увидите файлов с расширениями *.id0, .id1, .nam*, or *.til* в ваших рабочих директориях.

Создание базы данных IDA. После того, как вы укажете файл для анализа и выставите нужные опции, IDA начнет создание базы данных.

Загрузчик IDA определяет структуру виртуальной памяти на основе информации, содержащейся в заголовке файла, и соответствующим образом конфигурирует базу данных.

После загрузчика работать начинает движок дизассемблирования IDA. Он передает выбранному процессорному модулю по одному адресу. Задача процессорного модуля – определить тип команды, расположенной по этому адресу, длину команды и место, с которого программа продолжит работу после выполнения инструкции. Когда IDA отыщет все команды в файле, он делает второй проход по списку адресов команд и просит процессорный модуль сгенерировать ассемблерную версию каждой команды.

После дизассемблирования IDA автоматически проводит дополнительный анализ двоичного файла, чтобы извлечь дополнительную информацию, полезную для аналитика. Информация типов, перечисленных ниже, может быть найдена и интегрирована в базу данных после начального анализа:

– **Идентификация компилятора** может помочь понять соглашения вызова функций, использованных в двоичном файле, а также определить, с какими библиотеками может быть связан файл.

– **Идентификация аргументов функций и локальных переменных.** Для каждой идентифицированной функции IDA проводит детальный анализ поведения регистра указателя стека чтобы определить доступ к переменным внутри стека и понять структуру стекового кадра.

– **Информация типов данных.** Используя знания о распространенных библиотечных функциях и требуемых ими параметрах, IDA добавляет в базу данных комментарии, указывающие, в каких местах параметр передается этим функциям.

Закрытие баз данных IDA. Если это первое сохранение новой базы данных, имя базы будет соответствовать имени входного файла, а расширение будет заменено на *.idb* (например, *example.exe* получит базу данных *example.idb*). Если у входного файла нет расширения, *.idb* все равно добавляется к имени (например, *httpd* становится *httpd.idb*). Доступные опции сохранения и их описания приведены ниже:

– **Don't pack database.** Эта опция просто сбрасывает изменения четырех компонентов базы данных и закрывает рабочее окружение, *не создавая* IDB-файл.

– **Pack database (Store).** Выбор этой опции архивирует четыре компонента базы данных в единый IDB-файл. Предыдущий IDB будет перезаписан без подтверждения.

– **Pack database (Deflate).** Эта опция аналогична предыдущей, за исключением того, что компоненты базы данных будут сжаты внутри IDB-архива.

– **Collect garbage.** Чтобы создать IDB-файл минимального размера, выберете эту опцию вместе с Deflate. Эту опцию обычно использует только в случае, если места на диске недостаточно.

– **DON'T SAVE the database.** Эта опция – единственный способ не сохранять изменения, сделанные в базе данных с момента последнего сохранения и наиболее близкий аналог отмены, доступный в IDA.

Повторное открытие баз данных. Нужно выбрать базу данных одним из методов открытия файлов IDA. Базы данных открываются гораздо быстрее во второй раз потому что никакого анализа выполнять не надо.

Любые сбои могут повредить открытую базу данных.

В таком случае у IDA нет возможности закрыть активную базу данных, и потому временные файлы базы данных не удаляются. IDB-файл соответствует последнему стабильному состоянию базы данных, а временные файлы содержат изменения, которые могли быть сделаны с момента последнего сохранения. Вариант «Choosing Continue with Unpacked Base» вовсе не гарантирует, что вы восстановите свою работу.

Другая ситуация, если базу данных вообще никогда не сохраняли, и от нее остались только временные файлы, с которыми велась работа на момент сбоя. В этом случае IDA предложит использовать опцию исправления сразу, как только вы откроете исполняемый файл.

3.2. Рабочая область IDA

Здесь присутствуют следующие области:

Область панели инструментов содержит инструменты, соответствующие наиболее часто используемым операциями IDA. Панели инструментов добавляются и удаляются из рабочей области с помощью команды View ► Toolbars command.

Цветная горизонтальная панель - это ***навигатор обзора,*** или просто *полоска навигации*. Навигационная полоска отображает линейный вид адресного пространства загруженного файла. Маленький *индикатор текущей позиции* (по умолчанию желтый) на навигационной полоске указывает на адресный диапазон, который отображается в данный момент в главном окне. Щелчок по навигационной полоске отображает соответствующую часть двоичного файла в окне дизассемблирования.

Окна данных содержат информацию, извлеченную из двоичного файла и отображающуюся в виде различных представлений базы данных. Большая часть вашего анализа скорее всего будет заключаться в работе с этими окнами. Дополнительные окна видов данных доступны через меню View ► Open Subviews, также это меню используется для восстановления нечаянно закрытых окон данных.

Окно дизассемблирования служит основным средством вывода данных. У него есть два режима вывода: граф (по умолчанию) и листинг. В режиме графа IDA отображает схематический граф одной функции. О*бзор графов* поможет понять процесс выполнения функции. Когда окно IDA-View активно, пробел переключает режимы между графом и листингом. Если вы хотите сделать режим листинга режимом по умолчанию, вы должны снять галочку с опции Use Graph View by Default на вкладке Graph в меню Options ► General.

Обзор графа, доступный только когда активен режим графа, предоставляет общий вид базовой структуры графа. Пунктирный прямоугольник показывает текущую область, видимую в окне.

Окно сообщений предоставляет всю информацию, генерируемую IDA.

Поведение рабочей области при начальном анализе. Во время начального автоанализа нового файла в главном окне Вы можете увидеть такие процессы, как:

– Сообщения о прогрессе появляются в окне сообщений.

– Для окна дизассемблирования генерируется вывод дизассемблирования.

– Наполняется окно строк, после чего в конце фазы анализа происходит последнее сканирование.

– Окно имён наполняется и периодически обновляется в процессе анализа.

– Навигационная полоска изменяется, так как распознаются новые области кода и данных, затем блоки кода распознаются как функции и, наконец, с использованием техник сопоставления шаблонов IDA функции распознаются как конкретный библиотечный код.

– Индикатор текущей позиции перемещается по навигационной полоске, показывая область, анализируемую в данный момент.

Два особенно полезных сообщения это **You may start to explore the input file right now** и **The initial autoanalysis has been finished**. Первое сообщение информирует вас о том, что IDA достаточно проанализировал файл для того, чтобы уже можно было изучать его с помощью различных окон данных. Если вы попытаетесь изменить базу данных до завершения этапа анализа, аналитический движок может вернуться и модифицировать ваши собственные изменения, или вы можете помешать ему выполнить анализ. Второе оповещает, что никаких автоматических изменений больше не будет. Теперь можно вносить в базу данных любые изменения.

Советы и хитрости рабочего окружения IDA. IDA предлагает огромный объем информации, и окно может стать сильно загромождено. Вот некоторые советы, которые помогут вам наиболее эффективно использовать рабочее окружение:

Чем больше экранного пространства вы выделите для IDA, тем приятней будет работать. Этим фактом вполне можно оправдать покупку гигантского монитора (или двух)!

Не забудьте, что команду View ► Open Subviews можно использовать для восстановления нечаянно закрытых окон данных.

Команда Windows ► Reset Desktop позволяет быстро вернуть рабочее окружение к виду по умолчанию.

Используйте команду Windows ► Save Desktop, чтобы сохранить текущий вид рабочего окружения. The Windows ► Load Desktop позволит вернуть сохраненный вид рабочего окружения.

Единственное окно, для которого можно менять шрифт, это окно дизассемблирования (и в режиме графов, и в режиме листинга). Шрифт задается командой Options ► Font.

3.3. Отчет об ошибке

Как и любая другая программа, IDA содержит некоторое количество ошибок. Для отправки отчета об ошибке доступно два метода. Вы можете связаться с поддержкой Hex-Rays по электронной почте support@hex-rays.com. Или разместить отчет об ошибке на форуме Hex-Rays. В любом случае вы должны удостовериться, что можете воспроизвести ошибку и быть готовым предоставить Hex-Rays копию базы данных, с которой возникает проблема. Если ошибка возникнет в установленном вами плагине, вам нужно будет связаться с автором плагина.

4. ОКНА ДАННЫХ IDA

Перед тем, как мы займемся основными окнами IDA, полезно обсудить несколько базовых понятий, касающихся пользовательского интерфейса IDA:

– **В IDA нет отмены.** Если с вашей базой данных произойдет что-то неожиданное, вам самостоятельно придется восстанавливать все до прежнего состояния.

– **Практически все действия имеют собственный пункт меню, горячую клавишу и кнопку на панели инструментов.** Помните, что как панель инструментов IDA, так и горячие клавиши можно сконфигурировать под свои нужды.

– **IDA предоставляет мощное контекстно-зависимое меню по правому щелчку мыши**. Хотя это меню и не отображает полный список доступных действий, оно напоминает о наиболее часто используемых командах, которые могут вам понадобиться.

4.1. Основные окна IDA

В конфигурации по умолчанию во время начального этапа загрузки и анализа IDA создает восемь (в версии 5.2) окон. Каждое из этих окон доступно через одну из вкладок, которые отображаются сразу под навигационной полоской. Три непосредственно видимых окна это окно обзора, окно имен и окно сообщений. Любые окна, о которых пойдет речь в этой главе, можно открыть через меню View ► Open Subviews.

Когда активно окно дизассемблирования, ESC работает аналогично кнопке «Назад». Любое другое окно ESC просто закрывает.

4.1.1. Окно дизассемблирования

Для окна дизассемблирования доступно два формата: текстовый листинг и графы, добавленные в версии 5.0. Режим графа всегда включается по умолчанию. Вы можете изменить его поведения с помощью вкладки Graph диалога Options ► General. Когда окно дизассемблирования активно, вы можете переключаться между графом и листингом в любое время по нажатию пробела.

4.1.1.1. Просмотр графа

Просмотр графа немного напоминает блок-схему программы тем, что функция разбита на базовые блоки, что позволяет визуализировать переход функции от одного блока к другому.

IDA использует стрелки разных цветов, чтобы различать течения различных типов между блоками функции. Базовые блоки, которые прерываются переходом по условию, генерируют два возможных течения в зависимости от проверяемого условия: *край «Да»* (есть переход по этой ветке) по умолчанию зеленый, *край «Нет»* - красный. Блоки без ветвления, имеющие всего один вариант продолжения, используют *нормальный край* (по умолчанию синий) для указания на следующий блок.

В режиме графа IDA отображает одну функцию за раз. Зажав CTRL и прокручивая колесико мыши, можно приближать и отдалять граф. То же самое можно выполнить, зажав CTRL и нажимая клавиши + и – на дополнительной клавиатуре. В случаях больших или сложных функции может пригодиться окно Общего вида графа. Оно всегда отображает структуру графа целиком, а пунктирная рамка показывает область, которая отображена в окне дизассемблирования.

Существует несколько способов изменить вид дисплея графов для своих нужд:

Panning. IDA прячет менее необходимую информацию о каждой ассемблерной строчке (например, информацию о виртуальном адресе) чтобы блок занимал меньше места на экране. Вы можете включить отображение дополнительной информации с каждой строчкой кода, выбирая нужные пункты в *disassembly line parts, доступном во кладке Disassembly* через меню Options ► General. Например, чтобы добавить виртуальные адреса к каждой строчке, мы включаем *line prefixes*.

Перемещение блоков. Отдельные блоки внутри графа можно перемещать на новые места, нажимая на панель с названием блока и перетаскивая в нужное место. Вы можете вручную перенаправить края, перетаскивая вершины в новые места. Новую вершину можно сделать, дважды кликнув по желаемому месту на крае, зажав клавишу SHIFT. Если вам понадобится вернуться к расположению графа по умолчанию, вы можете щелкнуть по графу правой кнопкой и выбрать Layout Graph.

Группировка и сворачивание блоков. Блоки можно группировать, по одному или вместе с другими блоками, и сворачивать, чтобы освобождать место на экране. Свернуть блок можно, нажав правой кнопкой по его панели с названием и выбрав Group Nodes.

Создание дополнительных окон дизассемблирования. Если вы захотите видеть графы двух разных функций одновременно, вам надо всего лишь открыть новое окно дизассемблирования, используя Views ► Open Subviews ► Disassembly. Первое окно дизассемблирования называется *IDA View-A*, последующие - *IDA View-B, IDA View-C* и так далее. Каждое окно дизассемблирование независимо от других.

4.1.1.2. Режим текста

Текстовое окно дизассемблирования – традиционный режим просмотра сгенерированного IDA кода. Текстовый дисплей показывает полный листинг программы (а не единственную функцию, как в режиме графов) и только он предоставляет средства для просмотра области данных двоичного файла. Вся информация, доступная в режиме графа, в том или ином виде доступна и в текстовом режиме.

Ассемблерный код представленный линейно, виртуальные адреса по умолчанию включены и обычно отображаются в формате [название раздела]:[виртуальный адрес], например, **text:0040110C0**.

Окно стрелок используется для отображения нелинейных переходов внутри функции. Обычные стрелки соответствуют безусловным переходам, а пунктирные – переходам по условию. Когда переход передает управление более раннему адресу программы, используется жирная линия. Обратное течение в программе указывает на наличие цикла.

Объявления отражают оценку стекового кадра, сделанную IDA. IDA определяет структуру стекового кадра функции, выполняя детальный анализ поведения указателя стека и всех указателей стековых кадров, использованных в функции.

Комментарии (точка с запятой начинает комментарий) - это *перекрёстные ссылки*.

Для примеров в книге мы чаще всего будем использовать текстовый режим.

4.1.2. Окно имён

Окно имён предоставляет общий список всех глобальных имен в двоичном файле. *Имя* – не более чем символическое описание, данное виртуальному адресу программы. IDA получает список имен с помощью символьной таблицы и анализа подписи во время начальной загрузки файла. Имена можно отсортировать по алфавиту или в порядке виртуальных адресов (по возрастанию и по убыванию). Двойной щелчок по любой записи окна имён отобразит в окне дизассемблирования выбранное имя.

Отображаемые имена кодируются цветом и буквами. Система кодирования описана ниже:

- **F** Обычная функция. Эти функции IDA не распознает как библиотечные.
- **L** Библиотечная функция. IDA распознает библиотечные функции с помощью алгоритма сопоставления подписей.
- **I** Импортированное имя, чаще всего имя функции из общей библиотеки.
- **C** Именованный код. Это именованные места программы, которые IDA не относит ни к каким функциям.

- **D** Данные. Именованные секции даты обычно представляют собой глобальные переменные.
- **A** Строковые данные ASCII. Это секция данных, содержащая четыре или более последовательных символа ASCII, завершаемых нулевым байтом.

В процессе дизассемблирования IDA генерирует имена для всех мест, на которые есть прямые ссылки как на код (цель ветвления или вызова) или как на данные (чтение, запись или взятие адреса). Если место именовано в символьной таблице имен, IDA заимствует имя из таблицы. Если для данного места записи в таблице нет, IDA генерирует имя по умолчанию, которое будет использоваться при дизассемблировании. Когда IDA решает назвать место, виртуальный адрес места совмещается с префиксом, указывающим на тип места. Ниже перечислены наиболее часто встречаемые префиксы, используемые при автоматической генерации:

- sub_***xxxxxx*** **-** Подпрограмма по адресу *xxxxxx*
- loc_***xxxxxx*** **-** Инструкция, расположенная по адресу *xxxxxx*
- byte_***xxxxxx*** **-** 8-битовые данные по адресу *xxxxxx*
- word_***xxxxxx*** **-** 16-битовые данные по адресу *xxxxxx*
- dword_***xxxxxx*** **-** 32-битовые данные по адресу *xxxxxx*
- unk_***xxxxxx***

Данные неизвестного размера по адресу *xxxxxx*.

4.1.3. Окно сообщений

Окно сообщений служит для IDA выходной консолью, там надо искать информацию о задачах, выполняемых IDA. Когда открывается новый двоичный файл, генерируются сообщения, сообщающие об этапе анализа, который на данный момент выполняет IDA, а также о действиях, выполняемых IDA для создания новой базы данных.

4.1.4. Окно строк

Окно строк – это встроенный в IDA эквивалент утилиты **strings** с некоторыми дополнениями. Начиная с версии 5.2, окно строк не открывается само, но доступно с помощью View ► Open Subviews ► Strings.

Окно строк отображает список строк, извлеченных из двоичного файла, а также адреса, где они находятся. При использовании перекрестных ссылок, окно строк позволяет быстро обнаружить интересующую вас строку и отследить ее к месту ссылки на нее программой. Сканирование на предмет содержания строк выполняется в соответствии с настройками окна строк, вы можете получить к ним доступ, нажав правой кнопкой в пределах окна строк и выбрав Setup. Окно Setup Strings используется для указания типов строк, которые будет искать IDA. По умолчанию IDA ищет написанные в стиле Си, прерываемые нулём, 7-битовые ASCII-строки, состоящие по меньшей мере из пяти символов.

Для поиска строки другого типа, вы должны перенастроить сканирование и выбрать соответствующий тип строк. Каждый раз, когда вы закрываете окно Setup Strings, нажимая OK, IDA повторно сканирует базу данных в соответствии с новыми настройками. Две опции заслуживают отдельного упоминания:

– **Отображать только определенные строки.** Эта опция заставляет окно строк отображать только именованные строковые данные, которые были автоматически созданы IDA или вручную – пользователем. С этой IDA не будет автоматически сканировать файл на предмет строк.

– **Игнорировать определения команд/данных.** Эта опция заставляет IDA искать строки в командах и существующих данных. Использование этой опции позволяет IDA, во-первых, видеть строки, встроенные в программную часть двоичного файла и по ошибке сконвертированные в команды и, во-вторых, распознавать строки внутри данных, отформатированные как что-то другое.

IDA показывает не все строки внутри двоичного файла, если настройка не сконфигурирована соответствующим образом.

4.2. Дополнительные окна IDA

Также IDA открывает несколько окон, свернутых в вашем рабочем окружении. Вкладки для быстрого доступа этих окон находятся под навигационной лентой. Эти окна предлагают альтернативные или специализированные представления базы данных.

Окно шестнадцатеричного просмотра. Окно шестнадцатеричного просмотра отображает стандартный шестнадцатеричный дамп содержимого программы и списков, по 16 байт на строку. ASCII-эквиваленты отображаются сбоку. Одновременно может быть открыто несколько окон шестнадцатеричного просмотра. Первое окно называется *Hex View-A*, второе - *Hex View-B*, следующее *Hex View-C* и так далее. По умолчанию первое окно синхронизируется с первым окном дизассемблирования. Когда окно дизассемблирования синхронизировано с окном шестнадцатеричным просмотра, прокрутка одного из окон заставляет другое прокручиваться до того же места (того же виртуального адреса). Если что-то выделено в окне дизассемблирования, соответствующие байты подсвечиваются в окне шестнадцатеричного просмотра. Отключение опции синхронизации позволяет прокручивать окна независимо друг от друга.

В некоторых случаях шестнадцатеричное окно не показывает ничего, кроме вопросительных знаков. Таким способом IDA сообщает вам, что он не знает, какие значения могут находиться в данной области виртуального адресного пространства.

Окно экспортов. Окно экспортов перечисляет входные точки файла. Они включают в себя входную точку исполнения программы, как указано в заголовочном разделе, а также все функции и переменные, которые файл экспортирует для использования другими файлами. Экспортированные записи отображают имя, виртуальный адрес и, если это возможно, порядковый номер. Для исполняемых файлов окно экспортов всегда содержит по меньшей мере

одну запись: входную точку исполнения программы. IDA называет эту точку start. Далее приведена типичная запись окна экспортов:

```
LoadLibraryA                    7C801D77 578
```

Двойной щелчок по записи в окне экспортов перемещает окно дизассемблирования к адресу, связанному с данной записью. Функции окна экспортов доступны через утилиты командной строки, такие как objdump (-T), readelf (-s) и dumpbin (/EXPORTS).

Окно импортов. Окно импортов перечисляет все функции, импортируемые анализируемым двоичным файлом. Окно импортов нужно, только в случае если двоичный файл использует общие библиотеки. Каждая запись в окне импортов включает в себя имя импортируемой функции или данных и имя библиотеки, их содержащей. Так как код импортируемой функции находится в общей библиотеке, адресный список для каждой записи относится к виртуальным адресам связанной с ним записи в таблице импорта. Пример записи таблицы импорта показан ниже:

```
0040E108  GetModuleHandleA        KERNEL32
```

Двойной щелчок по этой записи заставит окно дизассемблирования переместиться к адресу 0040E108.

Функции окна импортов также доступны через утилиты командной строки, такие как objdump (-T), readelf(-s), и dumpbin (/IMPORTS).

Окно функций. Окно функций перечисляет все функции, распознанные IDA в базе данных. Запись окна функций может выглядеть так:

```
malloc        .text        00BDC260 00000180 R . . . B . .
```

Эта строка указывает на то, что функция **malloc** находится в разделе `.text` по виртуальному адресу `00BDC260`, размером 384 байт (180 в шестнадцатеричной), возвращается к вызову (`R`) и использует регистр EBP (`B`) для ссылки на свои локальные переменные. Флаги, используемые для описания функции (такие как R и B в примере выше), расшифровываются во встроенной помощи IDA.

Как и с другими окнами, двойной щелчок по записи в окне функций заставляет окно дизассемблирования перейти к местоположению выбранной функции.

Окно структур. Окно структур используется для отображения распознанных IDA сложных структур данных, таких, как structs и unions в Си. На этапе анализа IDA сопоставляет используемую в программе память с библиотекой подписей.

Двойной щелчок по имени структуры данных заставляет IDA расширить структуру, что позволяет вам увидеть ее детальную схему, включая имена и размеры отдельных полей.

Две основные функции окна структур - отображение стандартных структур данных и предоставление средств для создания собственных структур

данных для использования в качестве шаблонов размещения памяти, если вы узнаете, что программа использует собственные структуры данных.

Окно перечислений. Если IDA обнаруживает использование стандартного перечисляемого типа данных, этот тип данных будет указан в окне перечислений. Окно перечислений позволяет определять собственные перечисляемые типы, которые вы сможете использовать с дизассемблируемыми двоичными файлами.

4.3. Вспомогательные экраны IDA

Каждое из этих окон доступно через меню View ► Open Subviews, они предоставляют информацию, которая нужна не всегда, поэтому изначально они отсутствуют на экране.

Окно сегментов. Окно сегментов отображает общий список сегментов, присутствующих в двоичном файле. Обратите внимание, что в терминах IDA *сегментами* обычно называют *секции*, на которое разбит двоичный файл. Информация, отображаемая в данном окне, включает в себя имя сегмента, начальный и конечный адреса и флаги разрешений. Начальный и конечный адреса представляют диапазон виртуального адресного пространства, который будет назначен данной части программы при выполнении. Ниже приведен пример содержимого окна сегментов:

Листинг 4.3.1
Содержимое окна сегмента

```
      Name   Start    End      R W X D L Align  Base Type   Class  AD es   ss   ds   fs
gs
      UPX0    00401000 00407000 R W X . L para    0001 public CODE    32 0000 0000 0001
FFFFFFFF F
      FFFFFFF
      UPX1    00407000 00408000 R W X . L para    0002 public CODE    32 0000 0000 0001
FFFFFFFF F
      FFFFFFF
      UPX2    00408000 0040803C R W . . L para    0003 public DATA    32 0000 0000 0001
FFFFFFFF F
      FFFFFFF
      .idata 0040803C 00408050 R W . . L para    0003 public XTRN    32 0000 0000 0001
FFFFFFFF F
      FFFFFFF
      UPX2    00408050 00409000 R W . . L para    0003 public DATA    32 0000 0000 0001
FFFFFFFF F
      FFFFFFF
```

В данном случае мы можем подозревать, что с данным двоичным файлом что-то не так, поскольку он использует нестандартные имена сегментов, и два его сегмента кода доступны для записи, что позволяет реализовать самомодифицирующий код. То, что IDA знает размер сегмента, не означает, что нам известно и его содержимое. По ряду причин сегменты часто занимают меньше места на диске, чем в памяти. В таких случаях IDA отображает значения для тех частей сегмента, которые IDA смог взять из файла на диске. Для остальных сегментов отображаются вопросительные знаки.

Двойной щелчок по любой записи в окне перемещает окно дизассемблирования к началу выбранного сегмента. Щелчок правой кнопкой мыши по записи выводит контекстное меню, которое позволяет добавлять новые сегменты, а также изменять и удалять существующие.

Функции окна сегментов, доступные из командной строки, включают в себя objdump (-h), readelf (-S), и dumpbin (/HEADERS).

Окно сигнатур. Сигнатуры используются для идентификации распространенных последовательностей запуска, сгенерированных компилятором, чтобы определить, каким компилятором был собран данный двоичный файл. Кроме того, сигнатуры используются для категоризации функций в качестве известных библиотечных функций, вставленных компилятором, или в качестве функций, добавленных в двоичный файл в результате статической линковки.

Окно сигнатур используется для перечисления распознанных IDA сигнатур. Пример из Windows PE показан ниже:

```
File      State     #func  Library name
vc32rtf   Applied   501    Microsoft VisualC 2-8/net runtime
```

Данный пример показывает, что IDA сопоставил с файлом сигнатуры **vc32rtf** (из *<IDADIR>/sigs*) и, сделав это, смог распознать 501 библиотечных функций.

Применение дополнительных сигнатур может быть весьма полезным. IDA может не распознать компилятор и не сможет выбрать нужные сигнатуры для сопоставления. В этом случае вам придется заставить IDA применить одну или более сигнатур самостоятельно. Вы можете создавать собственные сигнатуры для библиотек, для которых у IDA сигнатур нет. Чтобы применить новые сигнатуры, нажмите клавишу INSERT или щелкните правой кнопкой по окну сигнатур, и вам будет предложено применить новую сигнатуру из списка всех сигнатур, известных вашему IDA.

Окно библиотеки типов. Окно библиотек типов похоже на окно сигнатур. Библиотеки типов отражают знания IDA о стандартных типах данных и прототипах функций, используемых наиболее популярными компиляторами. Обрабатывая заголовочные файлы, IDA понимает, какие типы данных можно ожидать от библиотечных функций и делает соответствующие комментарии к коду. Также из заголовочных файлов IDA узнает о размере и положении сложных структур данных. Вся эта информация собрана в TIL-файлах (*<IDADIR>/til*) и. Чтобы загрузить нужный набор TIL-файлов, IDA должен вначале определить, какие библиотеки используются в программе. Вы можете заставить IDA загрузить дополнительные библиотеки типов, нажав INSERT или щелкнув правой кнопкой в окне библиотеки типов и выбрав Load Type Library.

Окно вызовов функций. Граф, отображающий отношения между вызывающими и вызываемыми функциями, называют *граф вызова функций* или *дерево вызова функций* . Иногда нам будет нужно увидеть не весь граф целой программы, а только ближайших соседей конкретной функции. Будем называть Y соседом X, если Y непосредственно вызывает X или X вызывает Y.

Когда вы открываете окно вызовов функций, IDA определяет соседей выбранной функции и генерирует окно.

Двойной щелчок по любой строчке окна вызовов функций перемещает окно дизассемблирования к выбранной функции (вызывающей или вызванной). Перекрестные ссылки IDA (xrefs) – это механизмы, которые лежат в основе генерации окна вызовов функций.

Окно проблем. С помощью окна проблем IDA информирует вас обо всех трудностях, которые возникли у него в процессе дизассемблирования двоичного файла и о том, какие решения были приняты для их преодоления. Ниже приведен пример набора проблем:

Листинг 4.3.2
Пример набора проблем

```
Address          Type      Instruction
.text:0040104C   BOUNDS    call    eax
.text:004010B0   BOUNDS    call    eax
.text:00401108   BOUNDS    call    eax
.text:00401350   BOUNDS    call    dword ptr [eax]
.text:004012A0   DECISION  push    ebp
.text:004012D0   DECISION  push    ebp
.text:00401560   DECISION  jmp     ds:__set_app_type
.text:004015F8   DECISION  dd 0FFFFFFFFh
.text:004015FC   DECISION  dd 0
```

Каждая проблема характеризуется, во-первых, адресом, в котором она возникла, во-вторых, типом проблемы и, в-третьих, командой в месте возникновения проблемы. В этом примере мы видим проблемы BOUNDS и DECISION. Проблема BOUNDS возникает, когда место назначения вызова или перехода невозможно установить или находится внутри виртуального диапазона адресов программы. Проблема DECISION обычно представляет адрес, по которому IDA решил дизассемблировать байты как команду, а не как данные, несмотря на то, что на этот адрес не было ссылок при просмотре команд алгоритмом рекурсивного спуска.

Полный список типов проблем и их возможных решений доступен во встроенной помощи IDA.

5. НАВИГАЦИЯ В ДИЗАССЕМБЛЕРЕ

5.1. Стековые кадры

Перемещение между окнами – один из важнейших навыков, необходимых для работы в IDA. Как мы увидим, основа базы данных IDA позволяет реализовать исключительные навигационные возможности.

Стековые кадры. При работе с IDA Pro от пользователя ожидается знание особенностей низкоуровневых языков программирования, касающихся тонкостей генерирования машинного кода и использования памяти высокоуровневой программой.

Одно из таких низкоуровневых понятий – стековый кадр. **Стековые кадры** - это блоки памяти, выделенной в стеке программы и предназначенные

для конкретной функции. Программисты, как правило, группируют исполняемые выражения в единицы, называемые *функциями* (также их называют *процедуры, подпрограммы* и *методы*).

Соглашения вызова. Дальнейшие примеры относятся к архитектуре x86 и поведению типичных x86-компиляторов. Вызывающая функция должна хранить параметры именно в том виде, в котором их ожидает принять вызываемая,. Для установки формата, в котором функции будут принимать свои аргументы, существуют соглашения вызова.

Соглашение вызова определяет, куда вызывающая функция должна поместить требуемые параметры. Соглашения вызова могут потребовать поместить параметр в определенные регистры, стек программы или сразу и туда, и туда. При передаче параметров в программный стек также важно определить, кто должен будет удалить их, когда функция закончит свою работу. Некоторые соглашения вызова устанавливают, что вызывающий отвечает за удаление параметров, другие делают ответственной вызываемую функцию. Следование соглашением вызова необходимо для сохранения целостности указателя программного стека.

Размещение локальных переменных. В отличие от соглашений вызова, определяющих способ передачи параметров в функцию, никаких соглашений, определяющих расположение локальных переменных функции, не существует. Имея лишь исходный код обычно невозможно определить, куда будут помещены переменные, потому что ни вызывающая, ни вызываемая функции не определяют это единолично.

Окна стека IDA. Во время начального анализа IDA внимательно следит за указателем стека, помечая каждую команду push и pop, а также многие другие арифметические операции, которые могут изменить указатель стека, например, сложение и вычитание констант. Основная цель анализа – выяснить точный размер зоны локальных переменных, выделенных в стековом кадре функции. Также IDA выясняет, используется ли отдельный указатель кадра (например, распознавая последовательность push ebp/mov ebp, esp) и распознает все ссылки на переменные внутри стекового кадра. Например, если бы IDA пометил следующую инструкцию в теле demo_stackframe

```
mov    eax, [ebp+8]
```

он понял бы, что первый аргумент функции (a в данном случае) загружается в регистр EAX.

В результате внимательного анализа структуры стекового кадра IDA может различать ссылки на память, которые обращаются к аргументам функции (они находятся ниже сохраненного адреса возврата) и ссылки, обращающиеся к локальным переменным (они находятся выше сохраненного адреса возврата). IDA определяет, к каким места внутри стекового кадра есть прямые обращения.

Понимание поведения функции часто сводится к пониманию типов данных, с которыми она оперирует. При чтении листинга дизассемблирования первым ключом к пониманию типа данных функции будет структура стекового

кадра функции. IDA предлагает два окна, отображающих стековый кадр любой функции: общий вид и детальный вид. Чтобы понять оба этих окна, мы рассмотрим следующую версию demo_stackframe, скомпилированную gcc:

```
void demo_stackframe(int a, int b, int c) {
   int x = c;
   char buffer[64];
   int y = b;
   int z = 10;
   buffer[0] = 'A';
   bar(z, y);
}
```

В этом примере мы дали начальные значения переменным y и z, чтобы компилятор не жаловался на неинициализированные переменные, используемые в вызове bar. Кроме того, символ сохранен в первый элемент массива buffer, и мы решили не инициализировать локальную переменную x. Соответствующий результат дизассемблирования IDA показан ниже:

Листинг 5.1
Пример результата дизассемблирования

```
.text:00401090 ; ========= S U B R O U T I N E ============================
  .text:00401090
  .text:00401090 ; Attributes:     ❶ bp-based frame
  .text:00401090
  .text:00401090 demo_stackframe proc near      ; CODE XREF: sub_4010C1+41↓p
  .text:00401090
❹   .text:00401090 var_78          = dword ptr -78h
  .text:00401090 var_74          = dword ptr -74h
  .text:00401090 var_60          = dword ptr -60h
  .text:00401090 var_5C          = dword ptr -5Ch
  .text:00401090 var_58          = byte ptr -58h
  .text:00401090 var_C           = dword ptr -0Ch
  .text:00401090 arg_4           = dword ptr  0Ch
  .text:00401090 arg_8           = dword ptr  10h
  .text:00401090
  .text:00401090                 push    ebp
  .text:00401091                 mov     ebp, esp
  .text:00401093                 sub     esp,  ❷ 78h
  .text:00401096                 mov     eax, [ebp+ ❺ arg_8]
  .text:00401099                 ❻ mov     [ebp+var_C], eax
  .text:0040109C                 ❼ mov     eax, [ebp+arg_4]
  .text:0040109F                 ❼ mov     [ebp+var_5C], eax
  .text:004010A2                 ❽ mov     [ebp+var_60], 0Ah
  .text:004010A9                 ❾ mov     [ebp+var_58], 41h
  .text:004010AD                 mov     eax, [ebp+var_5C]
  .text:004010B0                 ❸ mov     [esp+78h+var_74], eax
  .text:004010B4                 mov     eax, [ebp+var_60]
  .text:004010B7                 ❸ mov     [esp+78h+var_78], eax
  .text:004010BA                 call    bar
  .text:004010BF                 leave
  .text:004010C0                 retn
  .text:004010C0 demo_stackframe endp
```

В этом листинге многое следует пояснить. Начнем с ❶. IDA считает, что эта функция использует регистр EBP как указатель кадра на основе анализа пролога функции. В ❷ мы узнаем, что gcc выделил 120 байтов (78h равно 120) места под локальные переменные в стековом кадре. В него входят 8 байт для передачи двух параметров функции bar ❸, но это все равно гораздо больше 76 байтов, которые мы ранее собирались запросить. Начиная с ❹ IDA предоставляет общий вид стека, перечисляющий каждую переменную, к которой напрямую обращались внутри стека, а также размер переменных и их смещение относительно указателя кадра.

IDA назначает переменным имена на основе их местоположения относительно сохраненного адреса возврата. Локальные переменные находятся над сохраненным адресом возврата, а параметры – под ним. Имена локальных переменных создаются путем сложения префикса var_ с шестнадцатеричным суффиксом, показывающим дистанцию в байтах, на которую переменная отстоит от сохраненного указателя кадра. Локальная переменная var_C в этом случае занимает 4 байта (dword) и находится на 12 байтов выше сохраненного указателя кадра ([ebp-0Ch]). Параметры функции называются с помощью префикса arg_ вместе с шестнадцатеричным суффиксом, показывающим дистанцию до самого верхнего параметра. Таким образом, верхний 4-байтовый параметр будет называться arg_0, а последующие - arg_4, arg_8, arg_C и т.д. В данном примере arg_0 не показан, потому что функция не использует параметр a. Так как IDA не может найти ссылки на память [ebp+8] (местоположение первого параметра), arg_0 не приводится в окне просмотра стека. Быстрый просмотр общего вида стека показывает, что есть много мест в стеке, которые IDA не смогла назвать, потому что в коде программы на них нет прямых ссылок.

Обратите внимание: IDA автоматически генерирует имена только для тех переменных стека, на которые есть непосредственные ссылки внутри функции.

Важное отличие между листингом дизассемблирования IDA и анализом стекового кадра, который мы выполнили ранее, заключается в том, что нигде в ассемблерном листинге мы не видим ссылок на память вроде [ebp-12]. Вместо этого IDA заменил все постоянные смещения на символические имена, соответствующие символам в окне стека и их относительным смещениям относительно указателя стекового кадра. Это позволяет IDA генерировать код более высокого уровня – просто гораздо удобней работать с символическими именами, а не с цифровыми константами. Окно общего вида стека показывает, какие имена, сгенерированные IDA, соответствуют каким смещениям. Например, там, где в коде появляется ссылка на память [ebp+arg_8], можно использовать [ebp+10h] или [ebp+16].

5.2. Поиск в базе данных

Если вы внимательно осмотрите содержимое меню Search, вы обнаружите длинный список опций, большинство из которых приведут вас к следующему

пункту какой-нибудь категории. Search ► Next Code перемещает курсор к следующему месту, содержащему команду. Вы также можете ознакомиться с опциями, доступными в меню Jump.

Текстовый поиск. Текстовый поиск IDA позволяет искать подстроку в листинге дизассемблирования. Текстовый поиск можно вызвать с помощью меню Search ► Text (горячая клавиша: ALT-T). Некоторое количество говорящих сами за себя функций позволяют тонкую настройку поиска.

Опция *Find all occurences* отображает результат поиска в новом окне, обеспечивая легкую навигацию к любому совпадению с критериями поиска. Предыдущий поиск может быть повторен для нахождения следующего совпадения по горячей клавише CTRL-T или из меню Search ► Next Text.

Двоичный поиск. Если вы хотите найти определенную двоичную информацию, например, последовательность байтов, текстовый поиск не подходит. Вместо этого используйте двоичный поиск IDA. Текстовый поиск ищет в окне дизассемблирования, бинарный поиск может проводиться только в часть шестнадцатеричного окна. Можно искать как в шестнадцатеричном дампе, так и в ASCII-дампе, в зависимости от того, как вы укажете строку для поиска. Бинарный поиск вызывается из меню Search ► Sequence of Bytes либо горячей клавишей ALT-B. Чтобы искать последовательность шестнадцатеричных байтов, поисковая строка должна быть представлена в виде разделенных пробелом двузначных двоичных значений, например, CA FE BA BE. К идентичному результату приведет поиск ca fe ba be, несмотря на включение опции «учитывать регистр».

Чтобы искать фрагмент ASCII-дампа в шестнадцатеричном окне, нужно заключить поисковой запрос в кавычки. Опция Unicode strings используется для поиска вашей строки в юникоде.

Опция Case-sensitive может вас запутать. С поиском строк все понятно: поиск "hello" найдет "HELLO", если Case-sensitive отключен. Всё становится немного интересней, если вы делаете шестнадцатеричный поиск с выключенным Case-sensitive. Если вы будете искать E9 41 C3, вы будете удивлены, что найдется E9 61 C3. Строки считаются совпадающими, потому что 0x41 соответствует символу «A», а 0x61 – символу «a». Так что не смотря на то, что вы указали шестнадцатеричный поиск, 0x41 считается равным 0x61, если не включена опция «Учитывать регистр».

Поиск последующих совпадений двоичных данных выполняется с помощью CTRL-B или Search ► Next Sequence of Bytes.

Вопросы:

1. Перечислите опции сохранения базы данных в IDA.
2. Назовите и охарактеризуйте основные окна IDA.
3. Раскройте суть понятия Соглашение вызова.

РАЗДЕЛ 3. МАНИПУЛИРОВАНИЕ РЕЗУЛЬТАТОМ ДИЗАССЕМБЛИРОВАНИЯ И МЕТОДЫ ЗАЩИТЫ ПРИЛОЖЕНИЙ

6. МАНИПУЛИРОВАНИЕ РЕЗУЛЬТАТОМ ДИЗАССЕМБЛИРОВАНИЯ

Одной из наиболее мощных функций, которые предлагает IDA, является возможность легко управлять дизассемблированием в плане добавления новой информации или изменения листинга. IDA автоматически обрабатывает операции, такие как глобальный поиск и замена, и делает тривиальной работу, состоящую из переформатирования инструкций и данных и наоборот, функции, не доступной в других инструментах дизассемблирования.

6.1 Имена и присваивание имён

В этом пункте мы столкнемся с двумя типами имен, встречающиеся при дизассемблировании с помощью IDA: имена, связанные с виртуальными адресами (именованными областями) и имена, относящиеся к переменным структуры стека. В большинстве случаев IDA автоматически генерирует эти имена согласно ранним документациям, причем ссылается на них, как на фиктивные.

Эти имена редко указывают на назначение области или переменной, а значит, не облегчают наше понимание программы. Благодаря IDA, Вы с легкостью измените все имена, в то время как IDA будет обрабатывать любые детали в изменениях ассемблерного кода. В большинстве случаев, изменение имени – это простая операция: клик на нужном имени подсветит его, после чего нужно нажать на клавишу N для открытия диалогового окна изменения имени. Есть другой вариант: щелчок правой клавишей мыши вызовет меню, в котором есть пункт – Rename.

Параметры и локальные переменные. Имена, связанные с переменными стека, являются самой простой формой именования в листинге ассемблерного кода, прежде всего потому, что они не связаны с определенным виртуальным адресом, а значит, они не могут появиться в окне Names. Как и в большинстве языков программирования, такие имена ограничены в возможностях внутри функции, которой принадлежит стековый фрейм. Таким образом, у каждой функции в программе должна быть переменная стека arg_0, но ни у одной функции не может быть больше чем одной переменной с именем arg_0.

Как только будет введено новое имя, IDA позаботится о том, чтобы каждое упоминание старого имени в контексте текущей функции изменилось на новое.

Если Вы когда-нибудь захотите вернуться к имени по умолчанию для данной переменной, то в предыдущем диалоговом окне строку нужно оставить пустой, тогда IDA сгенерирует имя, зарезервированное по умолчанию.

Именованные области. Переименование именованной области или обозначение имени области, которая его ещё не имела, проходит немного по-другому, нежели переименование переменной стека. Процесс доступа к диалоговому окну идентичен (по горячей клавише N).

Параметр «Maximum length of new names» дублирует значение из одного из файлов конфигурации IDA (<IDADIR>/cfg/ida.cfg). Вы можете использовать имя переменной и длиннее 15-ти символов, правда тогда IDA предложит вам увеличить максимально допустимую длину имени. Согласившись на это изменение, вы подтвердите изменение максимальной длины в рамках текущей базы данных. Любые другие базы данных, которые Вы создадите в дальнейшем, будут использовать значение, содержащееся в конфигурационном файле.

Возможные атрибуты, которые связаны с именем области:

– Локальное имя (local name) ограничено возможностями текущей функции, таким образом реализована уникальность имен в пределах одной функции. Как и локальные переменные, две различные функции могут содержать два идентичных локальных имени, однако два идентичных локальных имени не могут принадлежать единственной функции. Именованные области, которые находятся вне границ функции, не определяются как локальные имена. Как и глобальные переменные, они включают в себя имена функций.

– Включение в список имен (including in names list). Выбирая эту опцию, вы добавляете введенное имя в окно Names, которое может облегчить поиск данного имени, когда Вы захотите к нему вернуться. Имена, сгенерированные автоматически (фиктивные), по умолчанию никогда не включаются в окно Names.

– Общедоступное имя (public name). Обычно – это имя, которое экспортируется двоичным файлом как общая библиотека. Синтаксические анализаторы IDA обычно обнаруживают такие имена при парсинге заголовков во время начальной загрузки в базу данных. Выбирая этот атрибут, вы заставляете обработчик определять символ как public name.

– Автоматически сгенерированное имя (Autogenerated name). Этот атрибут не оказывает никакого видимого эффекта на дизассемблирование.

– Слабое имя (weak name). Слабый символ – специализированная форма общедоступного символа, использующегося только когда никакой общедоступный символ того же имени не находится для переопределения.

– Create name anyway. Для областей вне функции (в глобальной видимости) нельзя давать одинаковые имена.

Если Вы редактируете имя в пределах глобальных границ (например, имя функции или глобальной переменной) и пытаетесь присвоить имя, которое уже используется в базе данных, IDA предложит сгенерировать уникальный числовой суффикс, чтобы разрешить данный конфликт.

Если вы редактируете локальное имя в пределах функции и пытаетесь присвоить имя, которое уже используется, то по умолчанию Ваша попытка

будет отклонена. Выбирая «Create name anyway», вы вынуждаете IDA сгенерировать уникальный числовой суффикс для этого локального имени.

Имена регистров. IDA позволяет переименовывать регистры в пределах границ функции. Переименование регистра работает почти такие же, как и переименование в любой другой области. Используйте горячую клавишу N, или щелкните правой кнопкой по имени регистра, и выберите Rename, чтобы открыть диалоговое окно. Когда Вы переименовываете регистр, Вы, в действительности, задаете псевдоним, по которому можно обратиться к регистру для текущей функции. IDA выполнит замену всех экземпляров имени регистра новым псевдонимом. Невозможно переименовать регистр, используемый в коде, который не принадлежит функции.

6.2 Комментарии в IDA

Комментарии полезны для описания последовательностей инструкций ассемблера высокоуровневым способом. Например, Вы могли бы записать комментарии, используя операторы языка C, чтобы описать поведение определенной функции.

IDA предлагает несколько различных стилей комментариев, каждый из которых служит для определенной цели. Комментарии могут быть связаны с любой строкой ассемблерного кода с помощью опции Edit ► Comments. К функции комментариев IDA можно обратиться через горячие клавиши или контекстные меню.

Большинство комментариев IDA снабжено префиксом «точка с запятой», чтобы указать, что остаток строки нужно считать комментарием. Это подобно стилям комментариев, используемых многими ассемблерами, и приравнивается к # - во многих языках сценариев или // - в C++.

Стандартные комментарии. Самый простой комментарий - стандартный. Стандартные комментарии помещены в конце ассемблерных строк. Щелкните правой кнопкой в правом поле дизассемблирования или используйте двоеточие (горячая клавиша :), чтобы активировать диалоговое окно записи комментария. Стандартные комментарии можно разместить на нескольких строках, если Вы будете использовать многострочные комментарии в диалоговом окне записи комментария. Каждая из строк будет отмечена отступом, чтобы выстроить комментарии в линию на правой стороне ассемблерного кода. Чтобы отредактировать или удалить комментарий, Вы должны вновь открыть диалоговое окно записи комментария и, соответственно, отредактировать или удалить весь текст комментария. По умолчанию стандартные комментарии выделены на экране синим цветом.

Повторяющиеся комментарии. Этот тип комментариев примечателен тем, что, будучи поставлен однажды, он может автоматически появляться во многих местах в ассемблерном коде. Свойство повторяющихся комментариев связано с понятием перекрестных ссылок. Когда одна программная область обращается ко второй, которая содержит повторяющийся комментарий, тогда комментарий, связанный со второй областью, отражен в первой. По умолчанию

отраженный комментарий выделен серым. Горячая клавиша для таких комментариев - точка с запятой (;).

Различная форма повторяющегося комментария связана со строками. Всякий раз, когда IDA автоматически создает строковую переменную, виртуальный повторяющийся комментарий добавляется в область этой переменной. Мы говорим виртуальный, потому что этот комментарий не может редактировать пользователь. Текст виртуального комментария добавляется в содержимое строковой переменной и выводится на экран везде в базе данных, как повторяющийся комментарий. В результате любые области программы, которые обращаются к строковой переменной, выведут на экран строковую переменную вместе с повторяющимся комментарием.

«Следующие» и «предыдущие» строки. Данный тип комментариев выглядит как целая строка, которая появляется до (anterior) и после (posterior) ассемблерной строки. Такие комментарии цветом никак не отмечены и встречаются только в IDA.

Функциональные комментарии. Функциональные комментарии позволяют группировать и отображать комментарии вверху листинга ассемблерного кода функции. Комментарий вставляется при первом появлении имени функции в самом её начале, а затем добавляется либо обычный, либо повторяющийся комментарий. Повторяющиеся функциональные комментарии видны в любых областях, которые вызывают прокомментированную функцию.

6.3 Первичное преобразование кода

Преобразование кода IDA включает следующие операции:

- Преобразование данных в код.
- Преобразование кода в данные.
- Преобразование последовательности инструкций в функцию.
- Изменение начального и конечного адреса существующих функций.
- Изменение формата вывода операндов.

Множество факторов влияют на степень использования этих операций, включая Ваше личное мнение.

Параметры отображения кода. Каждая строка дизассемблированного кода состоит из частей, которые IDA называет «частями дизассемблированных строк». В такой строке всегда присутствуют метки, мнемоника и операнд. Вы можете выбрать дополнительные параметры для каждой дизассемблированной строки, используя Опции ► Общие (Option►General) на вкладке Disassembly.

Отображение частей дизассемблированных строк содержит несколько опций, которые выбраны по умолчанию: настройка текстового представления дизассемблирования IDA - префиксов строки, комментариев и повторяющихся комментариев.

Префиксы строки section: address – это часть каждой дизассемблированной строки. Если убрать галочку с этой опции, то из каждой строки дизассемблирования будет удален префикс.

Выбор Stack Pointer позволит видеть на экране относительное изменение по мере выполнения каждой функции. Это может быть полезно в распознавании несоответствий в условных обозначениях или необычных манипуляций с указателем на вершину стека.

Всякий раз, когда IDA встречается с функциональным оператором возврата и обнаруживает, что значение указателя вершины стека не нуль, отмечается ошибка, строка инструкции выделяется красным цветом. В некоторых случаях это может быть преднамеренной попыткой прервать автоматизированный анализ.

Простые и повторяющиеся комментарии. Отмена любой из этих опций запрещает отображать определенный тип комментария.

Автоматические комментарии. IDA может иногда напомнить, как именно выполняются некоторые команды, используя для этого автоматические комментарии. Для таких тривиальных инструкций, как x86 mov, комментарии естественно не добавляются. Комментарии пользователей всегда приоритетнее автоматических; поэтому, если Вы хотите видеть автоматический комментарий IDA для строки, Вам придется удалить все собственные комментарии (обычные или повторяющиеся).

Некорректная команда, маркер <BAD>. IDA может отмечать команды, корректные для процессора, но не распознаваемые некоторыми компиляторами. Недокументированные (в противоположность недопустимым) команды ЦП также могут попасть в эту категорию. В таких случаях IDA дизассемблирует инструкцию как последовательность байтов данных и выведет на экран недокументированную команду как комментарий с предисловием <BAD>.

Количество байтов кода операции. IDA позволяет Вам просматривать байты машинного языка, связанные с каждой инструкцией, синхронизируя шестнадцатеричный вывод с дисплеем листинга дизассемблирования. Дополнительно Вы можете просмотреть байты машинного языка с командами ассемблера, определяя число байтов машинного языка, которые IDA должна выводить на экран для каждой команды.

Форматирование операндов команд. Во время процесса дизассемблирования IDA принимает много решений относительно того, как отформатировать операнды, связанные с каждой инструкцией. Самые большие проблемы обычно связаны с тем, как отформатировать различные целочисленные константы, использующие большое разнообразие типов команд. Чтобы сделать результат дизассемблирования более читабельным, IDA пытается, когда возможно, использовать символьные имена, а не числа. Во многих других случаях, точный контекст, в котором используется константа, возможно, не вполне четкий. Когда это происходит, константа обычно форматируется как шестнадцатеричная константа.

Щелчок правой кнопкой по любой константе в ассемблерном коде открывает контекстно-зависимое меню. В этом случае предлагается переформатировать константу в десятичный, восьмеричный, или двоичный формат.

Очень часто в своем исходном коде программисты используют именованные константы. Такие константы могут быть результатом #define операторов (или их эквивалента), или могут принадлежать ряду исчисляемых констант. К сожалению, к тому времени, когда компилятор заканчивает с исходным кодом, уже невозможно определить, использовалась ли в исходнике символьная или числовая константа. IDA поддерживает большой каталог констант, связанных со многими библиотеками, такими как стандартная библиотека C или API Windows. Этот каталог доступен в опциях символьной константы – *Use standard symbolic constant* в меню любой константы.

6.3.1 Управление функциями

У Вас найдется много причин, чтобы захотеть управлять функциями после того, как был завершен начальный анализ. Например, если IDA не в состоянии определить местоположение вызова функции, она не будет распознана. Или IDA не может определить местоположение конца функции, требуя вмешательство с Вашей стороны, чтобы исправить дизассемблирование. IDA может быть трудно определить местоположение конца функции, если компилятор разделил функцию на несколько диапазонов адресов или когда в процессе оптимизации кода компилятор объединил общие последовательности конца двух или больше функций, чтобы оставить свободное место.

Создание новых функций. Новые функции могут быть созданы существующими командами, которые уже не принадлежат функции, или из байтов необработанных данных, которые не были определены IDA (такими как двойные слова или строки). Для создания функции, поместите курсор в первый байт или команду, которая будет включена в новую функцию, и выберите Edit ► Functions ► Create Function. IDA пытается преобразовать данные, чтобы кодировать их в случае необходимости. Затем сканирует следующие строки, чтобы проанализировать структуру функции и найти оператор возврата. Если IDA сможет определить местоположение конца функции, она генерирует новое имя функции, анализирует стековый фрейм, и реструктурирует код в теле функции. Если же она не смогла определить местоположение конца функции или встретилась с какими-либо запрещенными командами, то операция прерывается.

Удаление Функций. Удалить существующую функцию вы можете, нажав на Edit ► Functions ► Delete Function, если думаете, что IDA допустила ошибку в анализе.

Функциональные Блоки. Функциональные блоки обычно находятся в коде, сгенерированном компилятором Microsoft Visual C++. Блоки - результат перемещения компилятором участков кода, которые реже всего выполняются, чтобы уплотнить часто выполняемые блоки в страницах памяти, которые вряд ли будут выгружены.

Когда функция разделена таким способом, IDA пытается определить местоположение всех связанных блоков, следуя по переходам, которые ведут к каждому блоку. В большинстве случаев IDA правильно определяет

местоположения всех блоков по списку названий каждого блока в заголовке функции.

В некоторых случаях IDA не может определить местоположение каждого функционального блока, а в некоторых, функция не распознается как блочная. В таких случаях Вы должны вручную удалить созданные блоки или создать их самим.

Вы создаете новые функциональные блоки, выбирая диапазон адресов, принадлежащих блоку, который не должен быть частью другой уже существующей функции, выбрав для этого Edit ► Functions ► Append Function Tail. Далее Вас попросят выбрать родительскую функцию из списка всех определенных функций.

В листинге ассемблерного кода функциональные блоки упоминаются как function chunks, а в меню IDA блоки функций названы function tails.

Для удаления функционального блока, подведите курсор к любой строке блока и выберите Edit ► Functions ► Remove Function Tail. Далее Вас попросят подтвердить своё решение.

Вы можете сделать так, чтобы IDA не создала функциональные блоки, убрав галочку с Create functional tails loader при загрузке файла в IDA. Эта одна из опций загрузчика, доступная через Опции Ядра(Kernel options) (см. Главу 4) в начальном диалоговом окне загрузки файла. Если Вы отключаете Function Tails, Вы можете заметить, что теперь там содержатся переходы к областям вне границ функции. IDA выделяет такие переходы красным цветом и стрелками в окнах на левой стороне дизассемблирования. В графическом представлении для соответствующей функции не выведены на экран цели таких переходов.

Функциональные параметры. В IDA у функций есть много параметров:

Имя функции - альтернативный путь для смены имени функции.

Начальный адрес - адрес первой команды в функции. IDA чаще всего определяет это автоматически, либо во время анализа или от адреса, используемого во время создания функции.

Конечный адрес - наиболее часто это адрес команды возврата. IDA чаще всего определяет это автоматически, либо во время анализа или от адреса, используемого во время создания функции.

Область локальных переменных - представляет собой число байтов в стеке, выделенных для локальных переменных функции.

Сохраненные регистры (Saved registers) - число байтов, используемых, чтобы сохранить значение регистров от имени вызывающей стороны. IDA полагает, что сохраненная область регистра лежит выше сохраненного обратного адреса и ниже любых локальных переменных функции.

Очищенные(purged) байты - очищенные байты показывают число байтов параметров, которые функция удаляет из стека, когда возвращала значение.

Допустимая ошибка (корректировка) указателя фрейма. В некоторых случаях компиляторы могут скорректировать указатель на фрейм функции, чтобы указать примерно в середину области локальной переменной, а не в

сохраненное значение указателя фрейма у основания области локальной переменной. Это расстояние от скорректированного указателя фрейма до сохраненного указателя фрейма называют допустимой ошибкой указателя фрейма. В большинстве случаев любая допустимая ошибка указателя фрейма вычисляется автоматически при анализе. Компиляторы используют допустимую ошибку стекового фрейма для оптимизации скорости. Назначение допустимой ошибки заключается в том, чтобы сохранить как можно больше переменных стекового фрейма в пределах досягаемости введенного смещения 1 байта (-128. + 127) от указателя фрейма.

Дополнительные флажки опции доступны, чтобы характеризовать функцию. Как с другими полями в пределах диалогового окна, эти флажки обычно отражают результаты автоматического анализа IDA. Следующие атрибуты могут быть как включены, так и нет.

Невозвращение: функция не возвращается к ее вызывающей стороне.

Далекая функция используется, чтобы отметить функцию как далекую функцию на сегментированной архитектуре.

Библиотека func отмечает функцию как библиотечный код. Библиотечная функция отмечается в коде цветом, определенным только для таких функций, чтобы отличать её от обычных функции,

Статический func показывает только список всех статических параметров.

Фрейм, базирующийся на BP(BP based frame показывает, что функция использует указатель фрейма. Для фреймов, основанных на указателях, ссылки на память, которые используют указатели фреймов, отформатированы так, чтобы использовать символьные имена переменной стека, а не числовые смещения.

BP эквивалентен SP. Некоторые функции конфигурируют указатель фрейма, чтобы указать на вершину стекового фрейма (наряду с указателем вершины стека) после ввода функции. По существу это то же самое, что и наличие допустимой ошибки указателя фрейма, равной области локальной переменной.

Корректировка Указателя вершины стека. Если IDA не удается определить, изменяется ли указатель на команду вершины стека, вам придется вручную подкорректировать указатель на вершину стека.

Самый простой пример такого случая – когда происходит вызов функции другой функцией, используя stdcall. Если вызванная функция находится в совместно используемой библиотеке, о которой IDA неизвестно, то IDA не будет знать, что функция использует stdcall и будет не в состоянии учесть факт, что указатель вершины стека будет изменен вызванной функцией до возврата. Таким образом, IDA отобразит неточное значение указателя вершины стека для остальной части функции.

6.3.2. Преобразование данных для кодирования

При автоматическом анализе байты данных могут быть неправильно классифицированы как байты кода и дизассемблированы в команды, или байты кода могут быть неправильно классифицированы как байты данных и отформатированы как значения. Это происходит по многим причинам. Например, некоторые компиляторы встраивают данные в раздел кода программ или на некоторые байты кода никогда не ссылаются непосредственно как на код, и IDA решает не дизассемблировать их. Запутанные программы в особенности имеют тенденцию размывать различие между разделами кода и разделами данных.

Независимо от причины, по которой Вы хотите переформатировать свой код ассемблера, это сделать довольно легко. Первым делом удалите свое текущее форматирование (кода или данных). Можно убрать переопределение функции, кода или данных, щелкнув правой кнопкой по элементу, который Вы хотите переопределить и выбрать *undefine* (также Edit ► Undefine или горячая клавиша U) в выпадающем меню. Такое переопределение элемента заставляет базовые байты быть переформатированными как список необработанных значений. Чтобы переопределить большие области, используйте метод буксировки. К примеру:

```
.text:004013E0 sub_4013E0      proc near
.text:004013E0                 push    ebp
.text:004013E1                 mov     ebp, esp
.text:004013E3                 pop     ebp
.text:004013E4                 retn
.text:004013E4 sub_4013E0      endp
```

Переопределение этой функции привело бы к появлению серии несортированных байтов, показанных здесь, которые мы очень хотели переформатировать:

```
.text:004013E0 unk_4013E0      db  55h ; U
.text:004013E1                 db  89h ; ë
.text:004013E2                 db 0E5h ; s
.text:004013E3                 db  5Dh ; ]
.text:004013E4                 db 0C3h ; +
```

Чтобы дизассемблировать последовательность неопределенных байтов, щелкните правой кнопкой по первому нужному байту, и выберите Код (также Edit ► Code или горячая клавиша C). Это заставляет IDA дизассемблировать все байты, пока она не встречается с определенным элементом или запрещенной командой. Чтобы преобразовать большие области, используйте метод буксировки.

Преобразование кода в данные происходит немного сложнее. Во-первых, невозможно преобразовать код в данные, используя контекстное меню. Можно использовать только Edit ► Data или горячую клавишу D. Для объемного преобразования команд в данные легче всего сначала выполнить переопределение всех команд, которые Вы хотите преобразовать в данные, а затем форматировать уже данные.

7. ТИПЫ ДАННЫХ И СТРУКТУРЫ ДАННЫХ

Самый простой метод для того, чтобы связать определенный тип данных с переменной - использование переменной в качестве параметра к функции, о которой мы хоть что-то знаем. Во время ее анализа IDA комментирует типы данных, основанные на использовании переменных в функции, для которой у IDA есть прототип. Когда возможно, IDA использует имя формального параметра, взятого от прототипа функции вместо того, чтобы генерировать фиктивное имя по умолчанию для переменной. Это можно заметить в следующем дизассемблировании вызова connect:

```
.text:004010F3                push    10h             ; namelen
.text:004010F5                lea     ecx, ❶[ebp+name]
.text:004010F8                push    ecx             ; name
.text:004010F9                mov     edx, ❶[ebp+s]
.text:004010FF                push    edx             ; s
.text:00401100                call    connect
```

В нем мы можем видеть, что каждое нажатие было прокомментировано именем параметра (взятого из базы IDA о прототипе функции). Кроме того, две локальных переменные стека ❶ были названы по имени параметров, которым они соответствуют. В большинстве случаев эти имена будут намного более информативными, чем фиктивные имена, генерируемые IDA.

Недостаток IDA в том, что она распознает только параметры функции, которые помещены в стек.

Во многих случаях IDA будет знать прототип функции, и вы увидите это, наведя мышь на ее имя. Если о функции IDA ничего не известно, то лучше всего Вам обратиться к справочной литературе.

7.1 Использование распознавания структуры данных

Прежде, чем мы сможем обсудить функцию IDA для улучшения читабельности кода, который использует сложные типы данных, мы должны рассмотреть, на что похож такой код.

7.1.1. Доступ к элементам массива

Массивы - самая простая компонентная структура данных с точки зрения расположения памяти. Традиционно, массивы - непрерывные блоки памяти, которые содержат последовательные элементы одного типа данных. Размер массива легко вычислить – это произведение количества элементов на их размер.

Получить доступ к отдельным элементам можно, используя индексное значение, которое может быть переменной или константой. Способ, которым получают доступ к элементу массива, зависит не только от типа используемого индекса, но и от того, в пределах какого пространства памяти программы выделен участок под массив.

Глобальные Массивы. Когда массив определен в пределах глобальной области данных программы (в пределах .data или раздела .bss, например),

начальный адрес массива компилятор определяет во время компиляции. Фиксированный начальный адрес позволяет компилятору вычислить фиксированные адреса для любого элемента массива, к которому можно получить доступ, используя фиксированный индекс.

Дизассемблирование не даст нам точных данных о том, что в коде присутствует глобальный массив.

Использование значений индексных переменных приводит нас к началу массива, потому что базовый адрес будет показан, когда вычисленное смещение будет добавлено к нему, чтобы вычислить фактическое расположение массива.

Стековые Массивы. На практике компиляторы обрабатывают стековые массивы почти как и глобальные. Однако во время компиляции не известен адрес, в котором будет выделен массив, таким образом, компилятор не может вычислить адрес этого массива, в отличие от того, как это было бы сделано в глобальном массиве.

«Кучевые» (heap-allocated) Массивы. Кучевые массивы образуются, используя функцию динамического выделения памяти, такую как malloc (C) или new (C++). С точки зрения компилятора разница в том, что он должен генерировать все ссылки в массив, основанный на значении адреса, возвращенном из функции выделения памяти.

7.1.2. Доступ к элементам структуры

Главным отличительным признаком структур является то, что к полям данных в пределах структуры получают доступ по имени, а не по индексу, как это реализовано в массивах. К сожалению, имена полей преобразованы в числовые смещения компилятором. Таким образом, к тому времени, когда вы посмотрите на результат дизассемблирования, доступ к полю структуры становится удивительно похожим на доступ к элементам массива через постоянные индексы.

Минимальное требуемое пространство для структуры определено суммой пространства, требуемого каждому полю в пределах структуры. По умолчанию компиляторы стремятся выровнять поля структуры по адресам памяти, которые позволяют эффективнее читать и записывать в них.

Глобальные Структуры. Как и с глобальными массивами, адрес глобальных структур известен во время компиляции. Это позволяет компилятору вычислять адрес каждого элемента структуры во время компиляции и избавляет от необходимости делать любые вычисления во время выполнения.

Стековые Структуры. Как и стековые массивы, стековые структуры трудно распознать, основываясь только на расположении стека.

В случае стековых массивов в результате дизассемблирования мы получим пять отдельных переменных, а не единственную, которая содержит пять разных полей.

«Кучевые» Структуры. Кучевые структуры, оказывается, намного более прозрачны относительно размера структуры и расположения ее полей. Когда

кучевая структура выделена в программе, у компилятора нет никакого выбора, кроме как генерировать код, чтобы вычислить надлежащее смещение в структуре всякий раз, когда к полю требуется доступ. Для кучевой структуры единственная ссылка доступная компилятору, является указателем на начальный адрес структуры.

Массивы Структур. Предыдущие знания из глав о массивах и структурах применяются точно также при контакте с вложенными типами.

7.2. Создание Структур IDA

В следующих разделах мы рассмотрим как средства IDA могут улучшить читабельность кода, который управляет структурами.

7.2.1. Ручное Определение Структуры

Всякий раз, когда Вы обнаруживаете, что программа управляет структурой данных, Вы должны решить, хотите ли Вы включить имена полей структуры в код ассемблера или Вы можете понять все числовые смещения, появляющиеся в коде. В некоторых случаях IDA может распознать использование структуры, как часть стандартной библиотеки C или API Windows.

Создание Новой Структуры (или Объединение). Когда программа использует структуру, о которой у IDA нет никакой информации, IDA предлагает определить состав структуры и включить в дизассемблирование новую. Создание структуры в IDA происходит в окне Structures. Никакая структура не может быть включена в дизассемблирование, пока она сначала не занесена в окне Structures. Любая структура, которая известна IDA и распознана программой, автоматически присутствует в окне Structures.

Первые четыре строки в окне *Structures* служат постоянным напоминанием об операциях, которые возможны в пределах окна. Основные операции, которые нам нужны - добавление, удаление, и редактирование структур. Добавление структуры инициируется, используя клавишу INSERT.

Чтобы создать новую структуру, Вы должны задать имя в поле имени *Structure*. Первые два флажка определяют, где будет новая структура выведена на экран и будет ли выведена вообще. Третий флажок, *Create Union,* определяет, будет ли структуру C или нет. Для структур размер вычисляется как сумма размеров каждого компонента, в то время как для объединений, размер вычисляется как размер самого большого компонента. «Добавить стандартную структуру» используется для доступа к списку всех известных IDA типов данных структур. Действие этой кнопки обсуждается в "Использовании Стандартных Структур" (Раздел 8.5). Как только Вы определяете имя структуры и нажимаете OK, пустая структура создастся в окне *Structures*.

Редактирование Элементов структуры. Чтобы добавить поля к Вашей новой структуре, Вы должны использовать команды D, A, и звездочку (*). Первоначально, только команда D полезна, и, к сожалению, ее действие очень зависит от расположения курсора.

Чтобы добавить новое поле к структуре, расположите курсор на последнюю строку определения структуры (содержащий ends) и нажмите D. Это заставляет новое поле быть добавленным в конец структуры. Размер нового поля будет установлен согласно первому размеру, выбранному на карусели данных. Имя поля первоначально будет field_N, где N - числовое смещение от начала структуры к началу нового поля (field_0, например).

Если вам необходимо изменить размер поля, вы можете сделать это, убедившись, что курсор установлен на новом имени поля и выбрав правильный размер данных, затем нужно несколько раз нажать на клавишу D для переключения между типами данных на карусели данных. Также вы можете использовать Options ► Setup Data Types, чтобы определить размер, который не доступен на карусели данных. Если поле - массив, щелкните правой кнопкой по имени и выберите Array, чтобы открыть диалоговое окно спецификации массива.

Чтобы изменить название поля структуры, щелкните по имени поля и используйте горячую клавишу N, или щелкните правой кнопкой по имени и выберите *Name*; задайте новое имя для поля.

Стековые фреймы как специализированные структуры. Вы наверняка заметили, что определение структуры выглядит подобно подробному виду стекового фрейма. Это не случайно, поскольку IDA обрабатывает обоих одинаково.

7.3. Использование шаблонов структур

Есть два способа использовать объявление структуры в коде. Во-первых, Вы можете переформатировать ссылки в памяти, чтобы сделать их более читаемыми, преобразовывая числовые смещения структуры, такие как [ebx+8] в символьные ссылки, такие как [ebx+ch8_struct.field4]. Последняя форма предоставляет гораздо больше информации о том, на что ссылаются. Во-вторых, использование шаблонов структур можно для обеспечения дополнительных типов данных, которые могут быть применены к стеку и глобальным переменным.

IDA позволяет Вам переформатировать любую постоянную величину, используемую в операнде в эквивалентное символьное представление.

Как альтернатива форматированию отдельных ссылок памяти, стек и глобальные переменные могут быть отформатированы как все структуры. Чтобы отформатировать переменную стека как структуру, откройте подробное представление стекового фрейма, дважды щелкнув по нужной переменной, и затем Edit ► Struct Var (ALT-Q), чтобы вывести на экран список известных структур.

При выборе одной из имеющихся структур, содержащей в себе соответствующее количество байт в стеке структуры соответствующего типа, переформатируются все необходимые ссылки памяти в структуру ссылок.

Процедура форматирования глобальных переменных как структуры почти идентична используемой для переменных стека. Чтобы сделать это, выберите

переменную или адрес, который отмечает начало структуры, и используйте Edit ► Struct Var (ALT-Q), чтобы выбрать соответствующий тип структуры.

7.4. Импортирование новых структур

IDA способна к парсингу индивидуальных C (не C++) объявлений данных, так же, как и всех типов файлов C, и автоматически созданных представлений структуры IDA.

Парсинг структур Си. Версия 5.2 IDA ввела подвид Local Types, доступный через View ► OpenSubviews ► Local Types, в окне *Local Types* представляется как список всех типов, которые были проанализированы в текущей базе данных. Для новых баз данных окно *Local Types* первоначально пусто, но предлагается возможность проанализировать новые типы через клавишу INSERT или опцию Insert в контекстном меню.

В течение парсинга нового типа на экран в окне сообщений IDA могут выводиться сообщения об ошибках. Если описание типа успешно проанализировано, тип и его объявление перечисляются в окне *Local Types*,

Типы данных, добавленные в окно *Local Types*, не сразу доступны через окно *Structures*. Вместо этого каждый новый тип добавляется к списку стандартных структур и должен быть импортирован в окно *Structures*.

Parsing C Header Files. Чтобы проанализировать эти файлы, используйте File ► Load File ► Parse C Header File, чтобы выбрать заголовок, который Вы хотите проанализировать. Любые сообщения об ошибках будут выведены на экран в окне сообщений IDA. В другом случае, если ошибок нет, Вы увидите сообщение об успешной компиляции.

IDA добавляет все структуры, которые были успешно проанализированы к ее списку стандартных структур (в конец списка, чтобы быть точным) доступных в текущей базе данных. Когда у новой структуры есть то же имя, что и у существующей структуры, она будет перезаписана с новым расположением структуры. Ни одна из новых структур не появится в окне *Structures*, пока Вы не добавите их явно.

8. ПЕРЕКРЕСТНЫЕ ССЫЛКИ И ПОСТРОЕНИЕ ГРАФА

Некоторыми из большого количества общих вопросов, кроме вопросов о реверсивном проектировании двоичных файлов, являются "Откуда эта функция вызвана?" и "Какие функции получают доступ к этим данным?"

IDA помогает ответить на подобные типы вопросов с помощью ее обширных функций перекрестных ссылок. IDA обеспечивает много механизмов для отображения и доступа к данным перекрестной ссылки, включая возможности генерации графа, которые обеспечивают визуальное представление отношений между кодом и данными.

8.1. Перекрестные ссылки

Начиная наше обсуждение, заметим, что перекрестные ссылки в пределах IDA часто упоминаются просто как *xrefs*. В пределах этого текста мы будем использовать *xref* только там, где это нужно, чтобы обратиться к пункту меню

IDA или диалогового окна. Во всех других случаях мы будем придерживаться термина перекрестная ссылка.

Есть две основных категории перекрестных ссылок в IDA: перекрестные ссылки на код и перекрестные ссылки на данные. В каждой категории мы описываем несколько различных типов перекрестных ссылок. С каждой перекрестной ссылкой связано определенное направление. Все перекрестные ссылки совершают переход от одного адреса к другому. Это могут быть адреса кода или данных.

Перекрестные ссылки кода - очень важное понятие, поскольку они облегчают генерацию в IDA графов потока управления и графов вызова функции, каждый из которых мы обсудим позже.

Перекрестные ссылки на код. Перекрестная ссылка кода используется, чтобы указать, что инструкция передает или может передать управление другой инструкции и описывается как переход в IDA. IDA различает три основных типа перехода: обычный, прыжок, и вызов. Прыжок и вызов разделены по признаку, является ли конечный адрес близким или далеким адресом. Далекие адреса встречаются только в двоичных файлах, которые используют сегментированные адреса.

Обычный переход - самый простой тип перехода, и представляет собой переход от одной инструкции к другой. Это стандартный переход для всех инструкций, не являющихся командами.

Инструкции, используемые, чтобы вызвать функции, относятся к **переходам типа вызова**, указывая на передачу управления к целевой функции. В большинстве случаев обычный переход также принадлежит к числу команд вызова, поскольку большинство функций возвращается в место, которое следует за вызовом. Если IDA знает, что функция не возвращает значение, то вызовы этой функции не будут иметь обычный переход.

Прыжок присваивается каждой команде безусловного перехода и команде условного перехода. Условным переходам также присвоены обычные переходы, чтобы управлять переходом по ссылке, когда ответвлений нет. У безусловных переходов нет никакого связанного обычного перехода, потому что в таких случаях всегда берется ответвление. Перекрестные ссылки перехода выводят на экран адрес первоначального расположения (источник перехода). Перекрестные ссылки перехода отличают при помощи суффикса *j* (J как *Jump*).

Перекрестные ссылки на данные. Перекрестные ссылки данных используются, чтобы отследить способ получения доступа к данным в пределах двоичного файла. Перекрестные ссылки данных могут быть связаны с любым байтом в базе данных IDA, который связан с виртуальным адресом (другими словами, перекрестные ссылки данных никогда не связываются с переменными стека). Чтобы указать место, откуда считываем, место, куда записываем и адрес расположения объекта, используются три типа перекрестных ссылок данных.

Считывающая перекрестная ссылка используется, чтобы указать, что к содержанию ячейки памяти получают доступ. Считывающие перекрестные

ссылки могут выполниться только из адреса команды, но могут обратиться к любому месту программы. Для обозначения считывающей перекрестной ссылки используется суффикс r.

Перекрестные ссылки записи сгенерированы и выведены на экран как комментарии для переменной, указывая на места программы, которые изменяют содержание переменной. Для обозначения этого типа перекрестных ссылок используют суффикс w.

Перекрестная ссылка смещения, указывает, какой адрес используется. Обычно перекрестные ссылки смещения - результат операций указателя в коде или в данных. Операции доступа к массиву, например, обычно реализуются, добавляя смещение к начальному адресу массива. В результате первый адрес в большинстве глобальных массивов может часто распознаваться присутствием перекрестной ссылки смещения. В отличие от чтения и перекрестных ссылок записи, которые могут работать только из места инструкции, перекрестные ссылки смещения могут выполняться из места инструкции или данных. Примером смещения, которое может выполниться из раздела данных программы, является любая таблица указателей.

Списки перекрестных ссылок. Число комментариев перекрестной ссылки, которые могут быть выведены на экран в данном расположении, ограничено параметром конфигурации, которое принимает значение, по умолчанию равное 2.

Есть два метода для того, чтобы просмотреть полный список перекрестных ссылок на место. Первый метод - открытие окна перекрестных ссылок, связанных с определенным адресом. Наведя курсор на адрес, который является целью одной или более перекрестных ссылок, выбрать View ► Open Subviews ► Cross-References, Вы можете открыть полный список перекрестных ссылок на данную область

Второй способ получить доступ к списку перекрестных ссылок состоит в том, чтобы щелкнуть правой кнопкой по имени, и выбрать Переход к *xref* к операнду (горячая клавиша X), чтобы открыть диалоговое окно, которое перечисляет каждое место, которое ссылается на выбранный символ.

Вызовы функций. Специализированный список перекрестных ссылок, имеющий дело исключительно с вызовами функции, доступен, если выбрать View ► Open Subviews ► Function Calls.

Каждая перечисленная перекрестная ссылка может использоваться к быстро изменяющемуся списку дизассемблирования соответствующего расположения перекрестной ссылки.

8.2. Графы

Ограничиваясь непосредственно определенными типами перекрестных ссылок, мы можем получить много полезных графов для того, чтобы проанализировать наши двоичные файлы.

8.2.1. Построение графа IDA путем наследования

Возможность построения графа наследования IDA доступна в Windows и обеспечена связанным приложением, названным wingraph32. Всякий раз, когда граф наследования требует, источник для графа генерируется и сохраняется во временном файле, когда wingraph32 запущен, чтобы вывести на экран граф. Как только граф был загружен в память, wingraph32 сразу удаляет связанный временный файл графа; однако, Вы можете сохранить выведенный на экран граф, используя wingraph32's File ► Save as. Сгенерированные файлы графа используют Graph Description Language (GDL), чтобы определять графи. Доступные графы наследования включают следующее:

- Функциональная блок-схема.
- Вызов графа для всего двоичного файла.
- Граф перекрестных ссылок на символ.
- Граф перекрестных ссылок от символа.
- Специализированный граф перекрестной ссылки.

Существует много ограничений при контакте с любым графом наследования. Прежде всего, учтем, что графы наследования не являются интерактивными. Манипулирование выведенными на экран графами наследования по существу ограничено изменением масштаба и панорамированием. Невозможно отредактировать граф и управлять контентом дизассемблирования, как Вы привыкли это делать при использовании дизассемблирования IDA или интегрировании представления графа.

Блок-схемы наследования. Когда курсор в пределах функции, нажатие View ► Graphs ► Flow Chart (горячая клавиша F12) генерирует и выводит на экран блок-схему стиля наследования. Эти графы лучше назвать графами потока управления, поскольку они группируют инструкции функции в базовые блоки и используют ребра, чтобы указать, как один блок переходит к другому.

В компьютерной программе **базовый блок** - группировка одной или более инструкций с единственным входом в начало блока и единственным выходом из конца блока. Управление передачей инструкций вызова функции вне текущей функции обычно проигнорировано, если неизвестно, что функция вызывает ошибку, чтобы вернуть результат.

Графы наследования вызовов. Граф вызова функции полезен для быстрого понимания иерархии вызовов функции, сделанных в пределах программы. Графы вызовов генерируются путем создания узлов графа для каждой функции и затем присоединения функциональных узлов, основанных на существовании перекрестной ссылки вызова от одной функции до другой. Процесс генерирования графа вызовов для единственной функции может быть

просмотрен как рекурсивный спуск через все функции, которые вызваны от начальной функции.

Фактически, в случае динамически соединенного двоичного файла невозможно углубляться в библиотечные функции, так как код для таких функций не присутствует в пределах динамически соединенного двоичного файла. Статически соединенные двоичные файлы представляют некоторые проблемы при генерировании графов. Так как статически соединенные двоичные файлы содержат весь код для библиотек, которые были соединены с программой, связанные графы вызова функции могут стать чрезвычайно большими.

После компиляции динамически соединенного файла с использованием GNU gcc, мы можем попросить IDA сгенерировать граф вызова функции, используя View ► Graphs ► Function Calls.

Графы наследования перекрестной ссылки. Два типа графов перекрестной ссылки могут быть сгенерированы для глобальных символов (функции или глобальные переменные): перекрестные ссылки на символ (View ► Graphs ► Xrefs To) и перекрестные ссылки от символа (View ► Graphs ► Xrefs From).

Xrefs To граф может помочь Вам в визуализации всех мест, которые ссылаются на глобальную переменную и цепочку вызовов функций, требуемых для достижения эти места

Пользовательские Графы перекрестных ссылок. Пользовательские графы перекрестной ссылки, названные *User xref charts* в IDA, обеспечивают максимальную гибкость в генерировании графов перекрестной ссылки, чтобы удовлетворить Вашим потребностям. В дополнение к объединению перекрестных ссылок на символ и перекрестных ссылок от символа в единственный граф, пользовательские графы перекрестной ссылки позволяют Вам определять максимальную глубину рекурсии и типы символов, которые должны быть включены или исключены из получающегося графа.

– **Начальное направление** позволяет Вам решать, искать ли перекрестные ссылки от выбранного символа к выбранному символу, или обоим.

– **Параметры.** Опция *Recursive* включает рекурсивный спуск (*xrefs To*) или подъем (*xrefs From*) от выбранных символов. Следование только в текущем направлении вынуждает любую рекурсию пройти только в одну сторону.

– **Глубина рекурсии.** Эта опция устанавливает максимальную глубину рекурсии и полезна для ограничения размера сгенерированных графов..

– **Проигнорировать.** Эта опция диктует, какие типы узлов будут исключены из сгенерированного графа.

– **Опции печати** управляют двумя аспектами форматирования графа. Комментарии печати заставляют любые функциональные комментарии быть включенными в узел графа функции.

Вопросы:

1. Назовите характерное свойство повторяющихся комментариев. Каков механизм реализации этого свойства?
2. Охарактеризуйте операции со структурами в IDA.
3. Перечислите и кратко опишите перекрестные ссылки на код и на данные.

ЗАКЛЮЧЕНИЕ

Дизассемблеры и отладчики могут использоваться не только для нахождения ошибок в программном обеспечении, но также и как эффективный инструмент для реверс-инжиниринга. При анализе вредоносного ПО и обфусцирванного кода возможность использовать одно приложение как для статического, так и динамического анализа может сохранить множество времени и сил требуемых для создания данных при помощи одного инструмента и их анализа другим. В сравнении с другими современными дизассемблерами и отладчиками IDA это прекрасный инструмент для поиска ошибок в ваших приложениях и разработки средств его защиты.

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. Искусство дизасемблирования./Касперски К., Рокко Е. – СПб: БХВ Петербург, 2008. – 212 с.
2. Касперски К. Техника отладки программ без исходных текстов – БХВ-Петербург, 2005. – 156 с.
3. Касперски К. Техника защиты компакт-дисков от копирования. – БХВ-Петербург, 2004. – 194 с.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого 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, мультимедиатехнологии, 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 а 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 заверше-

нию 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

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 100 экз.
Отпечатано на ризографе

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