ОБ АВТОРАХ

Крис Игл занимается обратной разработкой уже 40 лет. Он автор книги «The IDA Pro Book», вышедшей в издательстве No Starch Press, и пользуется большим авторитетом как преподаватель обратной разработки. Его перу принадлежат многочисленные статьи по инструментам обратной разработки, он часто выступает на таких мероприятиях, как Blackhat, Defcon и Shmoocon.

Кара Нэнс – частный консультант по безопасности. В течение многих лет работала профессором информатики. Была членом совета директоров проекта Honeynet и много раз выступала с докладами на различных конференциях по всему миру. Обожает разрабатывать расширения Ghidra и регулярно читает курсы по Ghidra.

О ТЕХНИЧЕСКОМ РЕЦЕНЗЕНТЕ

Брайан Хэй много лет занимался обратной разработкой, был профессором и разработчиком программного обеспечения. Выступал на многих конференциях, читал курсы, а в настоящее время работает старшим научным сотрудником в компании, занимающейся исследованиями в области безопасности. Специализируется на проектировании и разработке виртуализированных сред для обучения и тестирования новых впечатляющих инструментов, таких как Ghidra.

КРАТКОЕ СОДЕРЖАНИЕ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Принимаясь за написание этой книги, мы ставили себе целью познакомить с Ghidra нынешних и будущих специалистов по обратной разработке. В руках опытного инженера Ghidra упрощает процесс анализа и позволяет настраивать и расширять свои возможности под потребности конкретного пользователя, так чтобы они соответствовали привычному ему технологическому процессу. Также Ghidra вполне доступна начинающим инженерам, чему немало способствует включенный в нее декомпилятор, который позволяет лучше понять связи между высокоуровневым языком программирования и листингами дизассеблера человеку, только вступающему в мир анализа двоичного кода.

Писать книгу о Ghidra нелегко. Ghidra – сложный инструмент с открытым исходным кодом, который постоянно развивается. Наш текст описывает движущуюся мишень, поскольку сообщество Ghidra продолжает улучшать и расширять возможности программы. Как и во многих других новых проектах с открытым исходным кодом, рождение Ghidra ознаменовалось серией быстро сменяющих друг друга выпусков. Основная цель авторов состояла в том, чтобы на фоне развития Ghidra все же предложить читателям широкий и глубокий фундамент для понимания и эффективного использования текущей и будущих версий Ghidra в их работе по обратной разработке. Насколько это возможно, мы старались сделать книгу независимой от версии. По счастью, новые

выпуски Ghidra хорошо документированы и содержат подробные списки изменений, которые помогут оценить различия между тем, что написано в книге, и вашей текущей версией.

Об этой книге

Это первая полная книга о Ghidra. Она задумана как всеобъемлющий источник для изучающих обратную разработку с помощью Ghidra. В ней имеется вводный материал, облегчающий начинающим вступление в мир обратной разработки, материал повышенной сложности, который поможет опытным инженерам расширить свое видение мира, а также примеры, которые будут полезны как новобранцам, так и ветеранам, желающим расширить возможности Ghidra и стать членами сообщества разработчиков.

На кого рассчитана эта книга?

Эта книга предназначена для начинающих и опытных специалистов по обратной разработке. Если у вас еще нет опыта в этой области, ничего страшного – в начальных главах достаточно материала, чтобы овладеть основами обратной разработки и приступить к исследованию и анализу двоичного кода с помощью Ghidra. Опытные инженеры, желающие добавить Ghidra в свой арсенал, могут быстро просмотреть первые две части, чтобы получить общее представление о Ghidra, а затем перейти к тем главам, которые им особенно интересны. Опытные пользователи и разработчики Ghidra могут сконцентрироваться на более поздних главах, где описывается, как создавать новые расширения, и применить свои знания и опыт для обогащения проекта Ghidra.

Структура книги

Эта книга разделена на пять частей. Часть I содержит введение в дизассемблирование, обратную разработку и сам проект Ghidra. В части II рассматриваются базовые приемы использования Ghidra. Часть III демонстрирует настройку и автоматизацию Ghidra. В части IV более глубоко объясняются конкретные типы модулей Ghidra и вспомогательные концепции. В части V показано, как применить Ghidra в некоторых реальных ситуациях, с которыми может столкнуться специалист по обратной разработке.

Часть I. Введение

Глава 1. Введение в дизассемблирование

В этой вводной главе мы познакомимся с теорией и практикой дизассемблирования и обсудим некоторые плюсы и минусы двух наиболее распространенных алгоритмов дизассемблирования.

Глава 2. Инструменты дизассемблирования и обратной разработки

В этой главе обсуждаются основные категории инструментов обратной разработки и дизассемблирования.

Глава 3. Первое знакомство с Ghidra

Здесь мы впервые встретимся с Ghidra, узнаем о ее истоках, о том, как ее получить и начать пользоваться этим свободным комплектом инструментов с открытым исходным кодом.

Часть II. Основы использования Ghidra

Глава 4. Начало работы с Ghidra

В этой главе начинается наше путешествие в мир Ghidra. Мы увидим Ghidra в действии, для чего создадим проект, проанализируем файл и познакомимся с графическим интерфейсом Ghidra.

Глава 5. Отображение данных в Ghidra

Здесь мы познакомимся с браузером кода (CodeBrowser), главным аналитическим средством Ghidra, и его основными окнами.

Глава 6. Дизассемблирование в Ghidra

В этой главе мы изучим концепции, необходимые для понимания процесса дизассемблирования в Ghidra.

Глава 7. Управление дизассемблированием

В этой главе мы научимся дополнять анализ Ghidra и управлять процессом дизассемблирования в своих целях.

Глава 8. Типы данных и структуры данных

В этой главе мы научимся распознавать и манипулировать простыми и сложными структурами данных, встречающимися в компилированных программах.

Глава 9. Перекрестные ссылки

Эта глава посвящена подробному рассмотрению перекрестных ссылок, их графическому представлению и той важной роли, которую они играют в понимании поведения программы.

Глава 10. Графы

В этой главе мы познакомимся с графическими возможностями Ghidra и графами как средством анализа двоичного кода.

Часть III. Поставить Ghidra себе на службу

Глава 11. Коллективная обратная разработка программ

В этой главе представлена уникальная возможность Ghidra – использование в качестве инструмента коллективной работы. Мы узнаем, как сконфигурировать сервер Ghidra и сделать проект доступным другим аналитикам.

Глава 12. Настройка Ghidra

Здесь мы начнем настраивать Ghidra, конфигурируя проекты и инструменты, так чтобы они отвечали нашему технологическому процессу анализа.

Глава 13. Расширение взгляда на мир Ghidra

В этой главе мы научимся генерировать и применять сигнатуры библиотек и другого специального содержимого, чтобы Ghidra могла распознавать новые конструкции в двоичном коде.

Глава 14. Основы написания скриптов для Ghidra

В этой главе мы познакомимся с основами написания скриптов для Ghidra на Python и Java с применением встроенного редактора Ghidra.

Глава 15. Eclipse и GhidraDev

В этой главе мы поднимем написание скриптов на новый уровень, интегрировав Eclipse с Ghidra и воспользовавшись мощными скриптовыми возможностями, предоставляемыми этой конфигурацией, в частности приведем рабочий пример построения нового анализатора.

Глава 16. Необслуживаемый режим Ghidra

Здесь мы познакомимся с использованием Ghidra в необслуживаемом режиме, когда не требуется никакого GUI. Вы, без сомнения, оцените преимущества этого режима в типичных крупномасштабных повторяющихся задачах.

Часть IV. Дополнительные темы

Глава 17. Загрузчики Ghidra

Здесь мы более глубоко познакомимся с тем, как Ghidra импортирует и загружает файлы. У нас будет возможность создать новые загрузчики для обработки ранее не распознаваемых типов файлов.

Глава 18. Процессоры Ghidra

В этой главе мы рассмотрим язык Ghidra SLEIGH, предназначенный для определения архитектуры процессора. Мы изучим процесс добавления новых процессоров и команд в Ghidra.

Глава 19. Декомпилятор Ghidra

Здесь мы подробнее рассмотрим одну из самых популярных возможностей Ghidra: декомпилятор. Вы увидите, как он работает на внутреннем уровне и какой вклад вносит в процесс анализа.

Глава 20. Зависимость от компилятора

Эта глава посвящена вариациям кода в зависимости от компилятора и целевой платформы.

Часть V. Реальные приложения

Глава 21. Анализ обфусцированного кода

Мы узнаем, как использовать Ghidra для анализа обфусцированного кода в статическом контексте, так чтобы код не нужно было исполнять.

Глава 22. Изменение двоичного кода

В этой главе мы узнаем о некоторых способах использования Ghidra для изменения двоичного кода в процессе анализа – как внутри самой Ghidra, так и для создания «залатанных» версий оригинальных двоичных файлов.

Глава 23. Определение разности двоичных файлов и отслеживание версий

В этой, последней, главе приводится обзор средств Ghidra, позволяющих вычислить дельту между двумя двоичными файлами, а также краткое введение в дополнительные средства отслеживания версий.

Приложение. Ghidra для пользователей IDA

Опытные пользователи IDA найдут в этом приложении информацию о соответствии между терминологией и сходной функциональностью IDA и Ghidra.

ПРИМЕЧАНИЕ

Код, встречающийся в листингах, можно найти на сайте https://nostarch.com/GhidraBook/ и https://ghidrabook.com/.

Часть I
Введение

1

ВВЕДЕНИЕ В ДИЗАССЕМБЛИРОВАНИЕ

Вам, наверное, интересно, чего ожидать от книги о Ghidra. Конечно, эта книга целиком посвящена Ghidra, но она не задумывалась как «Руководство пользователя Ghidra». Мы хотели использовать Ghidra как инструмент для обсуждения методов обратной разработки, полезных при анализе широкого круга программ – от уязвимых приложений до вредоносного программного обеспечения. Там, где это оправдано, мы будем подробно описывать шаги применения Ghidra для выполнения конкретных действий в данной ситуации. В результате получилась обзорная экскурсия по всем возможностям Ghidra, начиная с самых простых задач, решаемых в ходе начального исследования файла, и заканчивая настройкой Ghidra для более сложных задач обратной разработки. Мы не ставили себе целью рассмотреть весь потенциал Ghidra. Но средства, наиболее полезные для обратной разработки, мы включили.

Эта книга поможет вам сделать Ghidra самым эффективным орудием в своем арсенале.

Прежде чем с головой погрузиться в специфику Ghidra, поговорим об основах процесса дизассемблирования и других средствах обратной разработки откомпилированного кода. Хотя эти средства необязательно покрывают весь спектр возможностей Ghidra, каждое из них соответствует какой-то части функциональности Ghidra и позволяет лучше понять ее конкретные особенности. Далее в этой главе процесс дизассемблирования описывается на верхнем уровне.

ТЕОРИЯ ДИЗАССЕМБЛИРОВАНИЯ

Каждый, кто хоть немного изучал языки программирования, вероятно, знает о различных поколениях языков, но для тех, кто все это время проспал, дадим краткую справку.

Языки первого поколения. Это самая низшая форма языков, программируют на них, записывая команды в двоичном или шестнадцатеричном виде, а прочитать такой код могут немногие умельцы. На этом уровне трудно отличить данные от команд, потому что выглядят они одинаково. Языки первого поколения называют еще *машинными языками*, или байт-кодом, а написанные на них программы – *двоичными*.

Языки второго поколения. Их еще называют *языками ассемблера*, от машинных языков их отделяет только простой поиск в таблице, позволяющий сопоставить комбинациям битов, или кодам операций (опкодам), короткие, но легко запоминающиеся последовательности символов – *мнемонические коды*, облегчающие программистам запоминание команд. *Ассемблером* называется программа, которая транслирует код на языке ассемблера в машинный код, пригодный для выполнения. Помимо мнемонических кодов команд, полный язык ассемблера обычно включает *директивы*, указывающие ассемблеру, как размещать код и данные в памяти.

Языки третьего поколения. Это следующий шаг в направлении выразительности естественных языков – вводятся ключевые слова и конструкции, используемые в качестве строительных блоков, из которых создаются программы.

Языки третьего поколения обычно не зависят от платформы, хотя написанные на них программы могут быть платформенно зависимыми из-за использования особенностей конкретной операционной системы. Говоря о языках третьего поколения, чаще всего вспоминают FORTRAN, C и Java. Для трансляции программы на язык ассемблера или прямо на машинный язык (или его приблизительный эквивалент, например байт-код) обычно используются компиляторы.

Языки четвертого поколения. Такие существуют, но к теме этой книги не имеют отношения и потому не рассматриваются.

ЧТО ДЕЛАЕТ ДИЗАССЕМБЛЕР

В традиционной модели разработки ПО компиляторы, ассемблеры и компоновщики используются по отдельности либо совместно для создания исполняемых программ. Чтобы проделать обратный путь (т. е. подвергнуть программу обратной разработке), мы применяем инструменты, обращающие процессы ассемблирования и компиляции. Неудивительно, что они называются *дизассемблерами* и *декомпиляторами*. Дизассемблер обращает процесс ассемблирования, так что на выходе мы ожидаем получить код на языке ассемблера (а на вход подаем код на машинном языке). Задача декомпилятора – восстановить код на языке высокого уровня, получив на входе код на языке ассемблера или даже на машинном языке.

Обещание «восстановить исходный код» звучит очень заманчиво на конкурентном рынке программного обеспечения, поэтому разработка хороших декомпиляторов остается областью активных исследований в информатике. Ниже перечислено лишь несколько из множества причин, затрудняющих декомпиляцию.

- В процессе компиляции теряется часть информации. На уровне машинного языка не существует имен переменных и функций, а тип информации можно определить только по характеру использования данных, поскольку явных объявлений типов нет. Видя, что копируется 32 бита данных, мы должны проделать исследовательскую работу, чтобы установить, что это такое: 32-разрядное це-

лое число, 32-разрядное число с плавающей точкой или 32-разрядный указатель.

- **Компиляция – это операция вида «многие ко многим».** Это означает, что исходную программу можно транслировать на язык ассемблера разными способами, а коду на машинном языке могут соответствовать разные конструкции на исходном коде. Поэтому декомпиляция только что откомпилированного файла может дать исходный файл, сильно отличающийся от оригинала.
- **Декомпиляторы зависят от языка и библиотек.** Обработка двоичного файла, порожденного компилятором Delphi, с помощью декомпилятора, рассчитанного на генерирование C-кода, может привести к очень странным результатам. Аналогично прогон двоичного файла для Windows через декомпилятор, который ничего не знает о Windows API, вряд ли даст что-то полезное.
- **Для точной декомпиляции двоичного файла необходим почти идеальный дизассемблер.** Любая ошибка или пропуск на этапе дизассемблирования почти наверняка отразится на декомпилированном коде. Правильность дизассемблированного кода можно проверить, прибегнув к справочным руководствам по соответствующему процессору, но для проверки правильности результата декомпиляции нет никаких канонических справочных руководств.

В Ghidra встроен декомпилятор, который мы будем изучать в главе 19.

ЗАЧЕМ НУЖЕН ДИЗАССЕМБЛЕР

Цель инструментов дизассемблирования часто состоит в том, чтобы разобраться в программе, исходный код которой недоступен. Перечислим типичные ситуации:

- анализ вредоносного ПО;
- анализ программ с закрытым исходным кодом на уязвимость;
- анализ интероперабельности программ с закрытым исходным кодом;

- анализ сгенерированного компилятором кода с целью проверить его производительность или правильность;
- отображение команд программы в процессе отладки.

Далее мы рассмотрим эти ситуации более подробно.

Анализ вредоносного ПО

Авторы вредоносного ПО, если только это не скрипты, редко оказывают нам любезность, предоставляя исходный код своих творений. А в отсутствие исходного кода наши возможности понять, как ведет себя вредонос, крайне ограничены. Есть два основных вида анализа: динамический и статический. *Динамический анализ* подразумевает выполнение вредоносного кода в тщательно контролируемом окружении (песочнице), когда за всеми аспектами поведения наблюдают с помощью различных инструментальных утилит. Напротим, *статический анализ* – это попытка понять, что делает программа, читая ее код, который в случае вредоносного ПО чаще всего состоит только из листинга дизассемблера и, возможно, листинга декомпилятора.

Анализ на уязвимость

Для простоты разобьем весь процесс аудита безопасности на три этапа: обнаружение уязвимости, анализ уязвимости и разработка эксплойта. Одни и те же шаги выполняются вне зависимости от того, есть исходный код или нет, однако объем усилий резко возрастает, если имеется только двоичный код. Первый этап – найти место в программе, потенциально допускающее эксплуатацию. Для этого часто применяются динамические методы, например фаззинг[1], но то же самое можно сделать (обычно с гораздо большими усилиями) с помощью статического анализа. После того как проблема выявлена, часто требуется дальнейший анализ, чтобы понять, допускает или она эксплуатацию и если да, то при каких условиях.

[1] *Фаззинг* (букв. опыление) – это метод обнаружения уязвимостей, который заключается в генерировании большого объема случайных входных данных для программы в надежде, что какие-то приведут к отказу, который можно будет обнаружить, проанализировать и в конечном итоге эксплуатировать.

Нахождение переменных, которыми атакующий может с пользой для себя манипулировать, – важный ранний шаг процесса обнаружения. Листинги дизассемблера дают достаточный уровень детализации, чтобы точно понять, как компилятор решил размещать переменные в памяти. Например, бывает полезно знать, что 70-байтовый массив символов, объявленный программистом, компилятор округлил до 80 байтов. Кроме того, листинги дизассемблера – единственный способ точно выяснить, как компилятор решил упорядочить переменные, объявленные глобально или внутри функций. Знание пространственных отношений между переменными зачастую необходимо для разработки эксплойта. Так, совместно используя дизассемблер и отладчик, удается создать эксплойт.

Анализ интероперабельности

Если программа выпускается только в двоичной форме, конкурентам очень трудно создать программы, которые могут работать совместно с ней, или предложить подставляемые вместо нее программы, совместимые по интерфейсу. Типичный пример – код драйвера для оборудования, поддерживаемый только на одной платформе. Если производитель не торопится с его поддержкой или, хуже того, вообще отказывается поддерживать свое оборудование на других платформах, то необходима обратная разработка, чтобы написать драйверы. В таких случаях статический анализ кода – едва ли не единственное средство, и зачастую, чтобы разобраться в прошивке, приходится выходить за рамки программного драйвера.

Проверка компилятора

Поскольку цель компилятора (или ассемблера) – генерирование машинного кода, часто необходимы хорошие инструменты дизассемблирования, которые проверят, действует ли компилятор в соответствии с проектными спецификациями. Аналитикам также интересно найти дополнительные возможности для оптимизации выхода компилятора. Да и с точки зрения безопасности хорошо бы убедиться, что сам компилятор невозможно скомпрометировать, так что он будет оставлять закладки в сгенерированном коде.

Отображение команд в процессе отладки

Пожалуй, самое распространенное применение дизассемблеров – генерирование листингов в отладчиках. К сожалению, дизассемблерам, встроенным в отладчики, не хватает изощренности. В общем случае они не умеют дизассемблировать пакетно и иногда отказываются работать, если не могут определить границы функции. Это одна из причин, по которым лучше использовать отладчик в связке с высококачественным дизассемблером, чтобы лучше оценить контекст в процессе отладки.

КАК РАБОТАЕТ ДИЗАССЕМБЛЕР

Разобравшись с целями дизассемблирования, самое время заняться вопросом о том, как в действительности работает дизассемблер. Рассмотрим типичную внушающую страх задачу, стоящую перед дизассемблером: *взять предложенные 100 КБ, отличить код от данных, перевести код на язык ассемблера для представления пользователю и не наделать по дороге ошибок*. Можно было бы пристегнуть сюда кучу специальных запросов, например попросить дизассемблер найти функции, распознать таблицы переходов или выявить локальные переменные – все это сильно затрудняет его работу.

Чтобы учесть все наши требования, дизассемблер должен сделать выбор из множества алгоритмов обработки передаваемых ему файлов. Качество сгенерированных листингов напрямую зависит от качества используемых алгоритмов и их реализации.

В этом разделе мы обсудим два главных алгоритма, применяемых в настоящее время для дизассемблирования машинного кода. По ходу обсуждения мы будем отмечать их недостатки, чтобы вы были готовы к ситуациям, когда кажется, что дизассемблер не работает. Понимая ограничения дизассемблера, вы сможете вмешаться вручную и повысить качество результата.

Базовый алгоритм дизассемблирования

Для начала разработаем простой алгоритм, который на входе принимает код на машинном языке, а на выходе порождает код на языке ассемблера. По ходу дела вы получите представ-

ление о проблемах, предположениях и компромиссах, сопровождающих процесс автоматического дизассемблирования.

1. Первый шаг – найти область кода, подлежащую дизассемблированию. Далеко не всегда это так просто, как может показаться. Часто команды перемешаны с данными, и важно отличать одного от другого. В самом типичном случае, когда дизассемблируется исполняемый файл, известен формат этого файла, например *Portable Executable* (*PE*) в Windows или *Executable and Linkable Format* (*ELF*) во многих Unix-системах. В этих форматах обычно имеются механизмы (часто в виде иерархических заголовков файла) нахождения секций файла, содержащих код, а также точек входа в этот код[1].
2. Зная адрес команды, мы должны прочитать значение или значения, находящиеся по этому адресу (или смещению от начала файла), и найти в таблице мнемоническую команду ассемблера, соответствующую данному коду операции. В зависимости от сложности системы команд процессора этот процесс может быть нетривиальным и, возможно, включает ряд дополнительных операций, например интерпретацию префиксов, модифицирующих поведение команды, и определение того, какие операнды необходимы команде. Если команды имеют переменную длину, как в системе команд Intel x86, то не исключено, что для полного дизассемблирования одной команды придется прочитать дополнительные байты.
3. После того как команда прочитана из файла и все ее операнды декодированы, форматируется эквивалентная ей команда на языке ассемблера, и результат записывается в листинг дизассемблера. Бывает так, что есть выбор из нескольких вариантов синтаксиса языка ассемблера. Например, для ассемблера x86 есть два основных формата: Intel и AT&T.
4. Завершив вывод команды, мы должны продвинуться к началу следующей и повторить весь процесс, пока не будут дизассемблированы все команды в файле.

[1] *Точка входа в программу* – это просто адрес команды, которой операционная система передает управление после загрузки программы в память.

Синтаксис ассемблера X86: AT&T и INTEL

Есть два основных варианта синтаксиса кода на языке ассемблера: AT&T и Intel. Хотя тот и другой – языки второго поколения, их синтаксис существенно различается – от записи переменных, констант и доступа к регистрам до переопределения размера сегментов и команд, описания косвенности и задания смещений. Синтаксис AT&T выделяется использованием префикса % в именах всех регистров, префикса `$` в именах литералов (так называемых *непосредственных операндов*) и порядком операндов: исходный операнд находится слева, а конечный справа. В синтаксисе AT&T команда прибавления 4 к регистру `EAX` имеет вид `add $0x4,%eax`. В ассемблере GNU (`as`) и многих других инструментах GNU, включая `gcc` и `gdb`, по умолчанию используется синтаксис AT&T.

Синтаксис Intel отличается отсутствием префиксов у литералов и регистров и противоположным порядком записи операндов: исходный справа, конечный слева. Та же команда сложения в синтаксисе Intel имеет вид `add eax,0x4`. Из ассемблеров, в которых используется синтаксис Intel, отметим Microsoft Assembler (MASM) и Netwide Assembler (NASM).

Существуют различные алгоритмы определения места, с которого начинать дизассемблирование, выбора следующей команды, различения кода и данных и определения того факта, что все команды дизассемблированы. Два основных: линейная развертка и рекурсивный спуск.

Алгоритм линейной развертки

В случае алгоритма *линейной развертки* дизассемблер применяет прямолинейный подход к нахождению команд, подлежащих дизассемблированию: там, где одна команда кончается, начинается другая. Поэтому самые трудные решения – откуда начать и когда остановиться. Чаще всего предполагается, что всё находящееся в секциях программы, помеченных как код (обычно это указывается в заголовках исполняемого файла), – команды машинного языка. Дизассемблирование начинается с первого байта в секции кода и продвигается вперед линейно, пока не будет достигнут конец секции. Не предпринимается

никаких попыток учесть поток выполнения, распознавая команды нелинейного перехода, например ветвления.

В процессе дизассемблирования может поддерживаться указатель, отмечающий начало текущей команды. По ходу дела вычисляется длина каждой команды, и это знание используется для определения адреса начала следующей команды. Для систем команд с фиксированной длиной (например, MIPS) дизассемблирование немного проще, поскольку для разграничения команд не нужно прилагать усилий.

Главное преимущество алгоритма линейной развертки – то, что он полностью покрывает все имеющиеся в программе секции кода. А его главный недостаток – то, что он не учитывает возможность смешивания кода и данных. Это хорошо видно в листинге 1.1, где показан результат обработки функции дизассемблером, основанным на алгоритме 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
   401250: ff 24 95 57 12 40 00 jmp DWORD PTR [edx*4+0x401257]❶
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
```

Листинг 1.1. Дизассемблирование методом линейной развертки

Эта функция содержит предложение switch, и компилятор решил реализовать его с помощью таблицы переходов, в которой хранятся адреса меток case. Кроме того, компилятор решил разместить таблицу переходов в самой функции. Команда `jmp` ❶ ссылается на таблицу адресов ❷. К сожалению, дизассемблер рассматривает таблицу адресов как последовательность команд и неправильно генерирует следующий за ней код на языке ассемблера.

Если рассматривать 4-байтовые группы в таблице переходов ❷ как значения, записанные в прямом порядке байтов[1], то мы увидим, что каждая представляет собой указатель на близлежащий адрес, т. е. конечные адреса команд переходов (`004012e0, 0040128b, 00401290,` …). Таким образом, команда `loopne` ❷ – и не команда вовсе, а свидетельство того, что алгоритм линейной развертки не способен отличить встроенные данные от кода.

Алгоритм линейной развертки используется в движках дизассемблирования, входящих в состав отладчика GNU (gdb), отладчика Microsoft WinDbg и в утилите `objdump`.

Алгоритм рекурсивного спуска

В случае алгоритма *рекурсивного спуска* дизассемблер применяет другой подход к выделению команд: он ориентируется на поток управления и считает, что команду нужно дизассемблировать, если на нее ссылается какая-то другая команда. Чтобы понять, как устроен рекурсивный спуск, полезно классифицировать команды по их воздействию на указатель команд.

Последовательные команды

Последовательная команда передает управление команде, непосредственно следующей за ней. Примерами могут служить простые арифметические команды, в частности `add`; команды копирования из регистра в память (`mov`); команды манипулирования стеком (`push` и `pop`). Для таких команд дизассемблер ведет себя так же, как в случае алгоритма линейной развертки.

[1] В архитектуре x86 применяется прямой порядок байтов, т. е. младший байт многобайтового значения хранится первым – по меньшему адресу, чем последующие. В случае обратного порядка по меньшему адресу хранится старший байт. Процессоры можно классифицировать по порядку байтов, но в некоторых случаях применяются оба.

Команды условного перехода

Команда условного перехода, например `jnz` в системе команд x86, может продолжить выполнение по одному из двух путей. Если условие истинно, то производится переход и указатель команд следует изменить, отразив конечную точку перехода. Если же условие ложно, то выполнение продолжается линейно, так что для дизассемблирования следующей команды можно использовать метод линейной развертки. Поскольку в статическом контексте невозможно определить результат вычисления условия, алгоритм рекурсивного спуска дизассемблирует оба пути, но откладывает дизассемблирование целевой ветви на потом, помещая адрес целевой команды в список адресов, подлежащих дизассемблированию.

Команды безусловного перехода

Команда безусловного перехода не согласуется с моделью линейного выполнения, поэтому обрабатывается алгоритмом рекурсивного спуска по-другому. Как и в случае последовательных команд, выполнение может пойти только по одному пути, но следующая выполняемая команда необязательно находится сразу после команды перехода. Как видно из листинга 1.1, такого требования нет и в помине. Поэтому нет никаких причин дизассемблировать байты, следующие за командой безусловного перехода.

Дизассемблер пытается определить конечный адрес безусловного перехода и продолжает дизассемблирование с этого адреса. К сожалению, некоторые безусловные переходы вызывают трудности. Если конечный адрес команды перехода зависит от значения, вычисляемого во время выполнения, то определить его с помощью статического анализа невозможно. Иллюстрацией может служить команда x86 `jmp rax`. Регистр `rax` содержит разумное значение, только когда программа работает. А во время статического анализа в этом регистре ничего полезного нет, поэтому мы не можем определить адрес перехода и не знаем, с какого места продолжать дизассемблирование.

Команды вызова функций

Команда вызова функции похожа на команду безусловного перехода (и дизассемблер точно так же не может определить целевой адрес в команде типа `call rax`), отличие только в том, что после выполнения функции управление обычно возвращается команде, непосредственно следующей за командой вызова. В этом отношении команда вызова похожа на команду условного перехода, т. к. порождается два пути выполнения. Целевой адрес команды вызова добавляется в список адресов для отложенного дизассемблирования, а следующая за ней команда дизассемблируется сразу, как в алгоритме линейной развертки.

Метод рекурсивного спуска может давать сбои, если программа ведет себя не так, как ожидается, при возврате из функции. Например, функция может намеренно изменить адрес возврата, так что после завершения управление возвращается совсем не туда, куда ожидает дизассемблер. Простой пример приведен в следующем листинге, где некорректно написанная функция `badfunc` прибавляет 1 к адресу возврата, перед тем как вернуть управление вызывающей стороне.

```
badfunc proc near
48 FF 04 24      inc qword ptr [rsp] ; увеличивает сохраненный адрес
возврата на 1
C3                 retn
badfunc endp
; ------------------------------------
label:
E8 F6 FF FF FF   call badfunc
05 48 89 45 F8   add eax, F8458948h❶
```

В результате управление не попадает на команду `add` ❶, следующую за вызовом `badfunc`. Ниже показано, что должен был бы сделать дизассемблер.

```
badfunc proc near
48 FF 04 24    inc qword ptr [rsp]
C3               retn
badfunc endp
; -----------------------------------
label:
E8 F6 FF FF FF call badfunc
05               db 5 ;раньше это был первый байт команды add
48 89 45 F8    mov [rbp-8], rax❶
```

В этом листинге лучше виден поток управления в случае, когда функция `badfunc` возвращает управление команде `mov` ❶. Важно понимать, что метод линейной развертки тоже не сможет правильно дизассемблировать этот код, хотя и по другой причине.

Команды возврата

В некоторых ситуациях у алгоритма рекурсивного спуска не оказывается путей, по которым можно следовать. *Команда возврата из функции* (например, `ret` в случае x86) не содержит никакой информации о том, какая команда будет выполняться следующей. Если бы программа работала, то адрес возврата был бы извлечен из стека времени выполнения, и выполнение продолжилось бы с этого адреса. Но у дизассемблера нет доступа к стеку, поэтому он просто «упирается в стенку». Именно в этой точке алгоритм рекурсивного спуска обращается к списку адресов для отложенного дизассемблирования. Адрес выбирается из списка, и дизассемблирование продолжается с этого адреса. Этот рекурсивный процесс и дал название алгоритму.

Одно из главных достоинств алгоритма рекурсивного спуска – его непревзойденная способность отличать код от данных. Поскольку он основан на анализе потока управления, шансов неправильно дизассемблировать данные как код гораздо меньше. А его основной недостаток – неспособность следовать по косвенным путям в коде, как, например, бывает, когда в командах перехода или вызова используются таблицы указателей, в которых хранятся конечные адреса. Однако благодаря

добавлению некоторых эвристик для отличения указателей от кода дизассемблеры на основе рекурсивного спуска могут обеспечить весьма полное покрытие кода и прекрасно различают код и данные. В листинге 1.2 показан результат работы дизассемблера, встроенного в Ghidra, над тем же предложением switch, что в листинге 1.1.

```
0040123f   PUSH  EBP
00401240   MOV EBP,ESP
00401242   XOR EAX,EAX
00401244   MOV EDX,dword ptr [EBP + param_1]
00401247   CMP EDX,0xc
0040124a   JA  switchD_00401250::caseD_0
         switchD_00401250::switchD
00401250   JMP dword ptr [EDX*0x4 + ->switchD_00401250::caseD_0] =
004012e0
         switchD_00401250::switchdataD_00401257
00401257   addr  switchD_00401250::caseD_0
0040125b   addr  switchD_00401250::caseD_1
0040125f   addr  switchD_00401250::caseD_2
00401263   addr  switchD_00401250::caseD_3
00401267   addr  switchD_00401250::caseD_4
0040126b   addr  switchD_00401250::caseD_5
0040126f   addr  switchD_00401250::caseD_6
00401273   addr  switchD_00401250::caseD_7
00401277   addr  switchD_00401250::caseD_8
0040127b   addr  switchD_00401250::caseD_9
0040127f   addr  switchD_00401250::caseD_a
00401283   addr  switchD_00401250::caseD_b
00401287   addr  switchD_00401250::caseD_c
         switchD_00401250::caseD_1
0040128b   MOV EAX,dword ptr [EBP + param_2]
0040128e   JMP switchD_00401250::caseD_00040128E
```

Рис. 1.2. Результат дизассемблирования методом рекурсивного спуска

Заметим, что эта секция двоичного кода была распознана как предложение switch и соответственно отформатирована. Понимание процесса рекурсивного спуска поможет нам выявить ситуации, когда дизассемблер Ghidra ведет себя неоптимально, и выработать стратегии улучшения результата.

РЕЗЮМЕ

Важно ли глубоко понимать алгоритмы дизассемблирования при использовании дизассемблера? Нет. Полезно ли это? Да! Бороться со своими инструментами – последнее дело, на которое стоит тратить время, занимаясь обратной разработкой. Одно из многих преимуществ Ghidra заключается в том, что ее дизассемблер интерактивный, он предоставляет массу возможностей для управления процессом и отмены своих решений. Поэтому очень часто мы получаем полный и точный листинг.

В следующей главе мы рассмотрим ряд инструментов, доказавших свою полезность во многих возникающих при обратной разработке ситуациях. Хотя они и не связаны напрямую с Ghidra, многие из них оказали на Ghidra влияние, и знакомство с ними поможет лучше понять различные окна в пользовательском интерфейсе Ghidra.