УДК 004.451.5
ББК 32.371
Э64

Эндриесс Д.

Э64 Практический анализ двоичных файлов / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 460 с.: ил.

ISBN 978-5-97060-978-1

В книге представлено подробное описание методов и инструментов, необходимых для анализа двоичного кода, который позволяет убедиться, что откомпилированная программа работает так же, как исходная, написанная на языке высокого уровня.

Наряду с базовыми понятиями рассматриваются такие темы, как оснащение двоичной программы, динамический анализ заражения и символическое выполнение. В каждой главе приводится несколько примеров кода; к книге прилагается сконфигурированная виртуальная машина, включающая все примеры.

Руководство адресовано специалистам по безопасности и тестированию на проникновение, хакерам, аналитикам вредоносных программ и всем, кто интересуется вопросами защиты ПО.

УДК 004.451.5
ББК 32.371

Title of English-language original: Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly Reversing Modern Malware and Next Generation Threats, ISBN 9781593279127, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Russian-Language 1st edition Copyright © 2021 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 978-1-59327-912-7 (англ.)
ISBN 978-5-97060-978-1 (рус.)

© Dennis Andriesse, 21921
© Перевод, оформление, издание, ДМК Пресс, 2021

ОГЛАВЛЕНИЕ

Вступительное слово 17
Предисловие 20
Благодарности 21
Введение 22

ЧАСТЬ I. ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

Глава 1. Анатомия двоичного файла 32
Глава 2. Формат ELF 52
Глава 3. Формат PE: краткое введение 78
Глава 4. Создание двоичного загрузчика с применением libbfd 88

ЧАСТЬ II. ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

Глава 5. Основы анализа двоичных файлов в Linux 109
Глава 6. Основы дизассемблирования и анализа двоичных файлов 135
Глава 7. Простые методы внедрения кода для формата ELF 178

ЧАСТЬ III. ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ

Глава 8. Настройка дизассемблирования 212
Глава 9. Оснащение двоичных файлов 244
Глава 10. Принципы динамического анализа заражения 289
Глава 11. Практический динамический анализ заражения с помощью libdft 305
Глава 12. Принципы символического выполнения 335
Глава 13. Практическое символическое выполнение с помощью Triton 361

ЧАСТЬ IV. ПРИЛОЖЕНИЯ

Приложение A. Краткий курс ассемблера x86 402
Приложение B. Реализация перезаписи PT_NOTE с помощью libelf 422
Приложение C. Перечень инструментов анализа двоичных файлов 443
Приложение D. Литература для дополнительного чтения 447

Предметный указатель 451

СОДЕРЖАНИЕ

Вступительное слово 17

Предисловие 20

Благодарности 21

Введение 22

ЧАСТЬ I. ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

Глава 1. Анатомия двоичного файла 32

1.1 Процесс компиляции программы на C 33
  1.1.1 Этап препроцессирования 33
  1.1.2 Этап компиляции 35
  1.1.3 Этап ассемблирования 37
  1.1.4 Этап компоновки 38
1.2 Символы и зачищенные двоичные файлы 40
  1.2.1 Просмотр информации о символах 40
  1.2.2 Переход на темную сторону: зачистка двоичного файла 42
1.3 Дизассемблирование двоичного файла 42
  1.3.1 Заглянем внутрь объектного файла 43
  1.3.2 Изучение полного исполняемого двоичного файла 45
1.4 Загрузка и выполнение двоичного файла 48
1.5 Резюме 50

Глава 2. Формат ELF 52

2.1 Заголовок исполняемого файла 54
  2.1.1 Массив e_ident 55
  2.1.2 Поля e_type, e_machine и e_version 56

2.1.3 Поле e_entry ... 57
2.1.4 Поля e_phoff и e_shoff ... 57
2.1.5 Поле e_flags ... 57
2.1.6 Поле e_ehsize ... 58
2.1.7 Поля e_*entsize и e_*num ... 58
2.1.8 Поле e_shstrndx ... 58
2.2 Заголовки секций ... 59
2.2.1 Поле sh_name ... 60
2.2.2 Поле sh_type ... 60
2.2.3 Поле sh_flags ... 61
2.2.4 Поля sh_addr, sh_offset и sh_size ... 61
2.2.5 Поле sh_link ... 62
2.2.6 Поле sh_info ... 62
2.2.7 Поле sh_addralign ... 62
2.2.8 Поле sh_entsize ... 62
2.3 Секции ... 62
2.3.1 Секции .init и .fini ... 64
2.3.2 Секция .text ... 64
2.3.3 Секции .bss, .data и .rodata ... 66
2.3.4 Позднее связывание и секции .plt, .got, .got.plt ... 66
2.3.5 Секции .rel.* и .rela.* ... 70
2.3.6 Секция .dynamic ... 71
2.3.7 Секции .init_array и .fini_array ... 72
2.3.8 Секции .shstrtab, .symtab, .strtab, .dynsym и .dynstr ... 73
2.4 Заголовки программы ... 74
2.4.1 Поле p_type ... 75
2.4.2 Поле p_flags ... 76
2.4.3 Поля p_offset, p_vaddr, p_paddr, p_filesz и p_memsz ... 76
2.4.4 Поле p_align ... 76
2.5 Резюме ... 77

Глава 3. Формат PE: краткое введение ... 78

3.1 Заголовок MS-DOS и заглушка MS-DOS ... 79
3.2 Сигнатура PE, заголовок PE-файла и факультативный заголовок PE ... 79
3.2.1 Сигнатура PE ... 82
3.2.2 Заголовок PE-файла ... 82
3.2.3 Факультативный заголовок PE ... 83
3.3 Таблица заголовков секций ... 83
3.4 Секции ... 84
3.4.1 Секции .edata и .idata ... 85
3.4.2 Заполнение в секциях кода PE ... 86
3.5 Резюме ... 86

Глава 4. Создание двоичного загрузчика с применением libbfd ... 88

4.1 Что такое libbfd? ... 89
4.2 Простой интерфейс загрузки двоичных файлов ... 89

4.2.1 Класс Binary ... 92
4.2.2 Класс Section ... 92
4.2.3 Класс Symbol ... 92
4.3 Реализация загрузчика двоичных файлов ... 93
4.3.1 Инициализация libbfd и открытие двоичного файла ... 94
4.3.2 Разбор основных свойств двоичного файла ... 96
4.3.3 Загрузка символов ... 99
4.3.4 Загрузка секций ... 102
4.4 Тестирование загрузчика двоичных файлов ... 104
4.5 Резюме ... 106

ЧАСТЬ II. ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

Глава 5. Основы анализа двоичных файлов в Linux ... 109
5.1 Разрешение кризиса самоопределения с помощью file ... 110
5.2 Использование ldd для изучения зависимостей ... 113
5.3 Просмотр содержимого файла с помощью xxd ... 115
5.4 Разбор выделенного заголовка ELF с помощью readelf ... 117
5.5 Разбор символов с помощью nm ... 119
5.6 Поиск зацепок с помощью strings ... 122
5.7 Трассировка системных и библиотечных вызовов с помощью strace и ltrace ... 125
5.8 Изучение поведения на уровне команд с помощью objdump ... 129
5.9 Получение буфера динамической строки с помощью gdb ... 131
5.10 Резюме ... 134

Глава 6. Основы дизассемблирования и анализа двоичных файлов ... 135
6.1 Статическое дизассемблирование ... 136
6.1.1 Линейное дизассемблирование ... 136
6.1.2 Рекурсивное дизассемблирование ... 139
6.2 Динамическое дизассемблирование ... 142
6.2.1 Пример: трассировка выполнения двоичного файла в gdb ... 143
6.2.2 Стратегии покрытия кода ... 146
6.3 Структурирование дизассемблированного кода и данных ... 150
6.3.1 Структурирование кода ... 151
6.3.2 Структурирование данных ... 158
6.3.3 Декомпиляция ... 160
6.3.4 Промежуточные представления ... 162
6.4 Фундаментальные методы анализа ... 164
6.4.1 Свойства двоичного анализа ... 164
6.4.2 Анализ потока управления ... 169
6.4.3 Анализ потока данных ... 171
6.5 Влияние настроек компилятора на результат дизассемблирования ... 175
6.6 Резюме ... 177

Глава 7. Простые методы внедрения кода для формата ELF 178
7.1 Прямая модификация двоичного файла с помощью шестнадцатеричного редактирования 178
7.1.1 Ошибка на единицу в действии 179
7.1.2 Исправление ошибки на единицу 182
7.2 Модификация поведения разделяемой библиотеки с помощью LD_PRELOAD 186
7.2.1 Уязвимость, вызванная переполнением кучи 186
7.2.2 Обнаружение переполнения кучи 189
7.3 Внедрение секции кода 192
7.3.1 Внедрение секции в ELF-файл: общий обзор 192
7.3.2 Использование elfinject для внедрения секции в ELF-файл 195
7.4 Вызов внедренного кода 198
7.4.1 Модификация точки входа 199
7.4.2 Перехват конструкторов и деструкторов 202
7.4.3 Перехват записей GOT 205
7.4.4 Перехват записей PLT 208
7.4.5 Перенаправление прямых и косвенных вызовов 209
7.5 Резюме 210

ЧАСТЬ III. ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ

Глава 8. Настройка дизассемблирования 212
8.1 Зачем писать специальный проход дизассемблера? 213
8.1.1 Пример специального дизассемблирования: обфусцированный код 213
8.1.2 Другие причины для написания специального дизассемблера 216
8.2 Введение в Capstone 217
8.2.1 Установка Capstone 218
8.2.2 Линейное дизассемблирование с помощью Capstone 219
8.2.3 Изучение Capstone C API 224
8.2.4 Рекурсивное дизассемблирование с помощью Capstone 225
8.3 Реализация сканера ROP-гаджетов 234
8.3.1 Введение в возвратно-ориентированное программирование 234
8.3.2 Поиск ROP-гаджетов 236
8.4 Резюме 242

Глава 9. Оснащение двоичных файлов 244
9.1 Что такое оснащение двоичного файла? 244
9.1.1 API оснащения двоичных файлов 245
9.1.2 Статическое и динамическое оснащение двоичных файлов 246
9.2 Статическое оснащение двоичных файлов 248
9.2.1 Подход на основе int 3 248
9.2.2 Подход на основе трамплинов 250

9.3 Динамическое оснащение двоичных файлов 255
9.3.1 Архитектура DBI-системы 255
9.3.2 Введение в Pin 257
9.4 Профилирование с помощью Pin 259
9.4.1 Структуры данных профилировщика и код инициализации 259
9.4.2 Разбор символов функций 262
9.4.3 Оснащение простых блоков 264
9.4.4 Оснащение команд управления потоком 266
9.4.5 Подсчет команд, передач управления и системных вызовов 269
9.4.6 Тестирование профилировщика 270
9.5 Автоматическая распаковка двоичного файла с помощью Pin 274
9.5.1 Введение в упаковщики исполняемых файлов 274
9.5.2 Структуры данных и код инициализации распаковщика 276
9.5.3 Оснащение команд записи в память 278
9.5.4 Оснащение команд управления потоком 280
9.5.5 Отслеживание операций записи в память 280
9.5.6 Обнаружение оригинальной точки входа и запись распакованного двоичного файла 281
9.5.7 Тестирование распаковщика 283
9.6 Резюме 287

Глава 10. Принципы динамического анализа заражения 289
10.1 Что такое DTA? 290
10.2 Три шага DTA: источники заражения, приемники заражения и распространение заражения 290
10.2.1 Определение источников заражения 291
10.2.2 Определение приемников заражения 291
10.2.3 Прослеживание распространения заражения 292
10.3 Использование DTA для обнаружения дефекта Heartbleed 292
10.3.1 Краткий обзор уязвимости Heartbleed 292
10.3.2 Обнаружение Heartbleed с помощью заражения 294
10.4 Факторы проектирования DTA: гранулярность, цвета политики заражения 296
10.4.1 Гранулярность заражения 296
10.4.2 Цвета заражения 297
10.4.3 Политики распространения заражения 298
10.4.4 Сверхзаражение и недозаражение 300
10.4.5 Зависимости по управлению 300
10.4.6 Теневая память 302
10.5 Резюме 304

Глава 11. Практический динамический анализ заражения с помощью libdft 305
11.1 Введение в libdft 305
11.1.1 Внутреннее устройство libdft 306
11.1.2 Политика заражения 309

11.2 Использование DTA для обнаружения удаленного перехвата управления 310
11.2.1 Проверка информации о заражении 313
11.2.2 Источники заражения: заражение принятых байтов 315
11.2.3 Приемники заражения: проверка аргументов execve 317
11.2.4 Обнаружение попытки перехвата потока управления 318
11.3 Обход DTA с помощью неявных потоков 323
11.4 Детектор утечки данных на основе DTA 324
11.4.1 Источники заражения: прослеживание заражения для открытых файлов 327
11.4.2 Приемники заражения: мониторинг отправки по сети на предмет утечки данных 330
11.4.3 Обнаружение потенциальной утечки данных 331
11.5 Резюме 334

Глава 12. Принципы символического выполнения 335

12.1 Краткий обзор символического выполнения 336
12.1.1 Символическое и конкретное выполнение 336
12.1.2 Варианты и ограничения символического выполнения 340
12.1.3 Повышение масштабируемости символического выполнения 347
12.2 Удовлетворение ограничений с помощью Z3 349
12.2.1 Доказательство достижимости команды 350
12.2.2 Доказательство недостижимости команды 354
12.2.3 Доказательство общезначимости формулы 354
12.2.4 Упрощение выражений 356
12.2.5 Моделирование ограничений для машинного кода с битовыми векторами 356
12.2.6 Решение непроницаемого предиката над битовыми векторами 358
12.3 Резюме 359

Глава 13. Практическое символическое выполнение с помощью Triton 361

13.1 Введение в Triton 362
13.2 Представление символического состояния абстрактными синтаксическими деревьями 363
13.3 Обратное нарезание с помощью Triton 365
13.3.1 Заголовочные файлы и конфигурирование Triton 368
13.3.2 Конфигурационный файл символического выполнения 369
13.3.3 Эмуляция команд 370
13.3.4 Инициализация архитектуры Triton 372
13.3.5 Вычисления обратного среза 373
13.4 Использование Triton для увеличения покрытия кода 374
13.4.1 Создание символических переменных 376
13.4.2 Нахождение модели для нового пути 377

13.4.3 Тестирование инструмента покрытия кода ... 380
13.5 Автоматическая эксплуатация уязвимости ... 384
13.5.1 Уязвимая программа ... 385
13.5.2 Нахождение адреса уязвимой команды вызова ... 388
13.5.3 Построение генератора эксплойта ... 390
13.5.4 Получение оболочки с правами root ... 397
13.6 Резюме ... 400

ЧАСТЬ IV. ПРИЛОЖЕНИЯ

Приложение A. Краткий курс ассемблера x86 ... 402
A.1 Структура ассемблерной программы ... 403
A.1.1 Ассемблерные команды, директивы, метки и комментарии ... 404
A.1.2 Разделение данных и кода ... 405
A.1.3 Синтаксис AT&T и Intel ... 405
A.2 Структура команды x86 ... 405
A.2.1 Ассемблерное представление команд x86 ... 406
A.2.2 Структура команд x86 на машинном уровне ... 406
A.2.3 Регистровые операнды ... 407
A.2.4 Операнды в памяти ... 409
A.2.5 Непосредственные операнды ... 410
A.3 Употребительные команды x86 ... 410
A.3.1 Сравнение операндов и установки флагов состояния ... 411
A.3.2 Реализация системных вызовов ... 412
A.3.3 Реализация условных переходов ... 412
A.3.4 Загрузка адресов памяти ... 413
A.4 Представление типичных программных конструкций на языке ассемблера ... 413
A.4.1 Стек ... 413
A.4.2 Вызовы функций и кадры функций ... 414
A.4.3 Условные предложения ... 419
A.4.4 Циклы ... 420

Приложение B. Реализация перезаписи PT_NOTE с помощью libelf ... 422
B.1 Обязательные заголовки ... 423
B.2 Структуры данных, используемые в elfinject ... 423
B.3 Инициализация libelf ... 425
B.4 Получение заголовка исполняемого файла ... 428
B.5 Нахождение сегмента PT_NOTE ... 429
B.6 Внедрение байтов кода ... 430
B.7 Выравнивание адреса загрузки внедренной секции ... 431
B.8 Перезапись заголовка секции .note.ABI-tag ... 432
B.9 Задание имени внедренной секции ... 437
B.10 Перезапись заголовка программы PT_NOTE ... 439
B.11 Модификация точки входа ... 441

Приложение C. Перечень инструментов анализа двоичных файлов ... 443

C.1 Дизассемблеры ... 443
C.2 Отладчики ... 445
C.3 Каркасы дизассемблирования ... 445
C.4 Каркасы двоичного анализа ... 446

Приложение D. Литература для дополнительного чтения ... 447

D.1 Стандарты и справочные руководства ... 447
D.2 Статьи ... 448
D.3 Книги ... 450

Предметный указатель ... 451

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте *www.dmkpress.com* на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и No Starch Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Дэннис Эндриесс имеет степень доктора по безопасности систем и сетей, и анализ двоичных файлов – неотъемлемая часть его исследований. Он один из основных соразработчиков системы целостности потока управления PathArmor, которая защищает от атак с перехватом потока управления типа возвратно-ориентированного программирования (ROP). Также Эндриесс принимал участие в разработке атаки, которая положила конец P2P-сети ботов GameOver Zeus.

О технических рецензентах

Торстен Хольц (Thorsten Holz) – профессор факультета электротехники и информационных технологий в Рурском университете в Бохуме, Германия. Он занимается исследованиями в области технических аспектов безопасности систем. В настоящее время его интересуют обратная разработка, автоматизированное обнаружение уязвимостей и изучение последних векторов атак.

Tim Vidas – хакер-ас. На протяжении многих лет он возглавлял инфраструктурную команду в соревновании DARPA CGC, внедрял инновации в компании Dell Secureworks и курировал исследовательскую группу CERT в области компьютерно-технической экспертизы. Он получил степень доктора в университете Карнеги-Меллона, является частым участником и докладчиком на конференциях и обладает числом Эрдёша–Бейкона 4-3. Много времени уделяет обязанностям отца и мужа.

ВСТУПИТЕЛЬНОЕ СЛОВО

В наши дни нетрудно найти книги по языку ассемблера и даже описания двоичных форматов ELF и PE. Растет количество статей, посвященных прослеживанию потока информации и символическому выполнению. Но нет еще ни одной книги, которая вела бы читателя от основ ассемблера к выполнению сложного анализа двоичного кода. Нет ни одной книги, которая научила бы читателя оснащать двоичные программы инструментальными средствами, применять динамический анализ заражения (taint analysis) для прослеживания путей прохождений интересных данных по программе или использовать символическое выполнение в целях автоматизированного генерирования эксплойтов. Иными словами, нет книг, которые учили бы методам, инструментам и образу мыслей, необходимым для анализа двоичного кода. Точнее, до сих пор не было.

Трудность анализа двоичного кода состоит в том, что нужно разбираться в куче разных вещей. Да, конечно, нужно знать язык ассемблера, но, кроме того, понимать форматы двоичных файлов, механизмы компоновки и загрузки, принципы статического и динамического анализов, расположение программы в памяти, соглашения, поддерживаемые компиляторами, – и это только начало. Для конкретных задач анализа или оснащения могут понадобиться и более специальные знания. Конечно, для всего этого нужны инструменты. Многих эта перспектива настолько пугает, что они сдаются, даже не вступив в борьбу. Так много всего предстоит учить. С чего же начать?

Ответ: с этой книги. Здесь все необходимое излагается последовательно, логично и доступно. И интересно к тому же! Даже если вы ничего не знаете о том, как выглядит двоичная программа, как она

загружается и что происходит во время ее выполнения, в книге вы найдете отлично продуманное введение во все эти понятия и инструменты, с помощью которых сможете не только быстро освоить теоретические основы, но и поэкспериментировать в реальных ситуациях. На мой взгляд, это единственный способ приобрести глубокие знания, которые не забудутся на следующий день.

Но и в том случае, если у вас есть богатый опыт анализа двоичного кода с помощью таких инструментов, как Capstone, Radare, IDA Pro, OllyDbg или что там у вас стоит на первом месте, вы найдете здесь материал по душе. В поздних главах описываются продвинутые методики создания таких изощренных инструментов анализа и оснащения, о существовании которых вы даже не подозревали.

Анализ и оснащение двоичного кода – увлекательные, но трудные техники, которыми в совершенстве владеют лишь немногие опытные хакеры. Но внимание к вопросам безопасности растет, а вместе с ним и важность этих вопросов. Мы должны уметь анализировать вредоносные программы, чтобы понимать, что они делают и как им в этом воспрепятствовать. Но поскольку все больше вредоносных программ применяют методы обфускации и приемы противодействия анализу, нам необходимы более хитроумные методы.

Мы также все чаще анализируем и оснащаем вполне благопристойные программы, например, чтобы затруднить атаки на них. Так, может возникнуть желание модифицировать существующий двоичный код программы, написанной на C++, с целью гарантировать, что все вызовы (виртуальных) функций обращаются только к допустимым методам. Для этого мы должны сначала проанализировать двоичный код и найти в нем методы и вызовы функций. Затем нужно добавить оснащение, сохранив при этом семантику оригинальной программы. Все это проще сказать, чем сделать.

Многие начинают изучать такие методы, столкнувшись с интересной задачкой, оказавшейся не по зубам. Это может быть что угодно: как превратить игровую консоль в компьютер общего назначения, как взломать какую-то программу или понять, как работает вредонос, проникший в ваш компьютер.

К своему стыду, должен сознаться, что лично для меня все началось с желания снять защиту от копирования с видеоигр, покупка которых была мне не по средствам. Поэтому я выучил язык ассемблера и стал искать проверки в двоичных файлах. Тогда на рынке правил бал 8-разрядный процессор 6510 с аккумулятором и двумя регистрами общего назначения. Хотя для того чтобы использовать все 64 КБ системной памяти, требовалось исполнять танцы с бубнами, в целом система была простой. Но поначалу все было непонятно. Со временем я набирался ума от более опытных друзей, и постепенно туман стал рассеиваться. Маршрут был, без сомнения, интересным, но долгим, трудным и иногда заводил в тупик. Многое я бы отдал тогда за путеводитель! Современные 64-разрядные процессоры x86 не в пример сложнее, как и компиляторы, которые генерируют двоичный код. Понять, что делает код, гораздо труднее, чем раньше. Специалист, кото-

рый покажет путь и прояснит сложные вопросы, которые вы могли бы упустить из виду, сделает путешествие короче и интереснее, превратив его в истинное удовольствие.

Дэннис Эндриесс – эксперт по анализу двоичного кода и в доказательство может предъявить степень доктора в этой области – буквально. Однако он не академический ученый, публикующий статьи для себе подобных. Его работы по большей части сугубо практические. Например, он был в числе тех немногих людей, кто сумел разобраться в коде печально известной сети ботов GameOver Zeus, ущерб от которой оценивается в 100 миллионов долларов. И более того, он вместе с другими экспертами безопасности принимал участие в возглавляемой ФБР операции, положившей конец деятельности этой сети. Работая с вредоносными программами, он на практике имел возможность оценить сильные стороны и ограничения имеющихся средств анализа двоичного кода и придумал, как их улучшить. Новаторские методы дизассемблирования, разработанные Дэннисом, теперь включены в коммерческие продукты, в частности Binary Ninja.

Но быть экспертом еще недостаточно. Автор книги должен еще уметь излагать свои мысли. Дэннис Эндриесс обладает этим редким сочетанием талантов: он эксперт по анализу двоичного кода, способный объяснить самые сложные вещи простыми словами, не упуская сути. У него приятный стиль, а примеры ясные и наглядные.

Лично я давно хотел иметь такую книгу. В течение многих лет я читаю курс по анализу вредоносного ПО в Амстердамском свободном университете без учебника ввиду отсутствия такового. Мне приходилось использовать разнообразные онлайновые источники, пособия и эклектичный набор слайдов. Когда студенты спрашивали, почему бы не использовать печатный учебник (как они привыкли), я отвечал, что хорошего учебника по анализу двоичного кода не существует, но если у меня когда-нибудь выдастся свободное время, я его напишу. Разумеется, этого не случилось.

Это как раз та книга, которую я надеялся написать, но так и не собрался. И она лучше, чем мог бы написать я сам.

Приятного путешествия.

Герберт Бос

ПРЕДИСЛОВИЕ

Анализ двоичного кода – одна из самых увлекательных и трудных тем в хакинге и информатике вообще. Она трудна и для изучения, в немалой степени из-за отсутствия доступной информации.

В книгах по обратной разработке и анализу вредоносного ПО недостатка нет, но этого не скажешь о таких продвинутых вопросах анализа двоичного кода, как оснащение двоичной программы, динамический анализ заражения или символическое выполнение. Начинающий аналитик вынужден выискивать информацию по темным закоулкам интернета, в устаревших, а иногда и откровенно вводящих в заблуждение сообщениях в форумах или в непонятно написанных статьях. Во многих статьях, а также в академической литературе по анализу двоичного кода предполагаются обширные знания, поэтому изучение предмета по таким источникам становится проблемой курицы и яйца. Хуже того, многие инструменты и библиотеки для анализа плохо или никак не документированы.

Я надеюсь, что эта книга, написанная простым языком, сделает анализ двоичного кода более доступным и станет практическим введением во все важные вопросы данной области. Прочитав эту книгу, вы будете в достаточной степени экипированы, чтобы ориентироваться в быстро меняющемся мире анализа двоичного кода и отважиться на самостоятельные исследования.

Введение

Подавляющее большинство компьютерных программ написаны на языках высокого уровня типа C или C++, которые компьютер не может исполнять непосредственно. Такие программы необходимо откомпилировать, в результате чего создаются *двоичные исполняемые файлы*, содержащие машинный код, – его компьютер уже может выполнить. Но откуда мы знаем, что откомпилированная программа имеет такую же семантику, как исходная? Ответ может разочаровать – *а мы и не знаем*!

Существует семантическая пропасть между языками высокого уровня и двоичным машинным кодом, и как ее преодолеть, знают немногие. Даже программисты в большинстве своем плохо понимают, как их программы работают на самом нижнем уровне, и просто верят, что откомпилированная программа делает то, что они задумали. Поэтому многие дефекты компилятора, тонкие ошибки реализации, потайные ходы на двоичном уровне и другие вредоносные паразиты остаются незамеченными.

Хуже того, существует бесчисленное множество двоичных программ и библиотек – в промышленности, в банках, во встраиваемых системах, – исходный код которых давно утерян или является коммерческой собственностью. Это означает, что такие программы и библиотеки невозможно исправить или хотя бы оценить их безопасность на уровне исходного кода с применением традиционных методов. Это реальная проблема даже для крупных программных компаний, свидетельство чему – недавний выпуск компанией Microsoft созданного с большим трудом двоичного исправления ошибки переполнения буфера в программе «Редактор формул», являющейся частью Microsoft Office[1].

[1] *https://0patch.blogspot.nl/2017/11/did-microsoft-just-manually-patch-their.html*.

В этой книге вы научитесь анализировать и даже модифицировать программы на двоичном уровне. Будь вы хакер, специалист по безопасности, аналитик вредоносного кода, программист или просто любопытствующий, эти методы позволят вам лучше понимать и контролировать двоичные программы, которые вы создаете и используете каждый день.

Что такое анализ двоичных файлов, и зачем он вам нужен?

Анализ двоичных файлов, или просто *двоичный анализ*, – это наука и искусство анализа свойств двоичных компьютерных программ, а также машинного кода и данных, которые они содержат. Короче говоря, цель анализа двоичных файлов – определить (и, возможно, модифицировать) истинные свойства двоичных программ, т. е. понять, что они делают в действительности, а не доверяться тому, что они, по нашему мнению, должны делать.

Многие отождествляют двоичный анализ с обратной разработкой и дизассемблированием, и отчасти они правы. Дизассемблирование – важный первый шаг многих видов двоичного анализа, а обратная разработка – типичное приложение двоичного анализа и зачастую единственный способ документировать поведение проприетарного или вредоносного программного обеспечения. Однако область двоичного анализа гораздо шире.

Методы анализа двоичных файлов можно отнести к одному из двух классов, хотя возможны и комбинации.

Статический анализ В этом случае мы рассуждаем о программе, не выполняя ее. У такого подхода несколько преимуществ: теоретически возможно проанализировать весь двоичный файл за один присест, и для его выполнения не нужен процессор. Например, можно статически проанализировать двоичный файл для процессора ARM на компьютере с процессором x86. Недостаток же в том, что в процессе статического анализа мы ничего не знаем о состоянии выполнения двоичной программы, что сильно затрудняет анализ.

Динамический анализ Напротив, в случае динамического анализа мы запускаем программу и анализируем ее во время выполнения. Часто этот подход оказывается проще статического анализа, потому что нам известно все состояние программы, включая значения переменных и выбор ветвей при условном выполнении. Однако мы видим лишь тот код, который выполняется, поэтому можем пропустить интересные части программы.

И у статического, и у динамического анализов есть плюсы и минусы. В этой книге мы расскажем об обоих направлениях. Помимо пассивного двоичного анализа, вы узнаете о методах *оснащения* дво-

ичных файлов, которые позволяют модифицировать двоичные программы, не имея исходного кода. Оснащение двоичных файлов опирается на такие методы анализа, как дизассемблирование, но и само оно может помочь при проведении двоичного анализа. Из-за такого симбиоза приемов двоичного анализа и оснащения в этой книге рассматриваются как те, так и другие.

Я уже говорил, что анализ можно использовать, чтобы документировать или тестировать на отсутствие уязвимостей программы, для которых у вас нет исходного кода. Но даже если исходный код имеется, такой анализ может быть полезен для поиска тонких ошибок, которые более отчетливо проявляются на уровне двоичного, а не исходного кода. Многие методы анализа двоичных файлов полезны также для отладки. В этой книге рассматриваются методы, применимые во всех этих ситуациях, и не только.

В чем сложность анализа двоичных файлов?

Двоичный анализ – вещь гораздо более трудная, чем эквивалентный анализ на уровне исходного кода. На самом деле многие задачи в этом случае принципиально неразрешимы, т. е. невозможно сконструировать движок, который всегда возвращает правильный результат! Чтобы вы могли составить представление о том, с какими проблемами предстоит столкнуться, ниже приведен список некоторых вещей, затрудняющих анализ двоичных файлов. Увы, этот список далеко не исчерпывающий.

Отсутствует символическая информация В исходном коде, написанном на языке высокого уровня типа C или C++, мы даем осмысленные имена переменным, функциям, классам и т. п. Эти имена мы называем *символической информацией*, или просто *символами*. Если придерживаться хороших соглашений об именовании, то понять исходный код будет гораздо проще, но на двоичном уровне имена не имеют ни малейшего значения. Поэтому из двоичных файлов информация о символах часто удаляется, из-за чего понять код становится гораздо труднее.

Отсутствует информация о типах Еще одна особенность программ на языках высокого уровня – наличие у переменных четко определенных типов, например `int`, `float`, `string` или более сложных структурных типов. На двоичном же уровне типы нигде явно не упоминаются, поэтому понять структуру и назначение данных нелегко.

Отсутствуют высокоуровневые абстракции Современные программы состоят из классов и функций, но компиляторы отбрасывают эти высокоуровневые конструкции. Это означает, что двоичный файл выглядит как огромный «комок» кода и данных, а не хорошо структурированный код, и восстановить высокоуровневую структуру трудно и чревато ошибками.

Код и данные перемешаны Двоичные файлы могут содержать фрагменты данных, перемешанные с исполняемым кодом (и так оно в действительности и есть)[1]. Поэтому очень легко случайно интерпретировать данные как код или наоборот, что приведет к неправильным результатам.

Код и данные зависят от положения Двоичные файлы не рассчитаны на модификацию, поэтому добавление всего одной машинной команды может вызвать проблемы, поскольку следующий за ней код сдвигается, что делает недействительными адреса в памяти и ссылки из других мест кода. Поэтому любое изменение кода и данных чрезвычайно опасно, т. к. программа может вообще перестать работать.

Из-за всех этих проблем на практике нам часто приходится довольствоваться неточными результатами анализа. Важная составная часть анализа двоичных файлов – творчески подойти к созданию полезных инструментов, работающих вопреки ошибкам анализа!

Кому адресована эта книга?

Целевой аудиторией этой книги являются инженеры по безопасности, ученые, занимающиеся исследованиями в области безопасности, хакеры и специалисты по тестированию на проникновение, специалисты по обратной разработке, аналитики вредоносных программ и студенты компьютерных специальностей, интересующиеся анализом двоичных файлов.

Поскольку в книге рассматриваются темы повышенного уровня, предполагаются предварительные знания в области программирования и компьютерных систем. Для получения максимальной пользы от прочтения книги необходимы:

- достаточно свободное владение языками C и C++;
- базовые знания о внутреннем устройстве операционных систем (что такое процесс, что такое виртуальная память и т. д.);
- умение работать с оболочкой Linux (предпочтительно `bash`);
- рабочие знания о языке ассемблера x86/x86-64. Если вы вообще ничего не знаете о языках ассемблера, прочитайте сначала приложение А.

Если вы никогда раньше не программировали или не любите копаться в низкуровневых деталях компьютерных систем, то, вероятно, эта книга не для вас.

---

[1] Одни компиляторы делают это чаще, другие реже. Особенно печальной известностью в этом отношении пользуется Visual Studio, обожающая перемешивать код и данные.

Назначение и структура книги

Главная цель этой книги – сделать из вас разносторонне образованного аналитика двоичных файлов, знакомого как с основными вопросами, так и с такими продвинутыми темами, как оснащение двоичного кода, анализ заражения и символическое выполнение. Эта книга *не претендует* на роль единственного и исчерпывающего источника, поскольку в области двоичного анализа и его инструментария изменения происходят настолько быстро, что любая исчерпывающая книга устарела бы через год. Наша цель – снабдить вас достаточным объемом знаний по всем важным темам, чтобы дальше вы могли двигаться самостоятельно.

С другой стороны, мы не пытаемся разобраться во всех тонкостях обратной разработки кода для процессоров x86 и x86-64 (хотя основные сведения приведены в приложении A) и анализа вредоносных программ на этих платформах. На эти темы уже написано много книг, и дублировать их здесь не имеет смысла. Список книг, посвященных ручной обратной разработке и анализу вредоносного ПО, приведен в приложении D.

Книга состоит из четырех частей.

Часть I. Форматы двоичных файлов. В этой части мы познакомимся с форматами двоичных файлов, без чего невозможно понять последующий материал. Если вы уже знакомы с форматами ELF и PE, а также с библиотекой `libbfd`, то можете пропустить одну или несколько глав в этой части.

Глава 1. Анатомия двоичного файла. Содержит общее введение в анатомию двоичных программ.

Глава 2. Формат ELF. Введение в двоичный формат ELF, используемый в ОС Linux.

Глава 3. Формат PE: краткое введение. Содержит краткое введение в двоичный формат PE, используемый в Windows.

Глава 4. Создание двоичного загрузчика с применением libbfd. Показано, как разбирать двоичные файлы с помощью библиотеки `libbfd`. Строится двоичный загрузчик, используемый в остальной части книги.

Часть II. Основы анализа двоичных файлов. Содержит основополагающие методы двоичного анализа.

Глава 5. Основы анализа двоичных файлов в Linux. Введение в основные инструменты двоичного анализа в Linux.

Глава 6. Основы дизассемблирования и анализа двоичных файлов. Рассматриваются базовые методы дизассемблирования и фундаментальные приемы анализа.

Глава 7. Простые методы внедрения кода для формата ELF. Первые представления о том, как модифицировать двоичный ELF-файл с помощью внедрения паразитного кода и шестнадцатеричного редактирования.

Часть III. Продвинутый анализ двоичных файлов. Целиком посвящена продвинутым методам двоичного анализа.

Глава 8. Настройка дизассемблирования. Показано, как создать собственные инструменты дизассемблирования с помощью программы Capstone.

Глава 9. Оснащение двоичных файлов. Посвящена модификации двоичных файлов с помощью полнофункциональной платформы оснащения Pin.

Глава 10. Принципы динамического анализа заражения. Введение в принципы динамического анализа заражения – современного метода двоичного анализа, позволяющего прослеживать потоки данных в программах.

Глава 11. Практический динамический анализ заражения с помощью libdft. Описывается, как построить собственные инструменты динамического анализа заражения с применением библиотеки `libdft`.

Глава 12. Принципы символического выполнения. Посвящена символическому выполнению, еще одному продвинутому методу автоматических рассуждений о сложных свойствах программы.

Глава 13. Практическое символическое выполнение с помощью Triton. Показано, как построить практически полезные инструменты символического выполнения с помощью программы Triton.

Часть IV. Приложения. Включает полезные ресурсы.

Приложение A. Краткий курс ассемблера x86. Содержит краткое введение в язык ассемблера x86 для читателей, которые с ним еще незнакомы.

Приложение B. Реализация перезаписи PT_NOTE с помощью libelf. Приведены детали реализации инструмента `elfinject`, используемого в главе 7. Может служить введением в библиотеку `libelf`.

Приложение C. Перечень инструментов анализа двоичных файлов. Содержит перечень инструментов, которые могут вам пригодиться.

Приложение D. Литература для дополнительного чтения. Содержит перечень ссылок на статьи и книги по темам, обсуждаемым в этой книге.

Как использовать эту книгу

Ниже описаны соглашения, касающиеся примеров кода, синтаксиса языка ассемблера, а также платформы разработки. Знакомство с ними поможет получить максимум пользы от чтения книги.

Архитектура системы команд

Хотя многие методы, описанные в книге, можно перенести на другие архитектуры, я во всех примерах использую архитектуру системы команд (Instruction Set Architecture – ISA) процессора Intel x86 и его 64-разрядной версии x86-64 (для краткости x64). Обе архитектуры будут обобщенно называться «x86 ISA». Как правило, в примерах приведен код для x64, если явно не оговорено противное.

Архитектура x86 ISA интересна, потому что доминирует на рынке потребительской электроники, особенно настольных компьютеров и ноутбуков, а также в исследованиях по анализу двоичных файлов (отчасти из-за ее популярности на компьютерах конечных пользователей). Поэтому многие каркасы двоичного анализа ориентированы на x86.

Кроме того, сложность x86 ISA позволит вам узнать о некоторых проблемах двоичного анализа, которые не встречаются в более простых архитектурах. У архитектуры x86 долгая история обратной совместимости (начинающаяся с 1978 года), из-за чего система команд получилась очень плотной в том смысле, что подавляющее большинство возможных байтовых значений представляет допустимый код операции. Это порождает проблему различения кода и данных, из-за которой дизассемблеры могут не понять, что интерпретируют данные как код. Мало того, длины команд различаются, и допускается невыровненный доступ к памяти для слов любой корректной длины. Таким образом, x86 допускает чрезвычайно сложные двоичные конструкции, в частности частичное перекрытие команд и невыровненные команды. Иными словами, поняв, как обращаться с такой сложной системой команд, как у процессора x86, с другими системами (например, для ARM) вы разберетесь на раз!

Синтаксис языка ассемблера

Как объяснено в приложении A, существует два популярных формата записи машинных команд x86: *синтаксис Intel* и *синтаксис AT&T*. Я буду использовать синтаксис Intel, потому что он лаконичнее. В синтаксисе Intel помещение константы в регистр `edi` записывается так:

```
mov edi,0x6
```

Заметим, что конечный операнд (`edi`) записывается первым. Если вы плохо знакомы с различиями синтаксиса AT&T и Intel, обратитесь к приложению A, где описаны основные особенности того и другого.

Формат двоичного файла и платформа разработки

Я разрабатывал все примеры кода, приведенные в этой книге, в ОС Ubuntu Linux на языках C/C++ (за исключением очень немногочисленных примеров, написанных на Python). Это связано с тем, что многие

популярные библиотеки анализа двоичных файлов ориентированы в основном на Linux и имеют удобные API, рассчитанные на C/C++ или Python. Однако все используемые в книге методы и большинство библиотек применимы также к Windows, поэтому если вы предпочитаете платформу Windows, то без труда перенесете на нее все, чему научились. Что касается форматов двоичных файлов, то в этой книге рассматривается в основном формат ELF, подразумеваемый по умолчанию на платформах Linux, хотя многие инструменты поддерживают также двоичный формат Windows PE.

Примеры кода и виртуальная машина

В каждой главе этой книги имеется несколько примеров кода, а к книге прилагается уже сконфигурированная виртуальная машина (ВМ), включающая все примеры. ВМ работает под управлением популярного дистрибутива Linux Ubuntu 16.04, на нее установлены все обсуждаемые инструменты двоичного анализа с открытым исходным кодом. Вы можете использовать эту ВМ для экспериментов с примерами кода и для решения упражнений в конце каждой главы. ВМ доступна на сайте книги по адресу *https://practicalbinaryanalysis.com* или *https://nostarch.com/binaryanalysis/*.

На сайте книги имеется также архив с исходным кодом всех примеров и упражнений. Можете скачать его, если не хотите скачивать всю ВМ, но имейте в виду, что для некоторых средств двоичного анализа необходима сложная настройка, которую вам придется выполнить самостоятельно, если вы решите не использовать ВМ.

Чтобы воспользоваться ВМ, понадобится программа виртуализации. Данная ВМ рассчитана на работу под управлением программы VirtualBox, которую можно скачать бесплатно с сайта *https://www.virtualbox.org/*. Версии VirtualBox имеются для всех популярных операционных систем, включая Windows, Linux и macOS.

После установки VirtualBox запустите ее, выберите из меню команду **File → Import Appliance** и выберите виртуальную машину, скачанную с сайта книги. После добавления запустите эту ВМ, щелкнув по зеленой стрелке **Start** в главном окне VirtualBox. Когда ВМ загрузится, войдите в систему, указав в качестве имени пользователя и пароля слово «binary». Затем откройте терминал с помощью комбинации клавиш **Ctrl**+**Alt**+**T** и можете делать все, что предлагается в книге.

В каталоге *~/code* вы найдете подкаталоги, соответствующие главам; там находятся все примеры кода и прочие файлы, относящиеся к главе. Например, весь код из главы 1 находится в каталоге *~/code/chapter1*. Есть также каталог *~/code/inc*, в котором собран общий код, используемый в программах из разных глав. Я использую расширение *.cc* для файлов с исходным кодом на C++, *.c* – для файлов с кодом на чистом C, *.h* – для заголовочных файлов и *.py* – для скриптов на Python.

Для сборки всех примеров в данной главе откройте терминал, перейдите в соответствующий этой главе каталог и выполните команду

make. Это работает во всех случаях, кроме тех, для которых явно указана другая команда сборки.

Важные примеры кода по большей части подробно обсуждаются в соответствующих главах. Если листингу обсуждаемого в книге кода соответствует исходный файл на ВМ, то перед ним указывается имя файла:

filename.c

```
int
main(int argc, char *argv[])
{
  return 0;
}
```

В заголовке этого листинга указано, что приведенный код находится в файле *filename.c*. Если явно не оговорено противное, то файл с указанным именем находится в каталоге той главы, где встретился пример. Иногда встречаются листинги, в заголовках которых указаны не имена файлов; это означает, что примеру не соответствует никакой файл в ВМ. Совсем короткие листинги без соответствующих файлов могут даже не иметь заголовков, как, например, приведенный выше код, демонстрирующий синтаксис ассемблера.

В листингах, показывающих команды оболочки и их результаты, используется символ $, обозначающий приглашение, а строки, содержащие данные, введенные пользователем, набраны полужирным шрифтом. Такие строки являются командами, которые вы можете выполнить в виртуальной машине, а следующие за ними строки, не начинающиеся приглашением и не набранные полужирным шрифтом, соответствуют выведенным командой результатам. Вот, например, распечатка содержимого каталога *~/code* на виртуальной машине:

```
$ cd ~/code && ls
chapter1 chapter2  chapter3  chapter4  chapter5  chapter6  chapter7
chapter8 chapter9  chapter10 chapter11 chapter12 chapter13 inc
```

Иногда я редактирую вывод команды в интересах удобочитаемости, поэтому результат, который вы видите в ВМ, может выглядеть немного иначе.

Упражнения

В конце каждой главы имеются упражнения и задачи для закрепления навыков. Некоторые упражнения сравнительно просты, для их решения достаточно материала, изложенного в главе. Другие требуют больше усилий и самостоятельного исследования.

ЧАСТЬ I

ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

1
АНАТОМИЯ ДВОИЧНОГО ФАЙЛА

Смысл двоичного анализа – анализ двоичных файлов. Но что такое двоичный файл? В этой главе описывается общая структура формата двоичного файла и жизненный цикл двоичного файла. Прочитав ее, вы будете готовы к восприятию двух следующих глав, посвященных двум наиболее широко распространенным форматам: ELF и PE, соответственно в ОС Linux и Windows.

В современных компьютерах вычисления производятся в двоичной системе счисления, где числа записываются строками нулей и единиц. Машинный код, выполняемый такими компьютерами, называется *двоичным кодом*. Любая программа состоит из совокупности двоичного кода (машинных команд) и данных (переменных, констант и т. п.). Чтобы различать программы, хранящиеся в данной системе, необходим способ хранения всего кода и данных, принадлежащих программе, в одном замкнутом файле. Поскольку такие файлы содержат исполняемые двоичные программы, они называются *двоичными исполняемыми файлами*, или просто *двоичными файлами* (*жарг. бинарники*). Анализ двоичных файлов и является предметом данной книги.

Прежде чем переходить к специфике таких форматов двоичных файлов, как ELF и PE, дадим краткий обзор процесса порождения исполняемых двоичных файлов из исходного кода. Затем я дизассемблирую простой двоичный файл, чтобы вы составили представление о находящихся в нем коде и данных. Этот материал пригодится нам в главах 2 и 3, когда мы будем изучать форматы ELF и PE, и в главе 4, где мы напишем собственный загрузчик, который умеет разбирать двоичные файлы и открывать их для анализа.

1.1 Процесс компиляции программы на C

Двоичные файлы порождаются в процессе *компиляции*, т. е. трансляции понятного человеку исходного кода, например на языке C или C++, в машинный код, исполняемый процессором[1]. На рис. 1.1 показаны шаги типичного процесса компиляции C-кода (шаги компиляции кода, написанного на C++, аналогичны). Компиляция C-кода состоит из четырех этапов, один из которых называется (не слишком удачно) *компиляцией*, как и весь процесс в целом. Это *препроцессирование*, *компиляция*, *ассемблирование* и *компоновка*[2]. На практике современные компиляторы часто объединяют некоторые или даже все этапы, но для демонстрации я буду рассматривать их по отдельности.

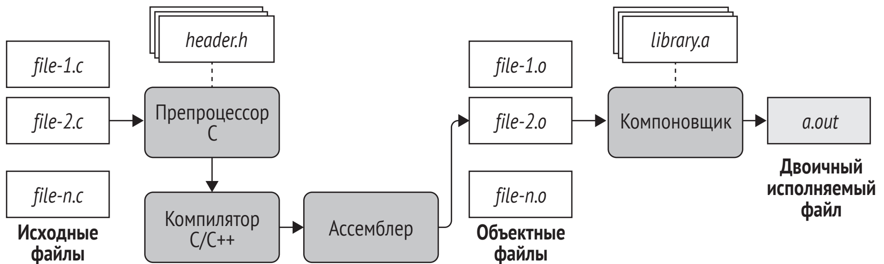

Рис. 1.1. Процесс компиляции программы на C

1.1.1 Этап препроцессирования

Процесс компиляции начинается с обработки нескольких файлов, которые вы хотите откомпилировать (на рис. 1.1 они обозначены

[1] Существуют также языки, например Python или JavaScript, программы на которых интерпретируются «на лету», а не компилируются как единое целое. Иногда части интерпретируемого кода компилируются «своевременно» (just in time – JIT), по мере выполнения программы. При этом порождается двоичный код в памяти, который можно проанализировать с применением описанных в этой книге методов. Поскольку анализ интерпретируемых языков требует зависящих от языка специальных шагов, я не стану подробно останавливаться на этом процессе.

[2] Раньше этап компоновки (linking) по-русски назывался *редактированием связей*, но сейчас этот термин вышел из употребления. – *Прим. перев.*

file-1.c, …, *file-n.c*). Исходный файл может быть всего один, но крупные программы обычно состоят из большого числа файлов. Это не только упрощает управление проектом, но и ускоряет компиляцию, потому что если изменится один какой-то файл, то перекомпилировать придется только его, а не весь код.

Исходные C-файлы могут содержать макросы (директивы `#define`) и директивы `#include`. Последние служат для включения *заголовочных файлов* (с расширением *.h*), от которых зависит исходный файл. На этапе препроцессирования все директивы `#define` и `#include` расширяются, так что остается только код на чистом C, подлежащий компиляции.

Проиллюстрируем сказанное на конкретном примере. Мы будем использовать компилятор `gcc`, подразумеваемый по умолчанию во многих дистрибутивах Linux (включая Ubuntu, операционную систему на нашей виртуальной машине). Результаты работы других компиляторов, например `clang` или Visual Studio, похожи. Как уже было сказано во введении, я компилирую все примеры (включая и текущий) в машинный код x86-64, если явно не оговорено противное.

Пусть требуется откомпилировать исходный файл на C, показанный в листинге 1.1, который выводит на экран знаменитое сообщение «Hello, world!».

Листинг 1.1. compilation_example.c

```
#include <stdio.h>

#define FORMAT_STRING "%s"
#define MESSAGE       "Hello, world!\n"

int
main(int argc, char *argv[]) {
  printf(FORMAT_STRING, MESSAGE);
  return 0;
}
```

Скоро мы увидим, что происходит с этим файлом на других этапах процесса компиляции, но пока рассмотрим только результат этапа препроцессирования. По умолчанию `gcc` автоматически выполняет все этапы компиляции, так что если мы хотим остановиться после препроцессирования и посмотреть на промежуточный результат, то об этом нужно явно сказать. В случае `gcc` это делается командой `gcc -E -P`, где флаг `-E` требует остановиться после препроцессирования, а `-P` заставляет компилятор опустить отладочную информацию, чтобы результат был немного понятнее. В листинге 1.2 показан результат этапа препроцессирования, для краткости отредактированный. Запустите ВМ и выполните предлагаемые команды.

Листинг 1.2. Результат работы препроцессора C для программы "Hello, world!"

```
$ gcc -E -P compilation_example.c

typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

/* ... */

extern int sys_nerr;
extern const char *const sys_errlist[];
extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern FILE *popen (const char *__command, const char *__modes) ;
extern int pclose (FILE *__stream);
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

int
main(int argc, char *argv[]) {
  printf(❶"%s", ❷"Hello, world!\n");
  return 0;
}
```

Заголовочный файл *stdio.h* включен целиком, т. е. все содержащиеся в нем определения типов, глобальные переменные и прототипы функций «скопированы» в исходный файл. Поскольку это делается для каждой директивы `#include`, результат работы препроцессора может оказаться очень длинным. Кроме того, препроцессор расширяет все макросы, определенные с помощью ключевого слова `#define`. В данном примере это означает, что оба аргумента `printf` (`FORMAT_STRING` ❶ и `MESSAGE` ❷) вычисляются и заменяются соответствующими константными строками.

1.1.2 Этап компиляции

После завершения этапа препроцессирования исходный файл готов к компиляции. На этапе компиляции обработанный препроцессором код транслируется на язык ассемблера. (Большинство компиляторов на этом этапе выполняют более или менее агрессивную оптимизацию, *уровень* которой задается флагами в командной строке; в случае `gcc` это флаги от `-O0` до `-O3`. В главе 6 мы увидим, что уровень оптимизации может оказывать значительное влияние на результат дизассемблирования.)

Почему на этапе компиляции порождается код на языке ассемблера, а не машинный код? Это проектное решение кажется бессмыслен-

ным в контексте одного конкретного языка (в данном случае C), но обретает смысл, если вспомнить о других языках. Из наиболее популярных компилируемых языков назовем C, C++, Objective-C, Common Lisp, Delphi, Go и Haskell. Писать компилятор, который порождает машинный код для каждого из них, было бы чрезвычайно трудоемким и долгим занятием. Проще генерировать код на языке ассемблера (тоже, кстати, достаточно трудное дело) и обрабатывать его на последнем этапе процесса одним и те же ассемблером.

Таким образом, результатом этапа компиляции является ассемблерный код, все еще понятный человеку, в котором вся символическая информация сохранена. Как уже было сказано, gcc обычно вызывает все этапы компиляции автоматически, поэтому чтобы увидеть ассемблерный код, сгенерированный на этапе компиляции, нужно попросить gcc остановиться после этого этапа и сохранить ассемблерные файлы на диске. Для этого служит флаг -S (расширение *.s* традиционно используется для файлов на языке ассемблера). Кроме того, передадим gcc флаг -masm=intel, чтобы ассемблерные команды записывались в синтаксисе Intel, а не AT&T, подразумеваемом по умолчанию. В листинге 1.3 показан результат этапа компиляции для нашего примера[1].

Листинг 1.3. Ассемблерный код, сгенерированный на этапе компиляции программы "Hello, world!"

```
$ gcc -S -masm=intel compilation_example.c
$ cat compilation_example.s

        .file   "compilation_example.c"
        .intel_syntax noprefix
        .section       .rodata
❶ .LC0:
        .string        "Hello, world!"
        .text
        .globl  main
        .type   main, @function
❷ main:
.LFB0:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        Sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     QWORD PTR [rbp-16], rsi
        mov     edi, ❸OFFSET FLAT:.LC0
```

[1] Обратите внимание, что в процессе оптимизации gcc заменил вызовы printf обращениями к puts.

```
        call    puts
        mov     eax, 0
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section .note.GNU-stack,"",@progbits
```

Пока что я не стану вдаваться в детали ассемблерного кода. Но интересно отметить, что код в листинге 1.3 читается сравнительно просто, потому что имена символов и функций сохранены. Так, константам и переменным соответствуют символические имена, а не просто адреса (пусть даже имя было сгенерировано автоматически, как в случае `LC0` ❶ для безымянной строки "Hello, world!"), а функции `main` ❷ (единственной функции в этом примере) – явная метка. Все ссылки на код и данные тоже символические, как, например, ссылка на строку "Hello, world!" ❸. Мы будем лишены такого удобства при работе с зачищенными двоичными файлами ниже в этой книге!

1.1.3 Этап ассемблирования

В конце этапа ассемблирования мы наконец получаем настоящий машинный код! На вход этого этапа поступают ассемблерные файлы, сгенерированные на этапе компиляции, а на выходе имеем набор *объектных файлов*, которые иногда называются *модулями*. Объектные файлы содержат машинные команды, которые в принципе могут быть выполнены процессором. Но, как я скоро объясню, прежде чем появится готовый к запуску исполняемый двоичный файл, необходимо проделать еще кое-какую работу. Обычно одному исходному файлу соответствует один ассемблерный файл, а одному ассемблерному файлу – один объектный. Чтобы сгенерировать объектный файл, нужно передать `gcc` флаг `-c`, как показано в листинге 1.4.

Листинг 1.4. Генерирование объектного файла с помощью gcc

```
$ gcc -c compilation_example.c
$ file compilation_example.o
compilation_example.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Чтобы убедиться, что сгенерированный файл *compilation_example.o* действительно объектный, можно воспользоваться утилитой `file` (весьма полезной, я вернусь к ней в главе 5). Как показано в листинге 1.4, это и вправду так: видно, что это файл типа `ELF 64-bit LSB relocatable`.

И что же это значит? Первая часть вывода `file` говорит, что файл отвечает спецификации формата исполняемых двоичных файлов ELF

(мы подробно рассмотрим этот формат в главе 2). Точнее, это 64-разрядный ELF-файл (поскольку в этом примере мы генерировали код для процессора x86-64), а буквы *LSB* означают, что при размещении чисел в памяти первым располагается младший байт (*L*east *S*ignificant *B*yte). Но самое главное здесь – слово *relocatable* (перемещаемый).

Перемещаемые файлы не привязаны к какому-то конкретному адресу в памяти, их можно перемещать, не нарушая никаких принятых в коде предположений. Увидев в напечатанной `file` строке слово *relocatable*, мы понимаем, что речь идет об объектном, а не исполняемом файле[1].

Объектные файлы компилируются независимо, поэтому, обрабатывая один файл, ассемблер не может знать, какие адреса упоминаются в других объектных файлах. Именно поэтому объектные файлы должны быть перемещаемыми, тогда мы сможем скомпоновать их в любом порядке и получить полный исполняемый двоичный файл. Если бы объектные файлы не были перемещаемыми, то это было бы невозможно.

Содержимое объектного файла мы увидим ниже в этой главе, когда будем готовы дизассемблировать свой первый файл.

1.1.4 Этап компоновки

Компоновка – последний этап процесса компиляции. На этом этапе все объектные файлы объединяются в один исполняемый двоичный файл. В современных системах этап компоновки иногда включает дополнительный проход, называемый *оптимизацией на этапе компоновки* (link-time optimization – *LTO*)[2].

Неудивительно, что программа, выполняющая компоновку, называется *компоновщиком*. Обычно она отделена от компилятора, который выполняет все предыдущие этапы.

Как я уже говорил, объектные файлы перемещаемы, потому что компилируются независимо друг от друга, и компилятор не может делать никаких предположений о начальном адресе объектного файла в памяти. Кроме того, объектные файлы могут содержать ссылки на функции и переменные, находящиеся в других объектных файлах или внешних библиотеках. До этапа компоновки адреса, по которым будут размещены код и данные, еще неизвестны, поэтому объектные файлы содержат только *перемещаемые символы*, которые определяют, как в конечном итоге будут разрешены ссылки на функции и переменные. В контексте компоновки ссылки, зависящие от перемещаемого символа, называются *символическими ссылками*. Если объектный файл ссылается на одну из собственных функций или переменных по абсолютному адресу, то такая ссылка тоже будет символической.

---

[1] Существуют также позиционно-независимые (перемещаемые) файлы, но о них `file` сообщает, что это разделяемые объекты, а не перемещаемые файлы. Отличить их от обыкновенных разделяемых библиотек можно по наличию адреса точки входа.

[2] Дополнительные сведения о LTO приведены в приложении D.

Задача компоновщика – взять все принадлежащие программе объектные файлы и объединить их в один исполняемый файл, который, как правило, должен загружаться с конкретного адреса в памяти. Теперь, когда известно, из каких модулей состоит исполняемый файл, компоновщик может разрешить большинство символических ссылок. Но ссылки на библиотеки могут остаться неразрешенными – это зависит от типа библиотеки.

Статические библиотеки (в Linux они обычно имеют расширение *.a*, как показано на рис. 1.1) включаются в исполняемый двоичный файл, поэтому ссылки на них можно разрешить окончательно. Но существуют также динамические (разделяемые) библиотеки, которые совместно используются всеми программами, работающими в системе. Иными словами, динамическая библиотека не копируется в каждый использующий ее двоичный файл, а загружается в память лишь один раз, и все нуждающиеся в ней двоичные файлы пользуются этой разделяемой копией. На этапе компоновки адреса, по которым будут размещаться динамические библиотеки, еще неизвестны, поэтому ссылки на них разрешить невозможно. Поэтому компоновщик оставляет символические ссылки на такие библиотеки даже в окончательном исполняемом файле, и эти ссылки разрешаются, только когда двоичный файл будет загружен в память для выполнения.

Большинство компиляторов, в т. ч. и `gcc`, автоматически вызывают компоновщик в конце процесса компиляции. Поэтому для создания полного двоичного исполняемого файла можно просто вызвать `gcc` без специальных флагов, как показано в листинге 1.5.

Листинг 1.5. Генерирование двоичного исполняемого файла с помощью gcc

```
$ gcc compilation_example.c
$ file a.out
a.out: ❶ELF 64-bit LSB executable, x86-64, version 1 (SYSV), ❷dynamically
linked, ❸interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=d0e23ea731bce9de65619cadd58b14ecd8c015c7, ❹not stripped
$ ./a.out
Hello, world!
```

По умолчанию созданный исполняемый файл называется *a.out*, но можно задать другое имя, предварив его флагом `-o`. Теперь утилита `file` сообщает, что мы имеем файл типа `ELF 64-bit LSB executable` ❶ , т. е. исполняемый, а не перемещаемый, как после этапа ассемблирования. Важно также, что файл динамически скомпонован ❷, т. е. в нем используются библиотеки, не включенные в его состав, а разделяемые с другими программами, работающими в системе. Наконец, слова `interpreter /lib64/ld-linux-x86-64.so.2` ❸ в выводе `file` говорят, какой *динамический компоновщик* будет использован для окончательного разрешения зависимостей от динамических библиотек на этапе загрузки исполняемого файла в память. Запустив двоичный

файл (командой ./a.out), вы увидите, что он делает то, что ожидалось (печатает строку "Hello, world!" на стандартный вывод), т. е. мы действительно получили работоспособный двоичный файл. Но что означают слова «not stripped» ❹ в выводе file? Обсудим это в следующем разделе.

1.2 Символы и зачищенные двоичные файлы

В исходном коде на языке высокого уровня, например C, используются функции и переменные с осмысленными именами. Компиляторы же порождают *символы*, являющиеся эквивалентом таких символических имен, и запоминают, какой двоичный код и данные соответствуют каждому символу. Например, символы функций отображают символические высокоуровневые имена функций на начальный адрес и размер функции. Эта информация обычно используется компоновщиком при объединении объектных файлов (например, чтобы разрешить ссылки на функции и переменные, находящиеся в другом модуле), а также полезна при отладке программы.

1.2.1 Просмотр информации о символах

Чтобы вы могли представить, как выглядит информация о символах, в листинге 1.6 показаны некоторые символы в двоичном файле нашей демонстрационной программы.

Листинг 1.6. Символы в двоичном файле a.out, показанные программой readelf

```
$ ❶readelf --syms a.out

Symbol table '.dynsym' contains 4 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
    0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
    1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
    2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
    3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
Symbol table '.symtab' contains 67 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
   ...
   56: 0000000000601030     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
   57: 00000000004005d0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
   58: 0000000000400550   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
   59: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   26 _end
   60: 0000000000400430    42 FUNC    GLOBAL DEFAULT   14 _start
   61: 0000000000601038     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
   62: 0000000000400526    32 FUNC    GLOBAL DEFAULT   14 ❷main
   63: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
   64: 0000000000601038     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
   65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
   66: 00000000004003c8     0 FUNC    GLOBAL DEFAULT   11 _init
```

В листинге 1.6 для отображения символов использована утилита `readelf` ❶. Мы вернемся в ней в главе 5 и расскажем, как интерпретировать ее вывод. А пока просто заметим, что среди множества незнакомых символов имеется символ для функции `main` ❷. Видно, что ему соответствует адрес (`0x400526`), с которого будет начинаться `main` после загрузки двоичного файла в память. Приведен также размер кода `main` (32 байта) и указано, что это символ функции (тип `FUNC`).

Информация о символах может быть включена в состав двоичного файла (как в примере выше) или выведена в виде отдельного файла символов. Кроме того, она может быть представлена в разных форматах. Компоновщику нужны только «голые» символы, но для отладки требуется гораздо более подробная информация. Отладочные символы содержат полное отображение между строками исходного кода и двоичными командами, они даже описывают параметры функции, кадр стека и т. д. Для двоичных файлов в формате ELF отладочные символы обычно генерируются в формате DWARF[1], тогда как для файлов в формате PE используется проприетарный формат Microsoft Portable Debugging (PDB)[2]. Данные в формате DWARF обычно встраиваются в сам двоичный файл, а в формате PDB записываются в отдельный файл символов.

Нетрудно представить, насколько полезной может быть информация о символах для анализа двоичных файлов. Приведу лишь один пример: наличие полного набора символов функций намного упрощает дизассемблирование, потому что каждый символ можно использовать как начальную точку дизассемблирования. Поэтому гораздо меньше шансов, что дизассемблер случайно интерпретирует данные как код (что привело бы к появлению бессмысленных команд на выходе). Кроме того, при наличии информации о том, какая часть двоичного кода какой функции принадлежит и какая функция вызывается, специалисту по обратной разработке будет гораздо проще разбить код на логические составляющие и понять, что он делает. Даже «голые» символы для компоновщика (лишенные дополнительной отладочной информации) оказывают огромную помощь во многих приложениях двоичного анализа.

Символы можно разобрать с помощью `readelf`, как показано выше, или программно с помощью библиотеки типа `libbfd`, как будет описано в главе 4. Имеются также библиотеки типа `libdwarf`, специально предназначенные для разбора отладочных символов в формате DWARF, но в этой книге они не рассматриваются.

К сожалению, отладочная информация обычно не включается в производственные двоичные файлы, и даже базовая информация

---

[1] Для тех, кому интересно, скажу, что акроним DWARF (*англ.* гном) никак не расшифровывается. Название было выбрано просто потому, что хорошо сочетается с «ELF» (по крайней мере, если это вызывает у вас ассоциации со сказочными существами).

[2] Для любознательных в приложении D приведены ссылки на документацию по форматам DWARF и PDB.

о символах часто удаляется, чтобы уменьшить размер файла и затруднить обратную разработку; особенно это характерно для вредоносного и коммерческого ПО. Это означает, что аналитику двоичных файлов часто приходится иметь дело с гораздо более трудным случаем зачищенных двоичных файлов, из которых вся информация о символах удалена. Поэтому в этой книге я не буду предполагать, что информация о символах присутствует, и сосредоточусь на зачищенных файлах, если явно не оговорено противное.

1.2.2 Переход на темную сторону: зачистка двоичного файла

Вы, конечно, помните, что наш демонстрационный двоичный файл еще не зачищен (что показывает утилита `file` в листинге 1.5). Видимо, по умолчанию `gcc` не зачищает откомпилированные двоичные файлы. А чтобы сделать это, нужно всего-то воспользоваться командой `strip`, как показано в листинге 1.7.

Листинг 1.7. Зачистка исполняемого файла

```
$ ❶strip --strip-all a.out
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=d0e23ea731bce9de65619cadd58b14ecd8c015c7, ❷stripped
$ readelf --syms a.out

❸ Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

Теперь двоичный файл зачищен ❶, что подтверждает выход `file` ❷. В таблице символов `.dynsym` осталось всего несколько символов ❸. Они нужны для разрешения динамических зависимостей (например, ссылок на динамические библиотеки) при загрузке двоичного файла в память, но для дизассемблирования особой ценности не представляют. Символ, соответствующий функции `main`, исчез, как и все остальные.

1.3 Дизассемблирование двоичного файла

Рассмотрев, как компилируется файл, обратимся к содержимому объектного файла, генерируемого на этапе ассемблирования. Затем я дизассемблирую сам исполняемый двоичный файл, чтобы показать, чем его содержимое отличается от содержимого объектного файла.

Это позволит нам лучше понять, что такое объектный файл и что именно добавляется на этапе компоновки.

1.3.1 Заглянем внутрь объектного файла

Пока что для дизассемблирования я воспользуюсь утилитой `objdump` (другие инструменты мы обсудим в главе 6). Это простой и легкий в использовании дизассемблер, включенный в состав большинства дистрибутивов Linux, его вполне достаточно, чтобы получить представление о коде и данных, содержащихся в двоичном файле. В листинге 1.8 приведен результат дизассемблирования объектного файла *compilation_example.o*.

Листинг 1.8. Дизассемблирование объектного файла

```
$ ❶objdump -sj .rodata compilation_example.o

compilation_example.o:     file format elf64-x86-64

Contents of section .rodata:
 0000 48656c6c 6f2c2077 6f726c64 2100      Hello, world!.

$ ❷objdump -M intel -d compilation_example.o

compilation_example.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 ❸<main>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   48 83 ec 10             sub    rsp,0x10
   8:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   b:   48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
   f:   bf 00 00 00 00          mov    edi,❹0x0
  14:   e8 00 00 00 00         ❺call   19 <main+0x19>
  19:   b8 00 00 00 00          mov    eax,0x0
  1e:   c9                      leave
  1f:   c3                      ret
```

Обратите внимание, что в листинге 1.8 утилита `objdump` вызывается дважды. При первом вызове ❶ я попросил показать содержимое секции `.rodata`. Ее название означает «read-only data» (данные, предназначенные только для чтения); именно в этой части двоичного файла хранятся все константы, включая строку «Hello, world!». Я вернусь к более подробному обсуждению `.rodata` и других секций ELF-файла в главе 2, где рассматривается этот двоичный формат. А пока заметим, что в секции `.rodata` находится строка в кодировке ASCII, показанная в левой части вывода. В правой же части находится понятное человеку представление тех же самых байтов.

При втором вызове `objdump` ❷ дизассемблируется весь находящийся в объектном файле код, и результат представляется в синтаксисе

Intel. Мы видим только код функции `main` ❸, потому что это единственная функция в исходном файле. По большей части, вывод очень близок к ассемблерному коду, сгенерированному на этапе компиляции (плюс-минус несколько ассемблерных макросов). Интересно отметить, что указатель на строку «Hello, world!» (❹) инициализируется нулем. Следующий вызов ❺, который должен бы вывести строку на экран с помощью функции `puts`, также содержит бессмысленный адрес (смещение 19, где-то в середине `main`).

Почему вызов, который должен ссылаться на `puts`, вместо этого указывает в середину `main`? Ранее я говорил, что ссылки на код и данные в объектных файлах еще не полностью разрешены, потому что компилятор не знает, по какому базовому адресу будет в конечном итоге загружена программа. Потому-то вызов `puts` и не разрешен. Объектный файл ждет, когда компоновщик подставит правильное значение вместо этой ссылки. Вы можете убедиться в этом, попросив `readelf` показать все перемещаемые символы в объектном файле, как показано в листинге 1.9.

Листинг 1.9. Перемещаемые символы, показанные readelf

```
$ readelf --relocs compilation_example.o

Relocation section '.rela.text' at offset 0x210 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
❶ 000000000010  00050000000a R_X86_64_32    0000000000000000 .rodata + 0
❷ 000000000015  000a00000002 R_X86_64_PC32  0000000000000000 puts - 4
...
```

Перемещаемый символ в строке ❶ говорит компоновщику, что нужно разрешить ссылку на строку, так чтобы она указывала на ее окончательный адрес в секции `.rodata`.

Заметьте, что из символа `puts` вычитается значение 4. Пока можете не обращать на это внимания; способ вычисления смещений компоновщиком довольно сложный, и вывод `readelf` может вызвать замешательство, поэтому я пока опущу детали перемещения и представлю общую картину дизассемблирования двоичного файла. А о перемещаемых символах мы поговорим в главе 2.

В левом столбце каждой строки вывода `readelf` (на сером фоне) показано смещение того места в объектном файле, в которое нужно подставить разрешенную ссылку. Приглядевшись, вы увидите, что в обоих случаях оно равно смещению подлежащей исправлению команды плюс 1. Например, обращение к `puts` в выводе `objdump` смещено от начала кода на величину `0x14`, однако перемещаемый символ указывает на смещение `0x15`. Объясняется это тем, что нам нужно перезаписать только *операнд* команды, но не *код операции*. Так уж случилось, что в обеих нуждающихся в исправлении командах код операции занимает 1 байт, поэтому для указания на операнд перемещаемый символ должен пропустить этот байт.

1.3.2 Изучение полного исполняемого двоичного файла

Познакомившись с содержимым объектного файла, перейдем к дизассемблированию полного двоичного файла. Начнем с файла, содержащего символы, а затем займемся его зачищенной версией, чтобы посмотреть, чем будут отличаться результаты дизассемблирования. Между дизассемблированными объектным и исполняемым файлами есть большая разница, в чем легко убедиться, взглянув на выход `objdump` в листинге 1.10.

Листинг 1.10. Дизассемблирование исполняемого файла с помощью objdump

```
$ objdump -M intel -d a.out

a.out:     file format elf64-x86-64

Disassembly of section ❶.init:

00000000004003c8 <_init>:
  4003c8:  48 83 ec 08           sub   rsp,0x8
  4003cc:  48 8b 05 25 0c 20 00  mov   rax,QWORD PTR [rip+0x200c25]
  4003d3:  48 85 c0              test  rax,rax
  4003d6:  74 05                 je    4003dd <_init+0x15>
  4003d8:  e8 43 00 00 00        call  400420 <__libc_start_main@plt+0x10>
  4003dd:  48 83 c4 08           add   rsp,0x8
  4003e1:  c3 ret

Disassembly of section ❷.plt:

00000000004003f0 <puts@plt-0x10>:
  4003f0: ff 35 12 0c 20 00      push  QWORD PTR [rip+0x200c12]
  4003f6: ff 25 14 0c 20 00      jmp   QWORD PTR [rip+0x200c14]
  4003fc: 0f 1f 40 00            nop   DWORD PTR [rax+0x0]

0000000000400400 <puts@plt>:
  400400: ff 25 12 0c 20 00      jmp   QWORD PTR [rip+0x200c12]
  400406: 68 00 00 00 00         push  0x0
  40040b: e9 e0 ff ff ff         jmp   4003f0 <_init+0x28>

...

Disassembly of section ❸.text:
  0000000000400430 <_start>:
  400430: 31 ed                  xor   ebp,ebp
  400432: 49 89 d1               mov   r9,rdx
  400435: 5e                     pop   rsi
  400436: 48 89 e2               mov   rdx,rsp
  400439: 48 83 e4 f0            and   rsp,0xfffffffffffffff0
  40043d: 50                     push  rax
  40043e: 54                     push  rsp
  40043f: 49 c7 c0 c0 05 40 00   mov   r8,0x4005c0
  400446: 48 c7 c1 50 05 40 00   mov   rcx,0x400550
```

```
  40044d: 48 c7 c7 26 05 40 00   mov   rdi,0x400526
  400454: e8 b7 ff ff ff         call  400410 <__libc_start_main@plt>
  400459: f4                     hlt
  40045a: 66 0f 1f 44 00 00      nop   WORD PTR [rax+rax*1+0x0]

0000000000400460 <deregister_tm_clones>:
...

0000000000400526 ❹<main>:
  400526: 55                     push  rbp
  400527: 48 89 e5               mov   rbp,rsp
  40052a: 48 83 ec 10            sub   rsp,0x10
  40052e: 89 7d fc               mov   DWORD PTR [rbp-0x4],edi
  400531: 48 89 75 f0            mov   QWORD PTR [rbp-0x10],rsi
  400535: bf d4 05 40 00         mov   edi,0x4005d4
  40053a: e8 c1 fe ff ff         call  400400 ❺<puts@plt>
  40053f: b8 00 00 00 00         mov   eax,0x0
  400544: c9                     leave
  400545: c3                     ret
  400546: 66 2e 0f 1f 84 00 00   nop   WORD PTR cs:[rax+rax*1+0x0]
  40054d: 00 00 00

0000000000400550 <__libc_csu_init>:
...

Disassembly of section .fini:
  00000000004005c4 <_fini>:
  4005c4: 48 83 ec 08            sub   rsp,0x8
  4005c8: 48 83 c4 08            add   rsp,0x8
  4005cc: c3                     ret
```

Как видите, в двоичном файле кода гораздо больше, чем в объектном. Теперь это не только функция `main`, да и секция отнюдь не единственная. Существуют секции с именами `.init` ❶, `.plt` ❷, `.text` ❸ и др. Все они содержат код, служащий разным целям, например предназначенный для инициализации программы или являющийся заглушкой для вызова функций из разделяемых библиотек.

Секция `.text` – это основная секция кода, она содержит функцию `main` ❹, а также ряд других функций, например `_start`, отвечающих, в частности, за подготовку аргументов командной строки, настройку среды выполнения для `main` и очистку после завершения `main`. Это стандартные функции, присутствующие в любом двоичном ELF-файле, сгенерированном `gcc`.

Видно также, что отсутствовавшие ранее ссылки на код и данные теперь разрешены компоновщиком. Например, обращение к `puts` ❺ сейчас указывает на нужную заглушку (в секции `.plt`) для доступа к разделяемой библиотеке, содержащей `puts`. (Как работают PLT-заглушки, я объясню в главе 2.)

Итак, полный исполняемый двоичный файл содержит значительно больше кода (и данных, хотя я их не показал), чем соответствующий объектный файл. Но до сих пор интерпретировать вывод было

ненамного труднее. Все меняется, если двоичный файл зачищен. Это видно в листинге 1.11, где показан результат дизассемблирования зачищенной версии демонстрационного файла утилитой `objdump`.

Листинг 1.11. Дизассемблирование зачищенного исполняемого файла с помощью objdump

```
$ objdump -M intel -d ./a.out.stripped

./a.out.stripped:     file format elf64-x86-64

Disassembly of section ❶.init:

00000000004003c8 <.init>:
  4003c8: 48 83 ec 08           sub   rsp,0x8
  4003cc: 48 8b 05 25 0c 20 00  mov   rax,QWORD PTR [rip+0x200c25]
  4003d3: 48 85 c0              test  rax,rax
  4003d6: 74 05                 je    4003dd <puts@plt-0x23>
  4003d8: e8 43 00 00 00        call  400420 <__libc_start_main@plt+0x10>
  4003dd: 48 83 c4 08           add   rsp,0x8
  4003e1: c3                    ret

Disassembly of section ❷.plt:
...

Disassembly of section ❸.text:

0000000000400430 <.text>:
❹ 400430: 31 ed                 xor   ebp,ebp
  400432: 49 89 d1              mov   r9,rdx
  400435: 5e                    pop   rsi
  400436: 48 89 e2              mov   rdx,rsp
  400439: 48 83 e4 f0           and   rsp,0xfffffffffffffff0
  40043d: 50                    push  rax
  40043e: 54                    push  rsp
  40043f: 49 c7 c0 c0 05 40 00  mov   r8,0x4005c0
  400446: 48 c7 c1 50 05 40 00  mov   rcx,0x400550
  40044d: 48 c7 c7 26 05 40 00  mov   rdi,0x400526
❺ 400454: e8 b7 ff ff ff        call  400410 <__libc_start_main@plt>
  400459: f4                    hlt
  40045a: 66 0f 1f 44 00 00     nop   WORD PTR [rax+rax*1+0x0]
❻ 400460: b8 3f 10 60 00        mov   eax,0x60103f
  ...
  400520: 5d pop rbp
  400521: e9 7a ff ff ff        jmp   4004a0 <__libc_start_main@plt+0x90>
❼ 400526: 55                    push  rbp
❽ 400527: 48 89 e5              mov   rbp,rsp
  40052a: 48 83 ec 10           sub   rsp,0x10
  40052e: 89 7d fc              mov   DWORD PTR [rbp-0x4],edi
  400531: 48 89 75 f0           mov   QWORD PTR [rbp-0x10],rsi
  400535: bf d4 05 40 00        mov   edi,0x4005d4
  40053a: e8 c1 fe ff ff        call  400400 <puts@plt>
  40053f: b8 00 00 00 00        mov   eax,0x0
```

```
❾   400544: c9                      leave
    400545: c3                      ret
    400546: 66 2e 0f 1f 84 00 00    nop   WORD PTR cs:[rax+rax*1+0x0]
    40054d: 00 00 00
    400550: 41 57                   push  r15
    400552: 41 56                   push  r14
    ...

  Disassembly of section .fini:

  00000000004005c4 <.fini>:
    4005c4: 48 83 ec 08             sub   rsp,0x8
    4005c8: 48 83 c4 08             add   rsp,0x8
    4005cc: c3                      ret
```

Какой урок мы можем вынести из листинга 11.1? Различные секции по-прежнему хорошо различимы (они помечены цифрами ❶, ❷ и ❸), но функции – уже нет. Все функции слились в один большой блок кода. Функция `_start` начинается в точке ❹, а функция `deregister_tm_clones` – в точке ❻. Функция `main` начинается в точке ❼ и заканчивается в точке ❽, но ни в одном из этих случаев нет ничего, что позволило бы сказать, что команды как-то связаны с началом функции. Единственные исключения – функции в секции `.plt`, которые по-прежнему имеют имена (что видно на примере вызова функции `__libc_start_main` в точке ❺). А в остальном мы должны сами попытаться извлечь смысл из результата дизассемблирования.

Даже в этом простом примере все запутано, а представьте, что было бы в случае большого двоичного файла, содержащего сотни функций, слипшихся в один ком! Именно поэтому во многих областях анализа двоичных файлов так важно иметь точный механизм автоматизированного обнаружения функций, который мы подробно обсудим в главе 6.

1.4 Загрузка и выполнение двоичного файла

Теперь вы знаете, как работает компилятор и как устроены внутри двоичные файлы. Вы также научились дизассемблировать двоичные файлы с помощью утилиты `objdump`. Если вы прорабатывали примеры, то на вашем диске даже есть новенький, с пылу с жару двоичный файл. Теперь посмотрим, что происходит во время загрузки и выполнения двоичного файла. Это будет полезно при обсуждении идей динамического анализа в последующих главах.

Детали зависят от платформы и формата двоичного файла, но процесс загрузки и выполнения двоичного файла, как правило, состоит из нескольких шагов. На рис. 1.2 показано, как загруженный двоичный ELF-файл (например, откомпилированный выше) расположен в памяти Linux-системы. На верхнем уровне загрузка двоичного PE-файла в Windows очень похожа.

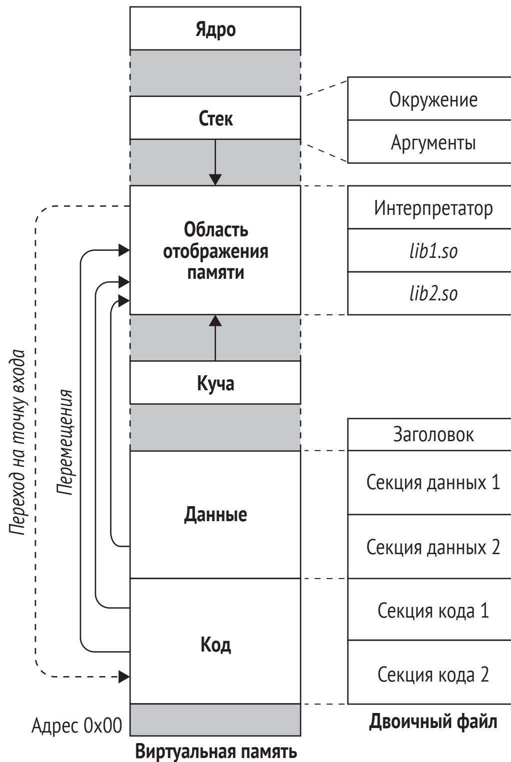

Рис. 1.2. Загрузка ELF-файла в Linux

Загрузка двоичного файла – сложный процесс, требующий большой работы от операционной системы. Важно также понимать, что представление двоичного файла в памяти необязательно один в один соответствует его представлению на диске. Например, большие участки данных, инициализированных нулями, могут быть свернуты на диске (для экономии места), но в памяти все эти нули будут присутствовать. Некоторые части двоичного файла на диске могут быть упорядочены в памяти по-другому или вообще отсутствовать. Поскольку детали зависят от формата файла, я отложу вопрос о представлениях двоичного файла в памяти и на диске до главы 2 (формат ELF) и главы 3 (формат PE). А пока рассмотрим в общих чертах, что происходит в процессе загрузки.

Когда вы запускаете двоичный файл, операционная система первым делом подготавливает новый процесс, в котором программа будет исполняться, и, в частности, виртуальное адресное пространство[1]. Затем операционная система отображает *интерпретатор*

[1] В современных операционных системах, где одновременно работает много программ, у каждой программы имеется свое виртуальное адресное пространство, изолированное от виртуальных адресных пространств других программ. Во всех обращениях к памяти со стороны приложений, работающих в режиме пользователя, используются виртуальные, а не фи-

в виртуальную память процесса. Эта программа работает в режиме пользователя и знает, как загружать двоичный файл и выполнять необходимые перемещения. В Linux в роли интерпретатора обычно выступает разделяемая библиотека *ld-linux.so*. В Windows функциональность интерпретатора реализована в библиотеке *ntdll.dll*. После загрузки интерпретатора ядро передает ему управление, и тот начинает работать.

В двоичных ELF-файлах в Linux имеется специальная секция `.interp`, где указан путь к интерпретатору, который будет загружать данный файл. Это видно из результата `readelf`, показанного в листинге 1.12.

Листинг 1.12. Содержимое секции .interp

```
$ readelf -p .interp a.out

String dump of section '.interp':
  [     0] /lib64/ld-linux-x86-64.so.2
```

Как уже было сказано, интерпретатор загружает двоичный файл в его виртуальное адресное пространство (то самое, в которое загружен он сам). Затем он разбирает двоичный файл и определяет (среди прочего), какие динамические библиотеки тот использует. Эти библиотеки интерпретатор отображает в виртуальное адресное пространство (с помощью функции `mmap` или эквивалентной ей), после чего выполняет оставшиеся перемещения в секциях кода, чтобы подставить правильные адреса вместо ссылок на динамические библиотеки. В действительности процесс разрешения ссылок на функции в динамических библиотеках часто откладывается на потом. Иначе говоря, вместо разрешения этих ссылок сразу в момент загрузки интерпретатор откладывает это на момент первого вызова. Это называется *поздним связыванием* и будет объяснено подробнее в главе 2. Завершив перемещение, интерпретатор находит точку входа в двоичный файл и передает ей управление, после чего начинается собственно выполнение двоичного файла.

1.5 Резюме

Познакомившись с общей анатомией и жизненным циклом двоичного файла, мы можем перейти к деталям конкретных двоичных форматов. Начнем с широко распространенного формата ELF, являющегося предметом следующей главы.

зические адреса. Операционная система может загружать части виртуальной памяти в физическую или выгружать оттуда по мере необходимости, благодаря чему многие программы прозрачно разделяют сравнительно небольшую физическую память.

Упражнения

1. Нахождение функций

Напишите на C программу, содержащую несколько функций, и откомпилируйте ее, получив ассемблерный файл, объектный файл и исполняемый двоичный файл. Попытайтесь найти написанные вами функции во всех трех файлах. Видите ли вы соответствие между кодом на C и на ассемблере? Зачистите исполняемый файл и снова попробуйте идентифицировать функции.

2. Секции

Как вы видели, двоичные ELF-файлы (и файлы в других форматах) разбиты на секции. Одни секции содержат код, другие – данные. Зачем, на ваш взгляд, нужно разделение между секциями кода и данных? Как вы думаете, чем различаются процессы загрузки кода и данных? Необходимо ли копировать все секции в память, когда двоичный файл загружается для выполнения?