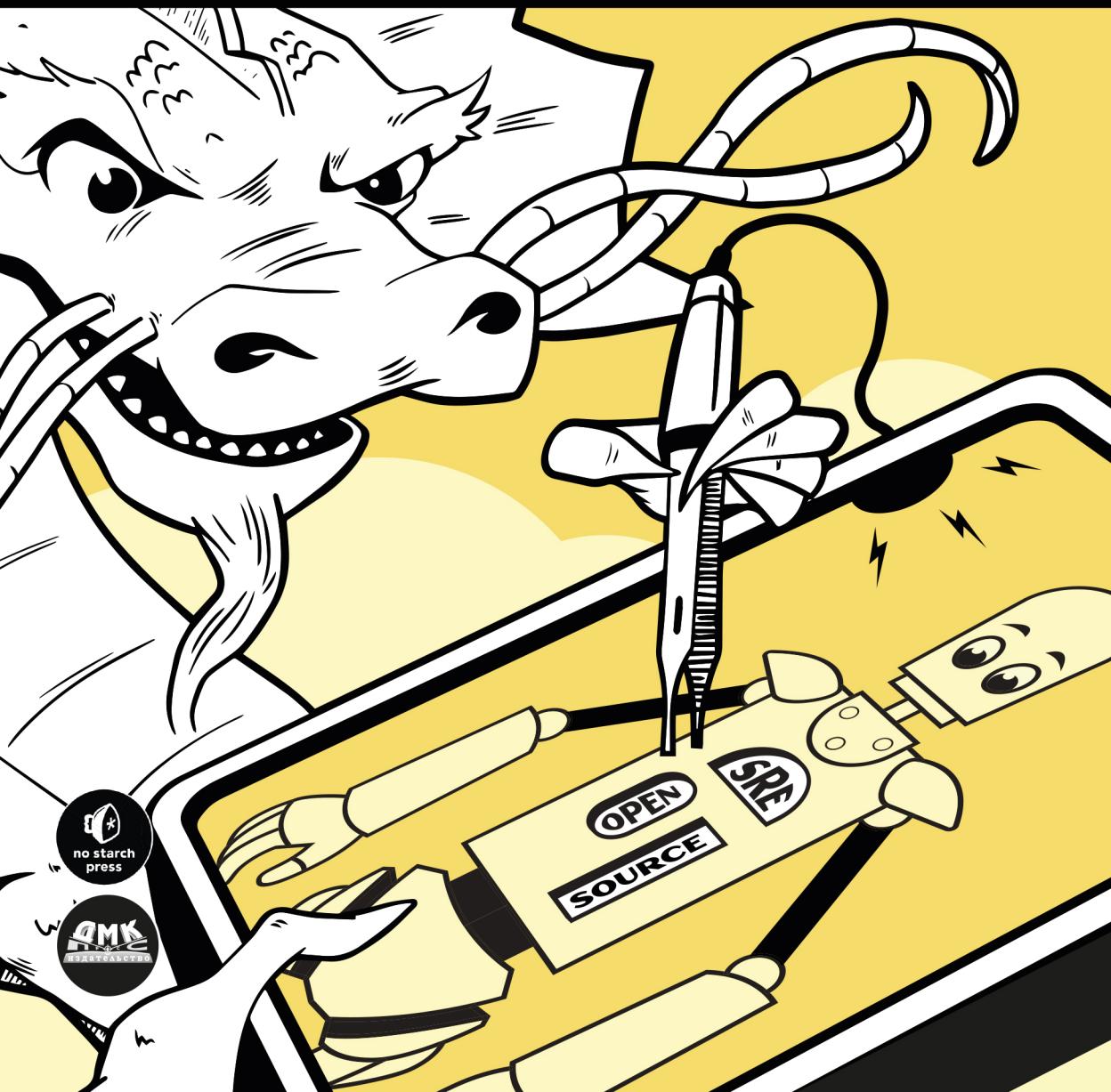


# GHIDRA

# ПОЛНОЕ РУКОВОДСТВО

Крис Игл, Кара Нэнс



Крис Игл, Кара Нэнс

# GHIDRA

Полное руководство

# THE GHIDRA BOOK

The Definitive Guide

by Chris Eagle and Kara Nance



San Francisco

# GHIDRA

Полное руководство

Крис Игл, Кара Нэнс



Москва, 2022

**УДК 004.4**

**ББК 32.97**

**И26**

**Игл К., Нэнс К.**

**И26** GHIDRA. Полное руководство / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. – 750 с.: ил.

**ISBN 978-5-97060-942-2**

Платформа Ghidra, ставшая итогом более десяти лет работы в Агентстве национальной безопасности, была разработана для решения наиболее трудных задач обратной разработки (Reverse Engineering – RE). После раскрытия исходного кода этого инструмента, ранее предназначавшегося только для служебного пользования, один из лучших в мире дизассемблеров и интуитивно понятных декомпиляторов оказался в руках всех специалистов, стоящих на страже кибербезопасности.

Эта книга, рассчитанная равно на начинающих и опытных пользователей, поможет вам во всеоружии встретить задачу RE и анализировать файлы, как это делают профессионалы.

**УДК 004.4**

**ББК 32.97**

Title of English-language original: The Ghidra Book: The Definitive Guide, ISBN 9781718501027, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Russian-Language 1st edition Copyright © 2020 by DMK Press Publishing under license by No Starch Press Inc. All rights reserved.”

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-71850-102-7 (англ.)

ISBN 978-5-97060-942-2 (рус.)

© 2020 Chris Eagle and Kara Nance

© Оформление, издание, перевод,  
ДМК Пресс, 2022

Всем, кто верит в науку и принятие решений на основе фактов, а также первым, кто ответил на угрозу COVID-19 и чья самоотверженность и тяжкий труд стали лучиком надежды в эпоху глобального кризиса.

Всем девушкам, которые питают страсть к технологиям, а также взрослым мужчинам и женщинам, оказывающим им помощь и поддержку.

Мечтайте по-крупному и исследуйте дальше!

## **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**

<b>Об авторах.....</b>	<b>6</b>
<b>О техническом рецензенте.....</b>	<b>6</b>
<b>Оглавление .....</b>	<b>8</b>
<b>Благодарности.....</b>	<b>17</b>
<b>Введение .....</b>	<b>18</b>
<b>Часть I. Введение .....</b>	<b>25</b>
Глава 1. Введение в дизассемблирование.....	27
Глава 2. Обратная разработка и инструменты дизассемблирования .....	43
Глава 3. Первое знакомство с Ghidra .....	65
<b>Часть II. Основы использования Ghidra .....</b>	<b>73</b>
Глава 4. Начало работы с Ghidra .....	75
Глава 5. Отображение данных в Ghidra .....	93
Глава 6. Дизассемблирование в Ghidra.....	135
Глава 7. Управление дизассемблированием.....	175
Глава 8. Типы данных и структуры данных .....	211
Глава 9. Перекрестные ссылки .....	259
Глава 10. Графы .....	277
<b>Часть III. Поставить Ghidra себе на службу .....</b>	<b>299</b>
Глава 11. Коллективная обратная разработка программ .....	301
Глава 12. Настройка Ghidra.....	331
Глава 13. Расширение взгляда на мир Ghidra .....	355
Глава 14. Основы написания скриптов для Ghidra .....	387
Глава 15. Eclipse и GhidraDev.....	423
Глава 16. Необслуживаемый режим Ghidra .....	457
<b>Часть IV. Дополнительные темы .....</b>	<b>483</b>
Глава 17. Загрузчики Ghidra .....	485
Глава 18. Процессорные модули в Ghidra.....	537
Глава 19. Декомпилятор Ghidra.....	571
Глава 20. Зависимость от компилятора.....	591
<b>Часть V. Реальные приложения.....</b>	<b>623</b>
Глава 21. Анализ обfuscированного кода .....	625
Глава 22. Изменение двоичного кода .....	673
Глава 23. Определение разности двоичных файлов и отслеживание версий.....	705
Приложение. Ghidra для пользователей IDA .....	731

# ОГЛАВЛЕНИЕ

## ЧАСТЬ I. ВВЕДЕНИЕ..... 25

<b>Глава 1. Введение в дизассемблирование.....</b>	<b>27</b>
Теория дизассемблирования.....	28
Что делает дизассемблер .....	29
Зачем нужен дизассемблер .....	30
Анализ вредоносного ПО.....	31
Анализ на уязвимость.....	31
Анализ интероперабельности .....	32
Проверка компилятора .....	32
Отображение команд в процессе отладки .....	33
Как работает дизассемблер .....	33
Базовый алгоритм дизассемблирования.....	33
Алгоритм линейной развертки .....	35
Алгоритм рекурсивного спуска .....	37
Резюме.....	42

## Глава 2. Обратная разработка и инструменты

### дизассемблирования..... 43

Средства классификации .....	44
file .....	44
PE Tools.....	47
PEiD .....	48
Обзорные инструменты.....	49
nm .....	49
ldd .....	52
objdump .....	55
otool .....	56
dumpbin.....	56
c++filt .....	57
Инструменты глубокой инспекции.....	59
strings.....	59
Дизассемблеры .....	61
Резюме.....	63

## Глава 3. Первое знакомство с Ghidra..... 65

Лицензионная политика Ghidra .....	66
Версии Ghidra .....	66
Ресурсы поддержки Ghidra .....	66
Скачивание Ghidra .....	68
Установка Ghidra.....	68
Запуск Ghidra .....	70
Резюме.....	71

## **ЧАСТЬ II. ОСНОВЫ ИСПОЛЬЗОВАНИЯ GHIDRA..... 73**

<b>Глава 4. Начало работы с Ghidra .....</b>	<b>75</b>
Запуск Ghidra .....	75
Создание нового проекта .....	77
Загрузка файла в Ghidra.....	78
Использование простого двоичного загрузчика .....	82
Анализ файлов в Ghidra .....	84
Результаты автоматического анализа .....	88
Поведение рабочего стола во время начального анализа .....	89
Сохранение работы и выход.....	90
Советы по организации рабочего стола Ghidra .....	91
Резюме.....	92
<b>Глава 5. Отображение данных в Ghidra.....</b>	<b>93</b>
Браузер кода .....	94
Окна браузера кода .....	97
Окно листинга.....	100
Создание дополнительных окон дизассемблера .....	105
Представление графа функции в Ghidra .....	106
Окно деревьев программы.....	112
Окно дерева символов.....	113
Импортируемые объекты.....	114
Экспортируемые объекты .....	115
Функции .....	115
Метки .....	116
Классы.....	116
Пространства имен.....	117
Окно диспетчера типов данных.....	117
Окно консоли.....	118
Окно декомпилятора.....	118
Другие окна Ghidra .....	121
Окно байтов .....	121
Окно определенных данных .....	123
Окно определенных строк .....	125
Окна таблицы символов и ссылок на символы.....	126
Окно карты памяти .....	130
Окно графа вызовов функции.....	131
Резюме.....	132
<b>Глава 6. Дизассемблирование в Ghidra .....</b>	<b>135</b>
Навигация по листингу дизассемблера .....	136
Имена и метки .....	136
Навигация в Ghidra .....	137
Перейти к .....	139
История навигации .....	139
Кадры стека.....	141
Механизмы вызова функций .....	141

Соглашения о вызове .....	144
Дополнительные сведения о кадре стека .....	150
Размещение локальных переменных .....	151
Примеры кадров стека.....	152
Представления стека в Ghidra.....	157
Анализ кадров стека в Ghidra .....	158
Кадры стека в листинге дизассемблера .....	159
Анализ кадра стека с помощью декомпиллятора.....	162
Локальные переменные как операнды .....	164
Редактор кадра стека в Ghidra .....	165
Поиск.....	168
Поиск по тексту программы .....	169
Поиск в памяти .....	171
Резюме.....	173
<b>Глава 7. Управление дизассемблированием .....</b>	<b>175</b>
Манипулирование именами и метками .....	176
Переименование параметров и локальных переменных .....	177
Переименование меток .....	182
Добавление новой метки .....	183
Редактирование меток .....	185
Удаление метки .....	187
Навигация по меткам .....	187
Комментарии .....	187
Концевые комментарии.....	189
Предварительные и заключительные комментарии.....	190
Вводные комментарии.....	190
Повторяемые комментарии.....	192
Комментарии для параметров и локальных переменных .....	192
Аннотации .....	193
Базовые преобразования кода .....	194
Изменение параметров отображения кода .....	194
Форматирование operandов команд .....	196
Манипулирование функциями .....	198
Преобразование данных в код (и наоборот) .....	202
Основы преобразования данных .....	203
Задание типов данных.....	204
Работа со строками .....	206
Определение массивов.....	208
Резюме.....	209
<b>Глава 8. Типы данных и структуры данных .....</b>	<b>211</b>
В чем смысл этих данных? .....	212
Распознавание структур данных в коде .....	215
Доступ к элементам массива.....	215
Доступ к полям структуры .....	228
Массивы структур.....	234
Создание структур в Ghidra .....	236

Создание новой структуры .....	237
Редактирование полей структуры.....	240
Наложение структур .....	242
Введение в обратную разработку кода на C++ .....	244
Указатель this .....	245
Виртуальные функции и vf-таблицы.....	246
Жизненный цикл объекта .....	251
Декорирование имен.....	253
Идентификация типа во время выполнения.....	254
Отношения наследования .....	256
Справочные материалы по обратной разработке кода на C++.....	257
Резюме.....	258
<b>Глава 9. Перекрестные ссылки.....</b>	<b>259</b>
Базовые сведения о ссылках .....	260
Перекрестные (обратные) ссылки.....	261
Пример анализа ссылок .....	265
Окна управления ссылками.....	271
Окно перекрестных ссылок .....	272
Ссылки на.....	273
Ссылки на символы.....	273
Дополнительные способы работы со ссылками .....	274
Резюме.....	276
<b>Глава 10. Графы.....</b>	<b>277</b>
Простые блоки.....	278
Графы функций .....	279
Графы вызовов функций .....	290
Деревья .....	297
Резюме.....	297

## **ЧАСТЬ III. ПОСТАВИТЬ GHIDRA СЕБЕ НА СЛУЖБУ ..... 299**

<b>Глава 11. Коллективная обратная разработка программ .....</b>	<b>301</b>
Коллективная работа .....	302
Подготовка сервера Ghidra.....	303
Разделяемые проекты .....	307
Создание разделяемого проекта.....	307
Управление проектом .....	310
Меню окна проекта .....	311
Меню File.....	311
Меню Edit .....	314
Меню Project.....	316
Репозиторий проекта.....	319
Управление версиями.....	321
Пример.....	324
Резюме.....	330

<b>Глава 12. Настройка Ghidra.....</b>	<b>331</b>
Браузер кода .....	332
Реорганизация окон .....	332
Редактирование параметров инструментов.....	334
Редактирование параметров инструмента.....	337
Специальные средства редактирования	
для некоторых инструментов .....	338
Сохранение конфигурации браузера кода.....	340
Окно проекта в Ghidra .....	340
Меню Tools.....	346
Рабочие пространства .....	352
Резюме.....	353
<b>Глава 13. Расширение взгляда на мир Ghidra .....</b>	<b>355</b>
Импорт файлов .....	356
Анализаторы .....	359
Модели слов .....	360
Типы данных.....	362
Создание новых архивов типов данных .....	365
Идентификаторы функций .....	369
Плагин Function ID .....	371
Пример применения плагина Function ID: UPX.....	374
Пример применения плагина Function ID:	
профилирование статической библиотеки.....	379
Резюме.....	385
<b>Глава 14. Основы написания скриптов для Ghidra .....</b>	<b>387</b>
Диспетчер скриптов .....	388
Окно диспетчера скриптов .....	388
Панель инструментов диспетчера скриптов .....	390
Разработка скриптов .....	391
Написание скриптов на Java (не JavaScript!).....	391
Пример редактирования скрипта: поиск	
по регулярному выражению .....	393
Скрипты на Python.....	399
Поддержка других языков .....	401
Введение в Ghidra API.....	402
Интерфейс Address.....	403
Интерфейс Symbol.....	403
Интерфейс Reference.....	403
Класс GhidraScript .....	404
Функции манипулирования программой .....	410
Класс Program.....	411
Интерфейс Function .....	413
Интерфейс Instruction .....	413
Примеры скриптов Ghidra.....	414
Пример 1: перечисление функций.....	414
Пример 2: перечисление команд.....	415

Пример 3: перечисление перекрестных ссылок .....	416
Пример 4: нахождение вызовов функции .....	417
Пример 5: эмуляция поведения языка ассемблера.....	419
Резюме.....	422
<b>Глава 15. Eclipse и GhidraDev .....</b>	<b>423</b>
Eclipse .....	423
Интеграция с Eclipse.....	424
Запуск Eclipse .....	425
Редактирование скриптов в Eclipse .....	426
Меню GhidraDev .....	427
GhidraDev ▶ New .....	428
Навигация в обозревателе пакетов .....	434
Пример: проект модуля анализатора .....	441
Шаг 1: постановка задачи .....	443
Шаг 2: создать модуль в Eclipse.....	443
Шаг 3: написать анализатор .....	443
Шаг 4: протестировать анализатор в Eclipse .....	451
Шаг 5: добавить анализатор в Ghidra.....	451
Шаг 6: тестирование анализатора в Ghidra.....	452
Резюме.....	455
<b>Глава 16. Необслуживаемый режим Ghidra .....</b>	<b>457</b>
Приступая к работе .....	458
Шаг 1: запуск Ghidra .....	459
Шаги 2 и 3: создать новый проект Ghidra в указанном месте .....	459
Шаг 4: импортировать файл в проект.....	460
Шаги 5 и 6: автоматический анализ файла, сохранение и выход .....	460
Флаги и параметры.....	465
Написание скриптов .....	475
HeadlessSimpleROP .....	475
Автоматизированное создание базы данных FidDb.....	480
Резюме.....	482
<b>ЧАСТЬ IV. ДОПОЛНИТЕЛЬНЫЕ ТЕМЫ .....</b>	<b>483</b>
<b>Глава 17. Загрузчики Ghidra .....</b>	<b>485</b>
Анализ неизвестного файла.....	487
Загрузка PE-файла Windows вручную .....	488
Пример 1: модуль загрузчика SimpleShellcode.....	502
Шаг 0: шаг назад.....	503
Шаг 1: поставить задачу .....	506
Шаг 2: создать модуль в Eclipse.....	507
Шаг 3: разработать загрузчик.....	507
Шаг 4: добавить загрузчик в Ghidra .....	514
Шаг 5: протестировать загрузчик в Ghidra .....	515
Пример 2: простой загрузчик шелл-кода из исходных файлов.....	517

Обновление 1: изменить ответ на опрос импортера.....	518
Обновление 2: найти shell-код в исходном коде.....	518
Обновление 3: преобразовать shell-код в байтовые значения.....	519
Обновление 4: загрузить преобразованный байтовый массив .....	520
Результаты .....	520
Пример 3: простой загрузчик shell-кода в формате ELF .....	522
Организационные мероприятия.....	523
Формат заголовков ELF.....	524
Определение поддерживаемых спецификаций загрузки .....	525
Загрузить содержимое файла в Ghidra .....	527
Отформатировать байты данных и добавить точку входа .....	528
Файлы определений языков .....	529
Opinion-файлы .....	530
Результаты .....	532
Резюме.....	535
<b>Глава 18. Процессорные модули в Ghidra.....</b>	<b>537</b>
Знакомство с процессорным модулем Ghidra .....	539
Процессорные модули в Eclipse .....	539
SLEIGH.....	541
Руководства по процессорам .....	543
Модификация процессорного модуля Ghidra .....	545
Постановка задачи .....	547
Пример 1: добавление команды в процессорный модуль .....	547
Пример 2: модификация команды в процессорном модуле .....	556
Вариант 1: записать в EAX константу .....	556
Пример 3: добавление регистра в процессорный модуль .....	567
Резюме.....	570
<b>Глава 19. Декомпилятор Ghidra .....</b>	<b>571</b>
Анализ с помощью декомпилятора .....	571
Параметры анализа .....	572
Окно декомпилятора.....	575
Пример 1: редактирование в окне декомпилятора .....	576
Пример 2: функции, не возвращающие управление .....	582
Пример 3: автоматизированное создание структуры .....	584
Резюме.....	590
<b>Глава 20. Зависимость от компилятора .....</b>	<b>591</b>
Высокоуровневые конструкции .....	592
Предложения switch .....	592
Пример: сравнение компиляторов gcc и Microsoft C/C++ .....	599
Параметры компилятора.....	602
Пример 1: оператор деления по модулю .....	603
Пример 2: тернарный оператор .....	606
Пример 3: встраивание функций .....	608
Реализация зависящих от компилятора особенностей C++ .....	610
Перегрузка функций.....	611
Реализации RTTI .....	612

Нахождение функции main .....	617
Пример 1: от _start к main с компилятором gcc для Linux x86-64 .....	618
Пример 2: от _start к main с компилятором clang для FreeBSD x86-64.....	619
Пример 3: от _start к main с компилятором Microsoft's C/C++ .....	620
Резюме.....	621

## **ЧАСТЬ V. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ ..... 623**

<b>Глава 21. Анализ обfuscированного кода .....</b>	<b>625</b>
Противодействие обратной разработке.....	626
Обфускация.....	626
Методы противодействия статическому анализу .....	627
Обфускация импортированной функции .....	643
Методы противодействия динамическому анализу .....	648
Статическая деобфускация двоичных файлов в Ghidra.....	654
Скриптовая деобфускация .....	654
Эмуляторная деобфускация .....	661
Резюме.....	670
<b>Глава 22. Изменение двоичного кода.....</b>	<b>673</b>
Планирование заплаты .....	674
Поиск того, что нуждается в изменении .....	675
Поиск в памяти.....	675
Поиск прямых ссылок .....	676
Поиск командных паттернов .....	677
Поиск конкретных типов поведения .....	682
Наложение заплаты.....	683
Внесение простых изменений .....	683
Внесение нетривиальных изменений .....	690
Экспорт файлов .....	694
Форматы экспорта из Ghidra .....	695
Двоичный формат экспорта .....	696
Экспорт с применением скрипта .....	697
Пример: латание двоичного файла.....	699
Резюме.....	703
<b>Глава 23. Определение разности двоичных файлов и отслеживание версий.....</b>	<b>705</b>
Разность двоичных файлов .....	706
Инструмент Program Diff.....	708
Пример: объединение двух проанализированных файлов .....	712
Сравнение функций .....	717
Окно сравнения функций.....	717
Пример: сравнение криптографических функций.....	720
Отслеживание версий .....	727
Концепции, относящиеся к отслеживанию версий .....	728
Резюме.....	730

<b>Ghidra для пользователей IDA.....</b>	<b>731</b>
Основы .....	731
Создание базы данных .....	732
Основные окна и навигация .....	734
Дерево символов .....	737
Скрипты.....	738
Резюме.....	738
<b>Предметный указатель.....</b>	<b>739</b>

# БЛАГОДАРНОСТИ

Эта книга не состоялась бы без помощи и поддержки со стороны исключительно профессионального коллектива издательства No Starch Press. Билл Поллок и Барбара Яин поддержали идею написать книгу о Ghidra, отражающую наше видение, и мы глубоко ценим их веру в нас, не ослабевавшую на протяжении всего пути. Отзывы Атабаска Уитши о первых главах стали для нас ценным подспорьем и указали верное направление. Постоянная поддержка Лорел Чан и ее терпеливые ответы на все наши вопросы помогли превратить книгу в готовый продукт, которым мы очень гордимся. Мы также хотим поблагодарить всех, кто оставался «за кулисами», за тяжелую работу по претворению мечты в реальность, в т. ч. Катрину Тэйлор, Бартона Д. Рида, Шарон Уилки и Даниэля Фостера.

Мы также благодарим технического редактора Брайана Хэя, который проштудировал наши пространные словеса и примеры. Его знания и опыт работы с Ghidra стали гарантией безошибочности содержания книги с технической точки зрения, а его опыт преподавания позволил изложить материал так, чтобы он представлял интерес как для начинающих, так и для опытных инженеров.

Мы признательны всей команде разработчиков Ghidra, прошлых и нынешних, из Агентства национальной безопасности за то, что они создали Ghidra и поделились ей со всем миром, сделав проектом с открытым исходным кодом.

Кара выражает благодарность Бену за терпение, которое он проявлял, когда она изучала технологию, и Кэти за терпение, проявленное во время написания книги. Она также благодарит Йена за вдохновляющее введение и Дики и Ленору, которые никогда не теряли веру в нее. Наконец, она благодарит Брайана за юмор и непрекращающуюся ежедневную и ежечасную поддержку. Не будь этой поддержки, книга не увидела бы свет.

# 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 дизассемблера 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 графиками 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 обfuscированного 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 сайтах <https://nostarch.com/GhidraBook/> и <https://ghidrabook.com/>, а также на сайте издательства <https://dmkpress.com/catalog/computer/security/978-5-97060-942-2/>*





# **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, С 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-разряд-

ное целое 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 С-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. Напротив, *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 декомпиллятора.

## 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<sup>1</sup>, 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 ли она эксплуатацию, и если да, то при каких условиях.

<sup>1</sup> *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 байт. 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. Аналистам 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 представле-

ние 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<sup>1</sup>.
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.

---

<sup>1</sup> 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 operandов: исходный operand 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 operandов: 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, различия 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 ветвлений.

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-0x6ffffbfee],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-0x4dfffbfee],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<sup>1</sup>, то мы увидим, что каждая представляет собой указатель на близлежащий адрес, т. е. конечные адреса команд переходов (`004012e0`, `0040128b`, `00401290`, ...). Таким образом, команда `loopne` ❸ – 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, не-посредственно 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.

<sup>1</sup> 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 *вызыва 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], gax❶
```

---

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 отличия 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
```

---

### *Листинг 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.

# 2

## ОБРАТНАЯ РАЗРАБОТКА И ИНСТРУМЕНТЫ ДИЗАССЕМБЛИРОВАНИЯ



Итак, мы располагаем базовыми знаниями о дизассемблировании. Но прежде чем приступить к специфике Ghidra, будет полезно познакомиться с другими инструментами, применяемыми для обратной разработки кода. Многие из них появились раньше Ghidra и по-прежнему используются, чтобы получить первое представление о файле, а также для проверки правильности результатов, выданных Ghidra. Мы увидим, что Ghidra включила многие возможности этих инструментов в свой интерфейс с целью предложить единую интегрированную среду для обратной разработки.

# СРЕДСТВА КЛАССИФИКАЦИИ

Сталкиваясь с незнакомым файлом, мы часто хотим получить ответ на простые вопросы, например: «Что это за зверь такой?» Первое правило – *никогда* не полагаться на расширение файла, желая узнать, что в нем находится. Это не только первое, но второе, третье и четвертое правила. Смирившись с мыслью, что *расширения файла не имеют никакого смысла*, можете познакомиться с одной или несколькими из описанных ниже утилит.

## file

Команда `file` – стандартная утилита, входящая в состав большинства операционных систем типа \*nix, а также в подсистему Windows для Linux (Windows Subsystem for Linux – WSL)<sup>1</sup>. Пользователи Windows могут получить эту команду, установив Cygwin или MinGW<sup>2</sup>. Команда `file` пытается определить тип файла, анализируя некоторые поля в нем. Иногда `file` распознает хорошо известные строки, например `#!/bin/sh` (скрипт оболочки) или `<html>` (HTML-документ).

Файлы с нетекстовым содержимым сложнее. В таких случаях `file` пытается определить, соответствует ли структура файла какому-нибудь известному формату. Часто она ищет некоторые признаки (называемые *магическими числами*)<sup>3</sup>, уникальные для файлов данного типа. Ниже приведены примеры некоторых магических чисел и соответствующих им типов файлов.

---

<sup>1</sup> См. <https://docs.microsoft.com/en-us/windows/wsl/about/>.

<sup>2</sup> О Cygwin см. <http://www.cygwin.com/>. О MinGW см. <http://www.mingw.org/>.

<sup>3</sup> *Магическое число* – это специальный признак, описанный в спецификациях некоторых форматов файлов, наличие которого означает соответствие данной спецификации. Иногда выбор магических чисел – шутка. Так, признак MZ в заголовке исполняемого файла MS-DOS – инициалы Марка Збиковски, одного из первых архитекторов MS-DOS, а шестнадцатеричное значение 0xcafebabe, ассоциированное с файлами классов в Java (с расширением `.class`), было выбрано, потому что эта последовательность легко запоминается.

---

```
Формат исполняемого файла Windows PE
00000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
MZ.....
00000010 B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00
.....@.....
Графический формат Jpeg
00000000 FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 60 .....
JFIF.....
00000010 00 60 00 00 FF DB 00 43 00 0A 07 07 08 07 06 0A
.`.....C.....
.class-файл Java
00000000 CA FE BA BE 00 00 00 32 00 98 0A 00 2E 00 3E 08
.....2.....>.
00000010 00 3F 09 00 40 00 41 08 00 42 0A 00 43 00 44 0A
.?..@.A..B..C.D.
```

Команда `file` распознает много форматов, в т. ч. несколько типов текстовых ASCII-файлов, различные исполняемые файлы и файлы данных. Правила проверки магических чисел хранятся в *магическом файле*. Где этот файл находится по умолчанию, зависит от операционной системы, типичные местоположения: `/usr/share/file/magic`, `/usr/share/misc/magic` и `/etc/magic`. Дополнительные сведения о магических файлах смотрите в документации по программе `file`.

В некоторых случаях `file` умеет распознавать вариации в пределах заданного типа файла. В листинге ниже показано, что `file` распознает не только несколько вариантов формата ELF, но также выдает информацию о том, как файл был скомпонован (статически или динамически) и был ли он обработан программой `strip`, удаляющей информацию о символах.

---

```
ghidrabook# file ch2_ex_*
ch2_ex_x64: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
              dynamically linked, interpreter /lib64/l, for GNU/Linux
              3.2.0, not stripped
ch2_ex_x64_dbg: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                  dynamically linked, interpreter /lib64/l, for GNU/Linux
                  3.2.0, with debug_info, not stripped
ch2_ex_x64_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
                     statically linked, for GNU/Linux 3.2.0, not stripped
ch2_ex_x64_strip: ELF 64-bit LSB shared object, x86-64, version 1
                   (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux
                   3.2.0, stripped
```

```
ch2_ex_x86: ELF 32-bit LSB shared object, Intel 80386, version 1
              (SYSV), dynamically linked, interpreter /lib/ld-, for
              GNU/Linux 3.2.0, not stripped
ch2_ex_x86_dbg: ELF 32-bit LSB shared object, Intel 80386, version 1
                  (SYSV), dynamically linked, interpreter /lib/ld-, for
                  GNU/Linux 3.2.0, with debug_info, not stripped
ch2_ex_x86_static: ELF 32-bit LSB executable, Intel 80386, version 1
                     (GNU/Linux), statically linked, for GNU/Linux 3.2.0,
                     not stripped
ch2_ex_x86_strip: ELF 32-bit LSB shared object, Intel 80386, version 1
                   (SYSV), dynamically linked, interpreter /lib/ld-, for
                   GNU/Linux 3.2.0, stripped
ch2_ex_Win32: PE32 executable (console) Intel 80386, for MS Windows
ch2_ex_x64: PE32+ executable (console) x86-64, for MS Windows
```

---

## Среда WSL

Подсистема Windows для Linux (WSL) предлагает командное окружение GNU/Linux прямо в Windows без необходимости создавать виртуальную машину. В процессе установки WSL пользователь выбирает дистрибутив Linux, после чего может запускать его в WSL. Тем самым пользователь получает доступ к стандартным командным утилитам (`grep`, `awk`), компиляторам (`gcc`, `g++`), интерпретаторам (Perl, Python, Ruby), сетевым утилитам (`nc`, `ssh`) и многому другому. После установки WSL многие программы, написанные для Linux, можно откомпилировать и выполнить в системах Windows.

Утилита `file` и ей подобные не дают стопроцентно верного результата. Вполне может случиться, что файл распознан неправильно, потому что он случайно содержит характерные признаки определенного формата. Можете убедиться в этом сами, заменив в шестнадцатеричном редакторе первые 4 байта любого файла магической последовательностью Java: `CA FE BA BE`. После этого `file` некорректно идентифицирует модифицированный файл как *откомпилированный класс Java*. Аналогично текстовый файл, содержащий всего два символа `MZ`, будет идентифицирован как *исполняемый файл MS-DOS*. В процессе обратной разработки лучше всего не доверять полностью результату любого инструмента, не сопоставив результаты нескольких инструментов и не проанализировав данные вручную.

## Зачистка двоичных исполняемых файлов

Зачисткой двоичного файла называется процесс удаления из него символов. В результате компиляции в двоичных объектных файлах остается информация о символах. Некоторые символы используются в процессе компоновки для разрешения ссылок между файлами, что необходимо при создании окончательного исполняемого файла или библиотеки. Другие символы несут дополнительную информацию для отладчика. По завершении процесса компоновки многие символы уже не нужны. Компоновщику можно передать параметр, заставляющий удалить ненужные символы на этапе сборки. Или же можно воспользоваться утилитой `strip`, которая удаляет символы из существующего файла. Результирующий файл будет меньше по размеру, но его поведение не изменится.

## PE Tools

PE Tools – набор инструментов, полезный для анализа выполняемых процессов и исполняемых файлов в системах Windows<sup>1</sup>. На рис. 2.1 показан основной интерфейс PE Tools – отображается список активных процессов и предоставляется доступ ко всем входящим в состав набора утилитам.

Отправляясь от списка процессов, пользователь может записать дамп памяти процесса в файл или воспользоваться утилитой PE Sniffer, чтобы определить, какой компилятор создал исполняемый файл и был ли файл обработан какой-нибудь из известных утилит обfuscации. Меню **Tools** предлагает аналогичные возможности для анализа файлов на диске. Пользователь может просматривать поля заголовка PE-файла с помощью встроенной утилиты PE Editor, которая также позволяет изменять значения полей. Модификация заголовка PE-файла часто необходима для реконструкции оригинального файла из его обfuscatedированной версии.

<sup>1</sup> См. <https://github.com/petoolse/petools/>.

## Обфускация двоичного файла

Обфускацией называется любая попытка запутать истинный смысл чего-либо. Применительно к исполняемым файлам обфускация – это попытка скрыть истинное поведение программы. Программисты используют обфускацию по ряду причин. Типичные примеры: защита коммерческих алгоритмов и скрытие злых намерений. Почти все виды вредоносного ПО применяют обфускацию с целью противостоять попыткам анализа. Существуют многочисленные инструменты, помогающие авторам создавать обфусцированные программы. Эти инструменты и методы, а также их влияние на процесс обратного конструирования будут обсуждаться в главе 21.

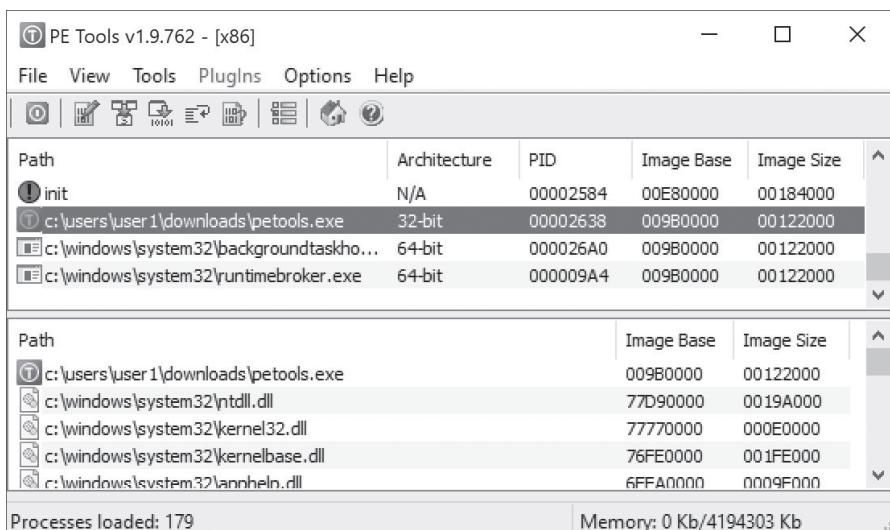


Рис. 2.1. Утилита PE Tools

## PEiD

PEiD – еще один инструмент для Windows, основная цель которого – определить, какой компилятор создал данный двоичный PE-файл и какие инструменты применялись для его обфускации<sup>1</sup>. На рис. 2.2 показано применение PEiD для определения инструмента (в данном случае ASPack), которым был обфусцирован вариант червя Gaobot<sup>2</sup>.

<sup>1</sup> См. <https://github.com/wolfram77web/app-peid/>.

<sup>2</sup> См. <https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/GAOBOT/>.

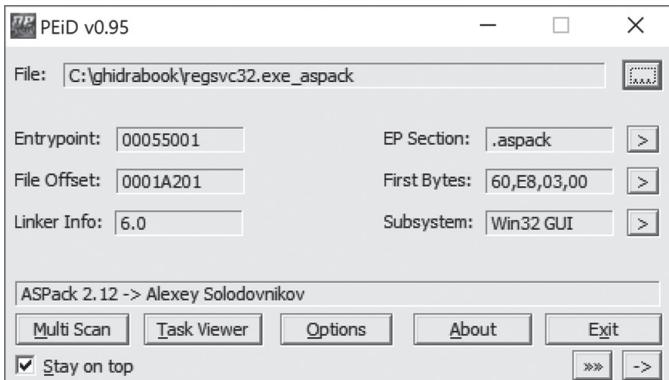


Рис. 2.2. Утилита PEiD

Многие дополнительные возможности PEiD перекрываются с возможностями PE Tools, в частности умение генерировать сводку заголовков PE-файла, собирать информацию о работающих процессах и выполнять простое дизассемблирование.

## ОБЗОРНЫЕ ИНСТРУМЕНТЫ

Поскольку наша цель – обратная разработка двоичных файлов программ, нам понадобятся более изощренные инструменты для извлечения подробной информации после начальной классификации файла. Инструменты, обсуждаемые в этом разделе, по необходимости гораздо больше знают о форматах обрабатываемых файлов. В большинстве случаев инструменты понимают один конкретный формат и используются для разбора файла и извлечения весьма специальной информации.

### ***nm***

Создавая объектный файл из исходного, компилятор должен включить информацию о местоположении всех глобальных (внешних) символов, чтобы компоновщик смог разрешить ссылки на них в процессе объединения нескольких объектных файлов с целью создания исполняемого. Если не было указаний удалить символы из окончательного исполняемого файла, то компоновщик обычно переносит символы из объектных файлов в исполняемый. Согласно странице руководства, утилита **nm** «печатает список символов в объектных файлах».

По умолчанию `nm`, примененная к промежуточному объектному файлу (с расширением `.o`), выводит имена всех функций и глобальных переменных, объявленных в файле. Ниже приведен пример работы `nm`.

```
ghidrabook# gcc -c ch2_nm_example.c
ghidrabook# nm ch2_nm_example.o
U exit
U fwrite
0000000000000002e T get_max
U __GLOBAL_OFFSET_TABLE__
U __isoc99_scanf
0000000000000000a6 T main
0000000000000000 D my_initialized_global
0000000000000004 C my_uninitialized_global
U printf
U puts
U rand
U srand
U __stack_chk_fail
U stderr
U time
0000000000000000 T usage
ghidrabook#
```

Мы видим, что `nm` выводит все символы и информацию о каждом из них. Буквенный код обозначает тип символа. В данном примере встречаются следующие коды:

- U** неопределенный символ (обычно ссылка на внешний символ);
- T** символ, определенный в секции `text` (обычно имя функции);
- t** локальный символ, определенный в секции `text`. В программе на С это обычно статическая функция;
- D** инициализированные данные;
- C** неинициализированные данные.

#### ПРИМЕЧАНИЕ

Заглавные буквы используются для глобальных символов, а строчные – для локальных. Дополнительные сведения, в т. ч. описание всех буквенных кодов, можно найти на странице руководства по `nm`.

Несколько больше информации выводится, когда `nm` используется для отображения символов в исполняемом файле.

В процессе компоновки символам сопоставляются виртуальные адреса (если возможно), поэтому `nm` доступно больше информации. Сокращенная распечатка, созданная при запуске `nm` для исполняемого файла, приведена ниже.

---

```
ghidraobook# gcc -o ch2_nm_example ch2_nm_example.c
ghidraobook# nm ch2_nm_example
...
U fwrite@@GLIBC_2.2.5
0000000000000938 t get_max
0000000000201f78 d __GLOBAL_OFFSET_TABLE__
w __gmon_start__
0000000000000c5c r __GNU_EH_FRAME_HDR
0000000000000730 T __init
00000000000201d80 t __init_array_end
00000000000201d78 t __init_array_start
0000000000000b60 R __IO_stdin_used
        U __isoc99_scanf@@GLIBC_2.7
        w __ITM_deregisterTMCloneTable
        w __ITM_registerTMCloneTable
0000000000000b50 T __libc_csu_fini
0000000000000ae0 T __libc_csu_init
        U __libc_start_main@@GLIBC_2.2.5
00000000000009b0 T main
0000000000202010 D my_initialized_global
000000000020202c B my_uninitialized_global
        U printf@@GLIBC_2.2.5
        U puts@@GLIBC_2.2.5
        U rand@@GLIBC_2.2.5
0000000000000870 t register_tm_clones
        U srand@@GLIBC_2.2.5
        U __stack_chk_fail@@GLIBC_2.4
0000000000000800 T __start
0000000000202020 B stderr@@GLIBC_2.2.5
        U time@@GLIBC_2.2.5
0000000000202018 D __TMC_END__
000000000000090a T usage
ghidraobook#
```

---

Теперь некоторым символам (например, `main`) сопоставлены виртуальные адреса. Как результат процесса компоновки появились новые символы (`__libc_csu_init`), для символов некоторых изменился тип (например, `my_uninitialized_global`), тогда как остальные остались неопределенными, поскольку продол-

жают ссылаться на внешние символы. В данном случае исследуемый двоичный файл скомпонован динамически, а неопределенные символы определены в разделяемой С-библиотеке.

## ***Idd***

В процессе создания исполняемого файла должны быть разрешены ссылки на библиотечные функции. Существует два метода разрешения таких ссылок: *статическая компоновка* и *динамическая компоновка*. Какой из них выбрать, определяется аргументами командной строки компоновщика. Любой исполняемый файл может быть скомпонован статически, динамически или обоими способами<sup>1</sup>.

Если запрашивается статическая компоновка, то компоновщик объединяет объектные файлы приложения с копией за- требованной библиотеки и таким образом создает исполняемый файл. Во время выполнения нет необходимости искать библиотечный код, потому что он уже содержится внутри исполняемого файла. Статическая компоновка имеет следующие преимущества: (1) функции вызываются чуть быстрее и (2) распространение двоичных файлов проще, потому что не делается никаких предположений о доступности библиотечного кода в системе пользователя. Но у нее есть и недостатки: (1) размер исполняемого файла увеличивается и (2) сложнее переходить на новую версию программы, когда изменяются библиотеки. Последнее связано с тем, что программу нужно пересобирать всякий раз, как изменяется какая-нибудь библиотека. С точки зрения обратной разработки, статическая компоновка несколько усложняет задачу. Анализируя статически скомпонованный файл, мы не можем так просто ответить на вопросы типа «С какими библиотеками скомпонован файл?» и «Какие из имеющихся функций взяты из библиотеки?». В главе 13 обсуждаются проблемы обратной разработки статически скомпонованных файлов.

Динамическая компоновка отличается от статической тем, что компоновщику не нужно копировать затребованные библиотеки. Вместо этого он вставляет только ссылки на эти библиотеки (часто файлы с расширением .so или .dll) в окончательный исполняемый файл, поэтому размер конечного файла

---

<sup>1</sup> Дополнительные сведения о компоновке см. в книге John R. Levine «Linkers and Loaders» (Morgan Kaufmann, 1999).

обычно оказывается гораздо меньше. Обновление библиотечного кода тоже существенно упрощается. Поскольку существует всего одна копия библиотеки, на которую ссылается много двоичных файлов, простая замена старой версии библиотеки новой приводит к тому, что все динамически скомпонованные с ней файлы автоматически начинают пользоваться новой версией. К недостаткам динамической компоновки можно отнести более сложный процесс загрузки. Все необходимые библиотеки должны быть найдены и загружены в память, тогда как в случае статической компоновки нужно загрузить только один файл, который уже содержит весь библиотечный код. Еще один недостаток динамической компоновки – необходимость включать в дистрибутив не только собственный исполняемый файл, но и все библиотеки, от которых этот файл зависит. Попытка выполнить программу в системе, где нет необходимых библиотек, приведет к ошибке.

Ниже демонстрируется создание динамически и статически скомпонованных версий программы, размер получающихся двоичных файлов и то, как их идентифицирует утилита `file`:

```
ghidrobook# gcc -o ch2_example_dynamic ch2_example.c
ghidrobook# gcc -o ch2_example_static ch2_example.c -static
ghidrobook# ls -l ch2_example_*
-rwxrwxr-x 1 ghidrobook ghidrobook 12944 Nov 7 10:07 ch2_example_dynamic
-rwxrwxr-x 1 ghidrobook ghidrobook 963504 Nov 7 10:07 ch2_example_static
ghidrobook# file ch2_example_*
ch2_example_dynamic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=e56ed40012accb3734bde7f8bca3cc2c368455c3, not stripped
ch2_example_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=430996c6db103e4fe76aea7d578e636712b2b4b0, not stripped
ghidrobook#
```

Для правильной работы динамически скомпонованный файл должен содержать информацию о том, от каких библиотек он зависит, и о том, что нужно взять из каждой библиотеки. Поэтому, в отличие от статически скомпонованных файлов, определить, от каких библиотек зависит динамически скомпонованный файл, очень просто. Утилита `ldd` (*list dynamic dependencies* – перечислить динамические зависимости) выводит список динамических библиотек, необходимых данному

исполняемому файлу. В примере ниже `ldd` используется, чтобы определить, от каких библиотек зависит веб-сервер Apache:

```
ghidrabook# ldd /usr/sbin/apache2
linux-vdso.so.1 => (0x00007ffffc1c8d000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fbebe7410000)
libaprutil-1.so.0 => /usr/lib/x86_64-linux-gnu/libaprutil-1.so.0 (0x00007fbebe71e0000)
libapr-1.so.0 => /usr/lib/x86_64-linux-gnu/libapr-1.so.0 (0x00007fbebe6fa0000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fbebe6d70000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbebe69a0000)
libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1 (0x00007fbebe6760000)
libexpat.so.1 => /lib/x86_64-linux-gnu/libexpat.so.1 (0x00007fbebe6520000)
libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007fbebe6310000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fbebe6100000)
/lib64/ld-linux-x86-64.so.2 (0x00007fbebe7a00000)
ghidrabook#
```

Утилита `ldd` входит в состав систем Linux и BSD. В системах macOS аналогичную функциональность предлагает утилита `otool` с флагом `-L`: `otool -L filename`. В Windows для вывода списка зависимых библиотек служит утилита `dumpbin`, входящая в состав Visual Studio: `dumpbin /dependents filename`.

## Остерегайтесь своих инструментов!

Может показаться, что `ldd` – простая программа, но страница руководства для нее предупреждает: «никогда не следует запускать `ldd` для исполняемого файла, полученного из ненадежного источника, потому что это может привести к выполнению произвольного кода». В большинстве случаев это маловероятно, но служит напоминанием, что запуск даже кажущихся простыми инструментов обратного конструирования может иметь нежелательные последствия при исследовании не заслуживающих доверия входных файлов. Мы надеемся, никого не нужно убеждать в том, что выполнение ненадежных двоичных программ вряд ли безопасно, но призываем принимать меры предосторожности даже при статическом анализе таких программ и предполагать, что компьютер, на котором производится обратное конструирование, а также все данные на нем и на других соединенных с ним компьютерах могут быть в результате скомпрометированы.

## ***objdump***

Если `ldd` – специализированная программа, то `objdump` – исключительно гибкий инструмент. Цель `objdump` – «вывести информацию, хранящуюся в объектных файлах»<sup>1</sup>. Это довольно широко поставленная задача, и для ее решения `objdump` принимает более 30 аргументов командной строки, указывающих, что именно нужно извлечь из объектных файлов. Утилита `objdump` умеет показывать следующие данные (и еще много чего).

**Заголовки секций.** Сводная информация о каждой секции программного файла.

**Частные заголовки.** Информация о размещении программы в памяти и другие сведения, необходимые загрузчику, в т. ч. такой же список требуемых библиотек, какой выводит `ldd`.

**Отладочная информация.** Вся отладочная информация, включенная в файл.

**Информация о символах.** Таблица символов в таком же формате, как ее выводит утилита `nm`.

**Листинг дизассемблера.** Программа `objdump` применяется алгоритм линейной развертки для дизассемблирования секций файла, помеченных как код. В процессе дизассемблирования кода x86 `objdump` может генерировать ассемблерный код в синтаксисе AT&T или Intel и записывать результат в текстовый файл. Такие текстовые файлы называются *мертвыми листингами*, и хотя их, безусловно, можно использовать для обратной разработки, они крайне неудобны для навигации, а еще труднее вносить в них согласованные изменения, не наделав ошибок.

Программа `objdump` входит в состав комплекта GNU binutils, который имеется в Linux, FreeBSD и Windows (если установлены WSL или Cygwin)<sup>2</sup>. Отметим, что `objdump` опирается на библиотеку *Binary File Descriptor* (`libbfd`), являющуюся составной частью binutils, которая применяется для доступа к объектным файлам, и потому умеет разбирать все форматы файлов, под-

---

<sup>1</sup> См. <http://www.sourceware.org/binutils/docs/binutils/objdump.html>.

<sup>2</sup> См. <http://www.gnu.org/software/binutils/>.

держиваемые libbfd (в т. ч. ELF и PE). Специально для разбора ELF-файлов имеется также утилита `readelf`, которая предлагает в основном те же возможности, что `objdump`, а отличается от нее прежде всего тем, что не зависит от libbfd.

## ***otool***

Утилиту `otool` проще всего описать как аналог `objdump` для macOS, она полезна для разбора файлов в формате Mach-O. В листинге ниже показано, как `otool` отображает зависимости двоичного файла в формате Mach-O от динамических библиотек, т. е. выполняет функцию `ldd`:

---

```
ghidrabook# file osx_example
osx_example: Mach-O 64-bit executable x86_64
ghidrabook# otool -L osx_example
osx_example:
/usr/lib/libstdc++.6.dylib (compatibility version 7.0.0, current version 7.4.0)
/usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1281.0.0)
```

---

Утилиту `otool` можно использовать для отображения информации из заголовков файла и таблицы символов, а также для дизассемблирования секции кода. Дополнительные сведения см. в странице руководства.

## ***dumpbin***

Командная утилита `dumpbin` входит в состав Microsoft Visual Studio. Подобно `otool` и `objdump`, `dumpbin` умеет отображать разного рода информацию, хранящуюся в файлах в формате Windows PE. Ниже показано, как `dumpbin` выводит динамические зависимости программы `notepad` примерно так же, как это делает `ldd`.

---

```
$ dumpbin /dependents C:\Windows\System32\notepad.exe
Microsoft (R) COFF/PE Dumper
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file notepad.exe

File Type: EXECUTABLE IMAGE
```

---

Image has the following delay load dependencies:

```
ADVAPI32.dll  
COMDLG32.dll  
PROPSYS.dll  
SHELL32.dll  
WINSPOOL.DRV  
urlmon.dll
```

Image has the following dependencies:

```
GDI32.dll  
USER32.dll  
msvcrt.dll  
...
```

---

Дополнительные параметры `dumpbin` позволяют извлекать информацию из различных секций PE-файла: символы, имена импортированных функций, имена экспортованных функций, дизассемблированный код. Более подробные сведения о программе `dumpbin` имеются на сайте Microsoft<sup>1</sup>.

## C++filt

Языки, допускающие перегрузку функций, должны предоставлять механизм различения перегруженных вариантов функции, потому что все они имеют одно и то же имя. В следующем примере, написанном на C++, показаны прототипы нескольких перегруженных вариантов функции `demo`:

---

```
void demo(void);  
void demo(int x);  
void demo(double x);  
void demo(int x, double y);  
void demo(double x, int y);  
void demo(char* str);
```

---

Вообще говоря, в одном объектном файле не может быть двух функций с одинаковым именем. Чтобы сделать перегрузку возможной, компиляторы генерируют уникальные имена, включая информацию о типах аргументов функции. Процедура порожде-

---

<sup>1</sup> См. <https://docs.microsoft.com/en-us/cpp/build/reference/dumpbin-command-line/>.

ния уникальных имен функций с одинаковым именем называется *декорированием имени* (name mangling)<sup>1</sup>. Воспользовавшись `nm` для вывода символов в откомпилированной версии показанного выше кода на C++, мы можем увидеть нечто подобное (вывод профильтрован, чтобы не отвлекаться ни на что, кроме `demo`):

---

```
ghidrabook# g++ -o ch2_cpp_example ch2_cpp_example.cc
ghidrabook# nm ch2_cpp_example | grep demo
0000000000000060b T _Z4demod
00000000000000626 T _Z4demodi
00000000000000601 T _Z4demoi
00000000000000617 T _Z4demoid
00000000000000635 T _Z4demoPc
000000000000005fa T _Z4demov
```

---

Стандарт C++ не определяет схему декорирования имен, оставляя это на усмотрение разработчиков компиляторов. Чтобы декодировать различные варианты `demo`, нужен инструмент, который понимает схему декорирования имен, принятую в нашем компиляторе (в данном случае `g++`). Именно в этом и состоит назначение утилиты `c++filt`, которая рассматривает каждое входное слово, как будто это декорированное имя, и пытается определить, какой компилятор его генерировал. Если строка выглядит как правильно декорированное имя, то на выходе печатается его декодированный вариант. Если же `c++filt` не может интерпретировать строку, то просто выводит ее без изменения.

Если пропустить результаты `nm` из предыдущего примера через `c++filt`, то будут восстановлены декорированные имена функций:

---

```
ghidrabook# nm ch2_cpp_example | grep demo | c++filt
0000000000000060b T demo(double)
00000000000000626 T demo(double, int)
00000000000000601 T demo(int)
00000000000000617 T demo(int, double)
00000000000000635 T demo(char*)
000000000000005fa T demo()
```

---

<sup>1</sup> О декорировании имен см. статью [http://en.wikipedia.org/wiki/Name\\_mangling](http://en.wikipedia.org/wiki/Name_mangling).

Важно отметить, что декорированные имена несут дополнительную информацию о функциях, которую *nm* обычно не предоставляет. Эта информация может быть очень полезна для обратной разработки и в более сложных случаях может включать данные об именах классов или соглашениях о вызове функций.

## ИНСТРУМЕНТЫ ГЛУБОКОЙ ИНСПЕКЦИИ

До сих пор мы обсуждали инструменты, выполняющие поверхностный анализ файлов на основе минимальных знаний об их внутренней структуре. Мы также видели инструменты, способные извлекать специфические данные из файлов, поскольку обладают очень детальными знаниями об их структуре. В этом разделе мы поговорим об инструментах для извлечения специфической информации независимо от типа анализируемого файла.

### ***strings***

Иногда полезно задавать более общие вопросы о содержимом файла, для ответа на которые необязательно во всех подробностях знать структуру. Один такой вопрос: «Есть ли в файле какие-нибудь строки?» Разумеется, сначала нужно решить, что же такое строка. Определим *строку* неформально – как непрерывную последовательность печатных символов. Часто это определение дополняется условием на минимальную длину и конкретную кодировку. Таким образом, можно было бы поискать последовательности, содержащие по меньшей мере четыре печатных символа ASCII подряд, и вывести результаты на консоль. Поиск таких строк, как правило, никак не связан со структурой файла. Их можно искать как в двоичном ELF-файле, так и в документе Microsoft Word.

Утилита *strings* предназначена специально для выделения строк из содержимого файла, часто безотносительно к формату последнего. Вызов *strings* с параметрами по умолчанию (не менее четырех символов в 7-битовой кодировке ASCII подряд) может дать такой результат:

```
ghidrobook# strings ch2_example
/lib64/ld-linux-x86-64.so.2
libc.so.6
exit
srand
__isoc99_scanf
puts
time
__stack_chk_fail
printf
stderr
fwrite
__libc_start_main
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMClockTable
__gmon_start__
_ITM_registerTMClockTable
usage: ch4_example [max]
A simple guessing game!
Please guess a number between 1 and %d.
Invalid input, quitting!
Congratulations, you got it in %d attempt(s)!

Sorry too low, please try again
Sorry too high, please try again
GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
...

```

## Почему утилита `strings` изменилась?

Исторически в применении к исполняемым файлам `strings` по умолчанию искала последовательности символов только в загружаемых секциях, содержащих инициализированные данные. Поэтому `strings` должна была уметь разбирать двоичный файл и искать такие секции, для чего использовались библиотеки типа `libbfd`. Но при разборе ненадежных двоичных файлов уязвимости в библиотеках могли привести к выполнению произвольного кода<sup>1</sup>. В итоге поведение `strings` по умолчанию было изменено, и она стала просматривать весь двоичный файл, а не только секции с инициализированными данными (это то же самое, что запуск с флагом `-a`). А прежнее поведение эквивалентно заданию флага `-d`.

<sup>1</sup> См. CVE-2014-8485 и <https://lcamtuf.blogspot.com/2014/10/psa-dont-run-strings-on-untrusted-files.html>.

К сожалению, хотя некоторые строки выглядят как вывод программы, есть и другие – похожие на имена функций и библиотек. Так что не следует торопиться с выводами о поведении программы. Аналитики часто совершают ошибку, пытаясь что-то сказать о поведении программы на основе распечатки `strings`. Не забывайте, что наличие строки в двоичном файле еще не означает, что эта строка как-то используется в процессе его работы.

В заключение сделаем несколько замечаний об использовании `strings`:

- ▶ по умолчанию `strings` ничего не говорит о том, в каком месте файла находится строка. Задайте флаг `-t`, если хотите, чтобы `strings` печатала смещение каждой найденной строки от начала файла;
- ▶ во многих файлах используются альтернативные кодировки. Флаг `-e` означает, что `strings` должна искать широкие символы, например в 16-разрядной кодировке 16 Юникод.

## Дизассемблеры

Как уже было сказано, имеются инструменты, генерирующие мертвые листинги дизассемблирования двоичных объектных файлов. Файлы в форматах PE, ELF и Mach-O можно дизассемблировать программами `dumpbin`, `objdump` и `otool` соответственно. Но ни одна из них не умеет работать с произвольными блоками двоичных данных. Иногда можно встретить файл в нестандартном формате, и в этом случае нужно использовать инструменты, способные начать дизассемблирование с заданного пользователем смещения.

Два примера таких *потоковых дизассемблеров* для набора команд x86: `ndisasm` и `diStorm`<sup>1</sup>. Утилита `ndisasm` входит в состав NASM<sup>2</sup>. В примере ниже показано использование `ndisasm` для дизассемблирования части шелл-кода, сгенерированного системой Metasploit<sup>3</sup>.

<sup>1</sup> См. <https://github.com/gdabah/distorm/>.

<sup>2</sup> См. <http://www.nasm.us/>.

<sup>3</sup> См. <https://metasploit.com/>.

```
ghidrabook# msfvenom -p linux/x64/shell_find_port -f raw > findport
ghidrabook# ndisasm -b 64 findport
00000000 4831FFxor rdi,rdi
00000003 4831DBxor rbx,rbx
00000006 B314mov bl,0x14
00000008 4829DCsub rsp,rbx
0000000B 488D1424 lea rdx,[rsp]
0000000F 488D742404 lea rsi,[rsp+0x4]
00000014 6A34push byte +0x34
00000016 58pop rax
00000017 0F05syscall
00000019 48FFC7inc rdi
0000001C 66817E024A67 cmp word [rsi+0x2],0x674a
00000022 75F0jnz 0x14
00000024 48FFCFdec rdi
00000027 6A02push byte +0x2
00000029 5Epop rsi
0000002A 6A21push byte +0x21
0000002C 58pop rax
0000002D 0F05syscall
0000002F 48FFCEdec rsi
00000032 79F6jns 0x2a
00000034 4889F3mov rbx,rsi
00000037 BB412F7368 mov ebx,0x68732f41
0000003C B82F62696E mov eax,0x6e69622f
00000041 48C1EB08 shr rbx,byte 0x8
00000045 48C1E320 shl rbx,byte 0x20
00000049 4809D8or rax,rbx
0000004C 50push rax
0000004D 4889E7mov rdi,rsp
00000050 4831F6xor rsi,rsi
00000053 4889F2mov rdx,rsi
00000056 6A3Bpush byte +0x3b
00000058 58pop rax
00000059 0F05syscall
ghidrabook#
```

Гибкость потокового дизассемблирования часто оказывается полезной. Одна из таких ситуаций – анализ сетевых атак, в которых сетевые пакеты могут содержать шелл-код. Для анализа поведения вредоносной полезной нагрузки в таком пакете можно воспользоваться потоковым дизассемблером. Другая ситуация – анализ образов ПЗУ, для которых недоступна документация по размещению в памяти. В одних частях ПЗУ

могут находиться данные, а в других код. С помощью потокового дизассемблера можно дизассемблировать именно те части образа, которые предположительно содержат код.

## РЕЗЮМЕ

Рассмотренные в этой главе инструменты необязательно лучшие в своем роде. Зато они доступны любому, кто хочет заняться обратной разработкой двоичных файлов. А еще важнее то, что именно эти инструменты побудили заняться разработкой Ghidra. В последующих главах мы иногда будем описывать автономные инструменты, предлагающие функциональность, аналогичную той, что включена в Ghidra. Осведомленность о таких инструментах позволит вам лучше понять пользовательский интерфейс Ghidra в целом и входящие в его состав информационные окна.



# 3

## ПЕРВОЕ ЗНАКОМСТВО С GHIDRA



Ghidra – свободно доступный инструмент обратной разработки (software reverse engineering – SRE) с открытым исходным кодом, разработанный Агентством национальной безопасности США (АНБ). Платформенно независимая среда Ghidra включает интерактивный дизассемблер и декомпилятор, а также множество других взаимосвязанных инструментов, помогающих анализировать код. Она поддерживает наборы команд разной архитектуры, различные форматы исполняемых файлов и может работать как в автономном, так и в коллективном режиме. Быть может, лучшей особенностью Ghidra является возможность настраивать рабочую среду и разрабатывать плагины и скрипты SRE в собственных интересах, а затем делиться своими находками со всем сообществом Ghidra.

# ЛИЦЕНЗИОННАЯ ПОЛИТИКА GHIDRA

Ghidra распространяется бесплатно на условиях лицензии Apache License, версия 2.0. Эта лицензия предоставляет физическим лицам широкие права по использованию Ghidra, но накладывает некоторые ограничения. Любой человек, который скачивает, использует или редактирует Ghidra, должен прочитать соглашение с пользователями (*docs/UserAgreement.html*), а также файлы, описывающие условия лицензирования, в каталогах *GPL* и *licenses*, и убедиться, что выполняет все прописанные соглашения, поскольку сторонние компоненты, входящие в Ghidra, имеют свои лицензии. На случай, если вы забыли о чем-то, упомянутом в этом абзаце, Ghidra любезно отображает сведения о лицензировании при каждом запуске или выборе пункта About Ghidra из меню **Help**.

## ВЕРСИИ GHIDRA

Ghidra доступна для Windows, Linux и macOS. Хотя Ghidra конфигурируется в широких пределах, большинство новых пользователей, скорее всего, скачают последнюю версию Ghidra Core, которая включает традиционную функциональность обратной разработки. В этой книге описывается в основном функционал Ghidra Core для индивидуальных проектов. Кроме того, мы уделим некоторое время обсуждению режимов совместной работы и работы в необслуживаемом режиме, а также конфигураций Developer (Для разработчиков), Function ID (Идентификатор функции) и Experimental (Экспериментальная).

## РЕСУРСЫ ПОДДЕРЖКИ GHIDRA

Работа с новой программой часто пугает, особенно когда требуется решить трудную реальную проблему с помощью обратной разработки. Как пользователю (или потенциальному разработчику) Ghidra вам наверняка интересно, куда обратиться за помощью, если возникнут вопросы. Если мы хорошо справились со своей работой, то во многих ситуациях

этой книги будет достаточно. Но на случай, если вам понадобится дополнительная помощь, подскажем, какие имеются ресурсы.

**Официальная документация.** Ghidra содержит подробную справочную систему, в которую можно попасть из меню или нажатием клавиши **F1**. Система предлагает иерархическое меню, а также средства поиска. Но в настоящее время не поддерживаются запросы в свободной форме типа «Как сделать *x*?».

**Файлы readme.** Иногда меню **Help** отсылает к дополнительному материалу по теме, например файлу *readme*. В документацию включено много таких файлов, которые сопровождают плагины, дополняют материал, представленный в меню **Help** (например, *support / analyzeHeadlessREADME.html*), помогают выполнять установку (*docs /InstallationGuide.html*) и служат подспорьем разработчику (например, *Extensions/Eclipse/GhidraDev/GhidraDev\_README.html*), если вы решите пойти по этому пути (и, чем черт не шутит, написать поддержку вопросов в свободной форме).

**Сайт Ghidra.** На домашней странице проекта Ghidra (<https://www.ghidra-sre.org/>) имеются предложения для потенциальных пользователей, текущих пользователей, разработчиков и соавторов, помогающие каждой категории расширить свои знания о Ghidra. Помимо подробной информации о скачивании каждого выпуска Ghidra, выложено полезное видео, описывающее процедуру установки по шагам.

**Каталог docs.** В состав установки Ghidra входит каталог, содержащий полезную документацию, в т. ч. допускающее распечатку руководство по меню и горячим клавишам (*docs/CheatSheet.html*), которое здорово поможет при первом знакомстве с Ghidra, а также многое другое. В каталоге *docs/GhidraClass* вы найдете пособия для пользователей Ghidra начального, среднего и продвинутого уровней.

## СКАЧИВАНИЕ GHIDRA

Для получения копии Ghidra нужно сделать три простых шага.

1. Перейдите на сайт <https://ghidra-sre.org/>.
2. Нажмите большую красную кнопку **Download Ghidra**.
3. Сохраните файл на своем компьютере.

Как часто бывает с простыми трехшаговыми процессами, есть пара мест, где упрямый ослушник может предпочесть немного отклониться от рекомендованного пути. Следующие пункты адресованы тем из вас, кто хочет чего-нибудь, отличного от традиционного меню.

- ▶ Если вы хотите установить другую версию, просто нажмите кнопку **Releases** – и вам будет предложено скачать другие выпускные версии. В каких-то моментах функциональность может отличаться, но в основе своей это все также Ghidra.
- ▶ Если вы хотите установить сервер для коллективной работы, подождите до главы 11, где описано, как внести это важное изменение в процедуру установки (или на свой страх и риск попробуйте воспользоваться информацией в каталоге *server*). В худшем случае будет несложно откатиться назад, начать с первого шага и установить локальный экземпляр Ghidra.
- ▶ Сильные духом могут попробовать собрать Ghidra из исходников. Исходный код Ghidra имеется на GitHub по адресу <https://github.com/NationalSecurityAgency/ghidra/>.

А мы продолжим рассказ о традиционном процессе установки.

## УСТАНОВКА GHIDRA

Итак, что произошло, когда вы нажали волшебную красную кнопку и выбрали место на своем компьютере? Если все пройдет по плану, то в выбранном каталоге должен появиться zip-файл. Для самой первой версии Ghidra файл назывался *ghidra\_9.0\_PUBLIC\_20190228.zip*.

Zip-файл представляет собой архив, содержащий больше 3400 файлов, составляющих каркас Ghidra. Если вас устраива-

ет место, куда был сохранен файл, распакуйте его (например, в Windows щелкните по нему правой кнопкой мыши и выберите из контекстного меню пункт **Извлечь все**), и вы получите доступ к иерархии каталогов Ghidra. Отметим, что Ghidra должна откомпилировать кое-какие внутренние файлы данных, поэтому пользователю понадобится доступ к подкаталогам с правом записи.

## Структура каталогов *Ghidra*

Близкое знакомство с содержимым установочного каталога Ghidra необязательно – начать работать можно и так. Но уж коль скоро мы заговорили о распаковке дистрибутива, кинем взгляд на структуру каталогов. Эти знания пригодятся позже, когда мы перейдем к продвинутым средствам Ghidra. Ниже приведено краткое описание всех подкаталогов, а на рис. 3.1 показан их перечень.



Рис. 3.1. Структура каталогов *Ghidra*

**docs.** Содержит общую документацию по Ghidra и работе с ней. Внутри этого каталога есть два достойных упоминания подкаталога. Во-первых, в подкаталоге *GhidraClass* находятся дополнительные учебные материалы, которые помогут вам больше узнать о Ghidra. Во-вторых, в подкаталоге *languages* описывается язык спецификации процессоров SLEIGH. Мы будем подробно обсуждать его в главе 18.

***Extensions.*** Содержит полезные уже собранные расширения, а также данные и информацию, важную для написания расширений Ghidra. Этот каталог подробно рассматривается в главах 15, 17 и 18.

***Ghidra.*** Содержит код Ghidra. О том, что здесь находится, мы подробно расскажем, когда начнем настраивать Ghidra в главе 12 и расширять ее возможности в главах 13–18.

***GPL.*** Некоторые компоненты были разработаны не командой Ghidra, но включают код, распространяемый по лицензии GNU General Public License (GPL). В каталоге *GPL* находятся относящиеся к таким компонентам файлы, а также информация о лицензии.

***licenses.*** Содержит файлы, описывающие юридические условия использования сторонних компонент Ghidra.

***server.*** Поддержка установки сервера Ghidra, необходимо для коллективной работы над SRE. Этот каталог подробно обсуждается в главе 11.

***support.*** Сборная солянка – различные специальные возможности и средства Ghidra. В качестве бонуса прилагается значок Ghidra (*ghidra.ico*), если вы захотите продолжить настройку своей среды (например, создать ярлык, указывающий на скрипт запуска Ghidra). Этот каталог будет обсуждаться по мере необходимости в разных частях книги.

## Запуск Ghidra

Наряду с каталогами, в корневом каталоге есть файлы, которые помогут начать путешествие в мир Ghidra и SRE. Здесь имеется еще один файл лицензии (*LICENSE.txt*), но главное – скрипты запуска Ghidra. После первого двойного щелчка по файлу *ghidraRun.bat* (или его эквивалента *ghidraRun* в Linux или macOS) нужно будет принять лицензионное соглашение с конечным пользователем (EULA), показанное на рис. 3.2, подтвердив тем самым, что планируемое использование Ghidra не противоречит соглашению с пользователем (Ghidra User Agreement). При последующих запусках это окно уже не будет показываться, но его всегда можно вызвать из меню **Help**.

Кроме того, может быть задан вопрос о пути к установке Java. (Если на вашем компьютере Java не установлен, см. «Руководство по установке» в подкаталоге *docs*, где имеется необходимая документация в разделе «Java Notes».) Для Ghidra необходима версия Java Development Kit (JDK) 11 или старше.

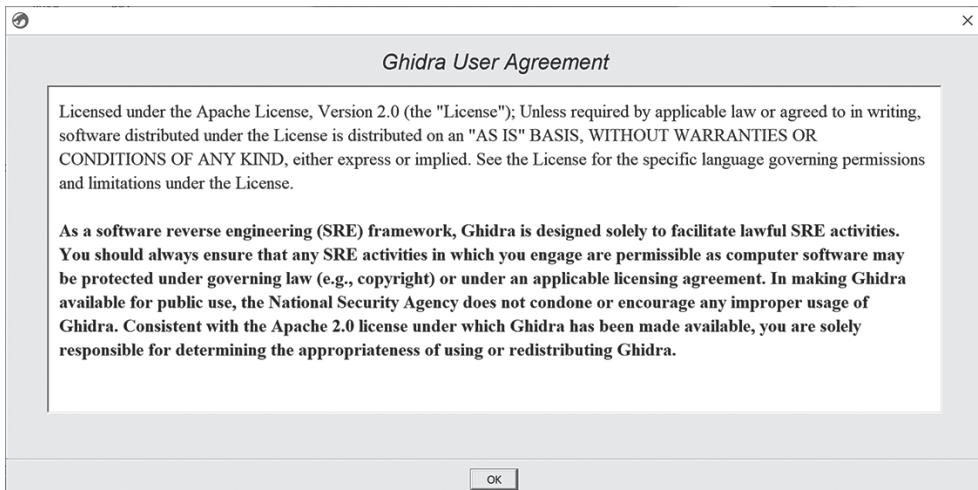


Рис. 3.2. Соглашение с пользователем Ghidra

## РЕЗЮМЕ

Успешно открыв Ghidra, вы готовы совершить что-нибудь полезное. В нескольких следующих главах вы узнаете, как использовать Ghidra для начального анализа файла, как работать с браузером кода и различными информационными окнами, а также научитесь конфигурировать эти окна и работать с ними, чтобы лучше понять поведение программы.



# **Часть II**

## **Основы**

## **использования**

## **Ghidra**



# 4

## НАЧАЛО РАБОТЫ С GHIDRA



Настало время немного поработать с Ghidra. Далее в этой книге мы будем изучать различные средства Ghidra и как ими пользоваться, чтобы наилучшим способом решить свои задачи обратной разработки. Начнем мы с того, что предстает взору сразу после запуска Ghidra, а затем опишем, что происходит, когда мы открываем один двоичный файл для анализа. И в конце главы дадим краткий обзор пользовательского интерфейса, чтобы заложить фундамент для последующих глав.

## ЗАПУСК GHIDRA

При каждом запуске Ghidra вы в течение короткого времени будете видеть заставку, содержащую логотип Ghidra, информацию о сборке, номера версий Ghidra и Java и сведения о лицензии. Если вам захочется внимательно прочитать, что написано на заставке, и, в частности, больше узнать об используемых версиях, то ее можно будет в любой момент вывести на экран, выбрав пункт **Help > About Ghidra** в окне проекта