Эльфы и пингвины: что такое ELF и как он работает в Linux?

Алексеев Бронислав

Всем привет! С вами как всегда я, Аргентум. Сегодня я расскажу и поведаю вам древние тайны, которые хранят горные старцы-сисадмины — тайны об эльфах, и как они взаимодействуют с древним народцем пингвинов. Дамы и господа, встречайте — статья о работе ELF и двоичных файлов в Linux!

Что такое ELF? Чем он отличается от PE в Windows? И многие другие ответы на ваши вопросы.

Перед тем как погрузиться в технические детали, будет нелишним объяснить, почему понимание формата ELF полезно. Это позволяет изучить внутреннюю работу операционной системы. Когда что-то пошло не так, эти знания помогут лучше понять, что именно случилось, и по какой причине. Также возможность изучения ELF-файлов может быть ценна для поиска дыр в безопасности и обнаружения подозрительных файлов. И наконец, для лучшего понимания процесса разработки. Даже если вы программируете на высокоуровневом языке типа Go или Rust, вы всё равно будет лучше знать, что происходит за сценой.

Итак, зачем изучать ELF?

- Для общего понимания работы операционной системы
- Для разработки ПО
- Цифровая криминалистика и реагирование на инциденты (DFIR)
- Исследование вредоносных программ (анализ бинарных файлов)

Статья была вдохновлена книгой Дэнниса Эндрисса "практический анализ двоичных файлов", и основана на этой книге.

Формат исполняемых и связываемых файлов (ELF) — это стандартный формат двоичных файлов, используемый для хранения исполняемых файлов, объектного кода, общих библиотек и дампов ядра в Linux и других Unix-подобных системах. Разработанный в начале 1990-х годов, ELF представляет собой гибкий и расширяемый формат, который работает в различных архитектурах и операционных системах. Распространённым заблуждением является то, что файлы ELF предназначены только для бинарных или исполняемых файлов. Мы уже сказали, что они могут быть использованы для частей исполняемых файлов (объектного кода). Другим примером являются файлы библиотек и дампы ядра (core-файлы и a.out файлы). Спецификация ELF также используется в Linux для ядра и модулей ядра.

На всякий случай, я создал репозиторий, который содержит исходный код, исполняемые файлы на разных этапах компиляции, и копию этой статьи в директории `docs/`.

Вступление, или Введение в бинарные файлы

Подавляющее большинство компьютерных программ написаны на языках высокого уровня — Java, C, C++ и другие. Такие программы необходимо скомпилировать, в результате чего создаются *двоичные исполняемые файлы*, содержащие машинный код из нулей и единиц. Любой кусочек информации в компьютере записан в двоичном формате, поэтому для того чтобы сохранить что-то полезное, нужно информацию преобразовать в этот самый вид.

Двоичный файл -это компьютерный файл, который не является текстовым файлом. Термин "двоичный файл" часто используется как термин, означающий "нетекстовый файл". Многие форматы двоичных файлов содержат части, которые могут быть интерпретированы как текст; например, некоторые компьютерные файлы документов, содержащие форматированный текст, такие как старые файлы документов Microsoft Word, содержат текст документа, но также содержат информацию о форматировании в двоичной форме.

```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 0300 0100 0000 8083 0408 3400 0000  ............4...
00000020: 9c11 0000 0000 0000 3400 2000 0900 2800  ........4. ...(.
00000030: 1e00 1b00 0600 0000 3400 0000 3480 0408  ........4...4...
00000040: 3480 0408 2001 0000 2001 0000 0500 0000  4... ... .......
00000050: 0400 0000 0300 0000 5401 0000 5481 0408  ........T...T...
00000060: 5481 0408 1300 0000 1300 0000 0400 0000  T...............
00000070: 0100 0000 0100 0000 0000 0000 0080 0408  ................
00000080: 0080 0408 1407 0000 1407 0000 0500 0000  ................
00000090: 0010 0000 0100 0000 080f 0000 089f 0408  ................
000000a0: 089f 0408 2001 0000 2401 0000 0600 0000  .... ...$.......
000000b0: 0010 0000 0200 0000 140f 0000 149f 0408  ................
000000c0: 149f 0408 e800 0000 e800 0000 0600 0000  ................
000000d0: 0400 0000 0400 0000 6801 0000 6881 0408  ........h...h...
000000e0: 6881 0408 4400 0000 4400 0000 0400 0000  h...D...D.......
000000f0: 0400 0000 50e5 7464 1806 0000 1886 0408  ....P.td........
00000100: 1886 0408 3400 0000 3400 0000 0400 0000  ....4...4.......
00000110: 0400 0000 51e5 7464 0000 0000 0000 0000  ....Q.td........
00000120: 0000 0000 0000 0000 0000 0000 0600 0000  ................
00000130: 0400 0000 52e5 7464 080f 0000 089f 0408  ....R.td........
00000140: 089f 0408 f800 0000 f800 0000 0400 0000  ................
00000150: 0100 0000 2f6c 6962 2f6c 642d 6c69 6e75  ..../lib/ld-linu
00000160: 782e 736f 2e32 0000 0400 0000 1000 0000  x.so.2..........
00000170: 0100 0000 474e 5500 0000 0000 0200 0000  ....GNU.........
00000180: 0600 0000 1800 0000 0400 0000 1400 0000  ................
00000190: 0300 0000 474e 5500 a5f4 4b82 9c47 27ed  ....GNU...K..G'.
000001a0: 369f 823f 19d5 7508 7673 f34e 0200 0000  6..?..u.vs.N....
000001b0: 0600 0000 0100 0000 0500 0000 0020 0020  ............. . 
000001c0: 0000 0000 0600 0000 ad4b e3c0 0000 0000  .........K......
000001d0: 0000 0000 0000 0000 0000 0000 3500 0000  ............5...
000001e0: 0000 0000 0000 0000 1200 0000 2900 0000  ............)...
000001f0: 0000 0000 0000 0000 1200 0000 0100 0000  ................
00000200: 0000 0000 0000 0000 2000 0000 3c00 0000  ........ ...<...
00000210: 0000 0000 0000 0000 1200 0000 2e00 0000  ................
00000220: 0000 0000 0000 0000 1200 0000 1a00 0000  ................
00000230: f485 0408 0400 0000 1100 0f00 005f 5f67  .............__g
00000240: 6d6f 6e5f 7374 6172 745f 5f00 6c69 6263  mon_start__.libc
00000250: 2e73 6f2e 3600 5f49 4f5f 7374 6469 6e5f  .so.6._IO_stdin_
00000260: 7573 6564 0070 7574 7300 6d65 6d73 6574  used.puts.memset
00000270: 006d 616c 6c6f 6300 5f5f 6c69 6263 5f73  .malloc.__libc_s
00000280: 7461 7274 5f6d 6169 6e00 474c 4942 435f  tart_main.GLIBC_
00000290: 322e 3000 0000 0200 0200 0000 0200 0200  2.0.............
```

Какую бы операционную систему мы не использовали, необходимо каким-то образом транслировать функции исходного кода на язык CPU — машинный код. Функции могут быть самыми базовыми, например, открыть файл на диске или вывести что-то на экран. Вместо того, чтобы напрямую использовать язык CPU, мы используем язык программирования, имеющий стандартные функции. Компилятор затем транслирует эти функции в объектный код. Этот объектный код затем линкуется в полную программу, путём использования линкера. Результатом является двоичный файл, который может быть выполнен на конкретной платформе и конкретном типе CPU.

Архитектура системы команд

Для всех примеров, я буду уверен, что вы используйте архитектуру (ISA) процессора Intel x86 и его 64-разрядной версии x86-64 (x64). Обе архитектуры обобщенно называются x86 ISA. Эта архитектура доминирует на рынке электроники и в анализе двоичных файлов. У архитектуры x86 длинная история — прямиком с 1978 года, и из за обратной совместимости система команд получилась очень плотная, а также много легаси-остатков. Это порождает проблему различения кода и данных. В общем, система команда x86 очень сложная, но изучив её, вы сможете легко изучить другую архитектуру — например ARM или другую, менее популярную архитектуру.

Анатомия бинарных файлов

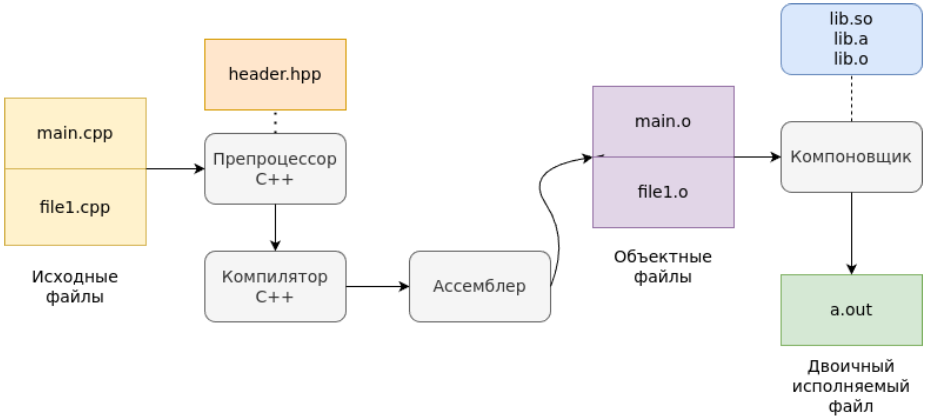
В этом разделе мы рассмотрим общую структуру бинарного файла. В современных компьютерах вычисления производятся в двоичной системе счисления, где числа записываются строками нулей и единиц. Машинный код, выполняемый такими компьютерами, называется бинарный или двоичный код. Любая программа состоит из совокупности двоичного кода (машинных команд) и данных (переменных, констант, и прочие). Чтобы различать программы, хранящиеся в данной системе, необходим способ хранения всего кода и данных, принадлежащих программе, в одном исполняемом и замкнутом файле. Поскольку такие файлы содержат исполняемые двоичные программы, они называются двоичными исполняемыми файлами или просто двоичными файлами — проще говоря, бинарники.

Чтобы перейти к специфике формата двоичных файлов ELF, надо кратко узнать процесс создания исполняемых файлов из исходного кода.

Процесс компиляции программы на C/C++

Для того, чтобы узнать, как устроен бинарный формат, давайте скомпилируем программу на C. Это поможет нам узнать, какие этапы проходит наша программа перед тем, как оказаться запущенной.

Двоичные файлы создаются в процессе компиляции, то есть трансляции понятного человеку исходного кода, например на языке программирования C/C++, в машинный код, исполняемый процессором.



Компиляция C/C++, или другого кода состоит из четырех этапов — препроцессирование, компиляция, ассемблирование и компоновка. На практике современные компиляторы часто объединяют некоторые или даже все этапы, но для демонстрации можно будет использовать их по отдельности.

Этап препроцессирования

Давайте изучим этап препроцессирования. Процесс компиляции начинается с обработки нескольких файлов, которые вы хотите откомпилировать.

Хэдеры, включенные в программу с помощью директивы `#include`, рекурсивно проходят стадию препроцессинга и включаются в выпускаемый файл. Однако, каждый хэдер может быть открыт во время препроцессинга несколько раз, поэтому, обычно, используются специальные препроцессорные директивы, предохраняющие от циклической зависимости.

Препроцессор — это макро процессор, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит происходит работа с препроцессорными директивами. Например, препроцессор добавляет хэдеры в код (#include), убирает комментирования, заменяет макросы (#define) их значениями, выбирает нужные куски кода в соответствии с условиями #if, #ifdef и #ifndef.

Исходный файл может быть всего один, но крупные программы обычно состоят из большого количества файлов (это является хорошей практикой, чтобы главный файл не был засорен ненужным кодом).
Исходные C и C++ файлы могут содержать макросы (директивы `#define`) и директивы `#include`. Последний служат для включение библиотек и заголовочных файлов (с расширением .h), от которых зависит исходный файл. На этапе препроцессирования все директивы `#define` и `#include` расширяются, так что остается код на чистом C, подлежащий компиляции.

Давайте узнаем побольше об этом, написав код. Я использую gcc версии 12.2.0. Для начала создадим код, который выведет на экран строку "Hello, Habr!"

```
 #include <stdio.h> // подключаем хедер стандартной сишной библиотеки ввода-вывода

// Создаем макросы GREETING и MESSAGE
#define GREETING "Hello, %s" // макрос для printf-функции, который принимает %s - строку
#define MESSAGE "Habr" // макрос сообщения, в нашем случае - Habr

// Главная функция
int main() {
    printf(GREETING, MESSAGE); // выводим макросы

    return 0; // выходим из программы посредством возвращения нуля
}
```

Скоро мы увидим, что происходит на других этапах процесса компиляции, но пока рассмотрим только результат этапа препроцессирования. По умолчанию GCC выполняет все этапы компиляции разом, но существуют флаги для остановки компиляции на разных этапах. Для остановки компиляции на режиме препроцессирования нам нужно ввести команду `gcc -E -P <example.c> -o <example_processed.ii>`.

Разберем данную команду. Флаг -E требует остановиться после препроцессирования, а -P заставляет опустить отладочную информацию. Флаг -o означает, куда должен записаться результат. Ниже я приведу пример файла на этапе препроцессирования, для краткости измененный.

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

/* ... */

extern int __uflow (FILE *);
extern int __overflow (FILE *, int);

// Ага! Вот и наш код. Но немного измененный - нету макросов
int main() {
    printf("Hello, %s", "Habr");
    return 0;
}
```

Заголовочный файл `stdio.h` включен целиком, вместе со всеми определениями типов, глобальными переменными, прототипов функций — все это скопировано в главный файл. Поскольку это делается для каждой директивы `#include`, если подключить несколько библиотек или заголовочных файлов, результат будет очень длинный. Кроме того, препроцессор расширяет все макросы, определенные с помощью ключевого слова `#define`. В данном примере это означает, что оба аргумента printf (GREETING и MESSAGE) вычисляются и заменяются соответствующими строками.

Этап компиляции

Итак, после того как мы получили код на C, нам нужно его скомпилировать. Но не сразу в двоичный исполняемый файл — а в ассемблерный код. После завершения препроцессирования исходный файл готов к компиляции. На этапе компиляции обработанный препроцессором код транслируется на *язык ассемблера* (Ассемблерный код — это доступное для понимания человеком представление машинного кода). Большинство компиляторов на этом этапе выполняют более или менее агрессивную оптимизацию, уровень которой задается флагами, в случае с gcc это флаги от -O0 до -O3.

Почему на этапе компиляции порождается код на ассемблере, а не машинный код? Это решение кажется бессмысленным в контексте одного конкретного языка (например, C), но обретает смысл, если вспомнить о других языках программирования. Из наиболее популярных компилируемых языков назовем C, C++, Common Lisp, 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 и обрабатывать его на последнем этапе процесса одним и тем же ассемблером.

Таким образом, результатом этапа компиляции становится ассемблерный код, все еще понятый 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, хотя довольно часто используют и просто .asm). Кроме того, передадим gcc флаг `-masm=intel`, который заставляет использовать язык ассемблера в синтаксисе Intel, а не AT&T. Синтаксис AT&T менее популярный и менее читаемый, по сравнению с Intel-овским синтаксисом. Итак, и вот команда для компиляции: `gcc -S -masm=intel <example.c> -o <example_asm>.s`

Ниже я вставил краткую выдержку из ассемблерного файла нашего кода:

```
        .file   "hello.c"
        .intel_syntax noprefix
        .text
.Ltext0:
        ;;; /home/argentum/Coding/ELF-x86-research это моя директория для изучения ELF файлов, а
        ;;; src/hello.c - директория с исходным кодом нашей программы
        .file 0 "/home/argentum/Coding/ELF-x86-research" "src/hello.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Habr"
.LC1:
        .string "Hello, %s"
        .section        .text.startup,"ax",@progbits
        .p2align 4,,10
        .p2align 3
        .globl  main
        .type   main, @function
```

Кстати, в процессе оптимизации кода ваш исходный код может немного измениться. Компилятор знает, что сложение стоит дешевле умножения, и если требуется умножить 2 на 2, то он вместо умножения просто сложит 2 и 2.

Этап ассемблирования

Так как x86 процессоры исполняют команды на бинарном коде, необходимо перевести ассемблерный код в машинный с помощью ассемблера. Ассемблер преобразовывает ассемблерный код в машинный код, сохраняя его в объектном файле.

В конце этапа ассемблирования мы наконец-то получим настоящий, с пылу с жару, машинный код! На вход этого этапа поступают ассемблерные файлы, сгенерированные на этапе компиляции, а на выходе имеем набор *объектных файлов*, которые иногда называют *модулями*. Объектные файлы содержат машинные команды, которые в принципе могут быть выполнены процессором. Но, как мы скоро узнаем, 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.

Объектный файл — это созданный ассемблером промежуточный файл, хранящий кусок машинного кода. Этот кусок машинного кода, который еще не был связан вместе с другими кусками машинного кода в конечную выполняемую программу, называется объектным кодом.

Далее возможно сохранение данного объектного кода в статические библиотеки для того, чтобы не компилировать данный код снова.

Чтобы убедиться, что сгенерированный объектный файл действительно объектный, можно воспользоваться утилитой file:

```
file <example_obj.o>
```

```
<example_obj.o>: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), with debug_info, not stripped
```

Первая часть нам говорит, что файл отвечает спецификации формата исполняемых двоичных файлов ELF. Точнее, это 64-разрядный ELF-файл, а буквы LSB означают, что при размещении чисел в памяти первым располагается младший байт (Least Significant Byte). 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 выводе file слово relocatable, мы понимаем, что речь идет об объектном, а не исполняемом файле (но существуют также позиционно-независимые (пермещаемые) 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). Объектные файлы компилируются независимо, поэтому, обрабатывая один файл, ассемблер может не 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. Но на данном шаге еще ничего не закончено, ведь объектных файлов может быть много и нужно их все соединить в единый исполняемый файл с помощью компоновщика (линкера). Поэтому мы переходим к следующей стадии.

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)*.

Неудивительно, что программа, выполняющая компоновку, называется компоновщиком (линкером). Обычно линкер отделен от компилятора, который выполняет все предыдущие этапы. Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о таблице символов.

Таблица символов — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах.
Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

Как мы уже выяснили, объектные файлы перемещаемые, 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 перемещающего символа, называются *символическими ссылками*. Если объектный файл ссылается на одну из собственных функций или переменных по абсолютному адресу, то такая ссылка тоже будет символической.

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.

Статистические библиотеки (в Linux они обычно имеют расширение .a) 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 загружаются в память лишь однажды, 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.

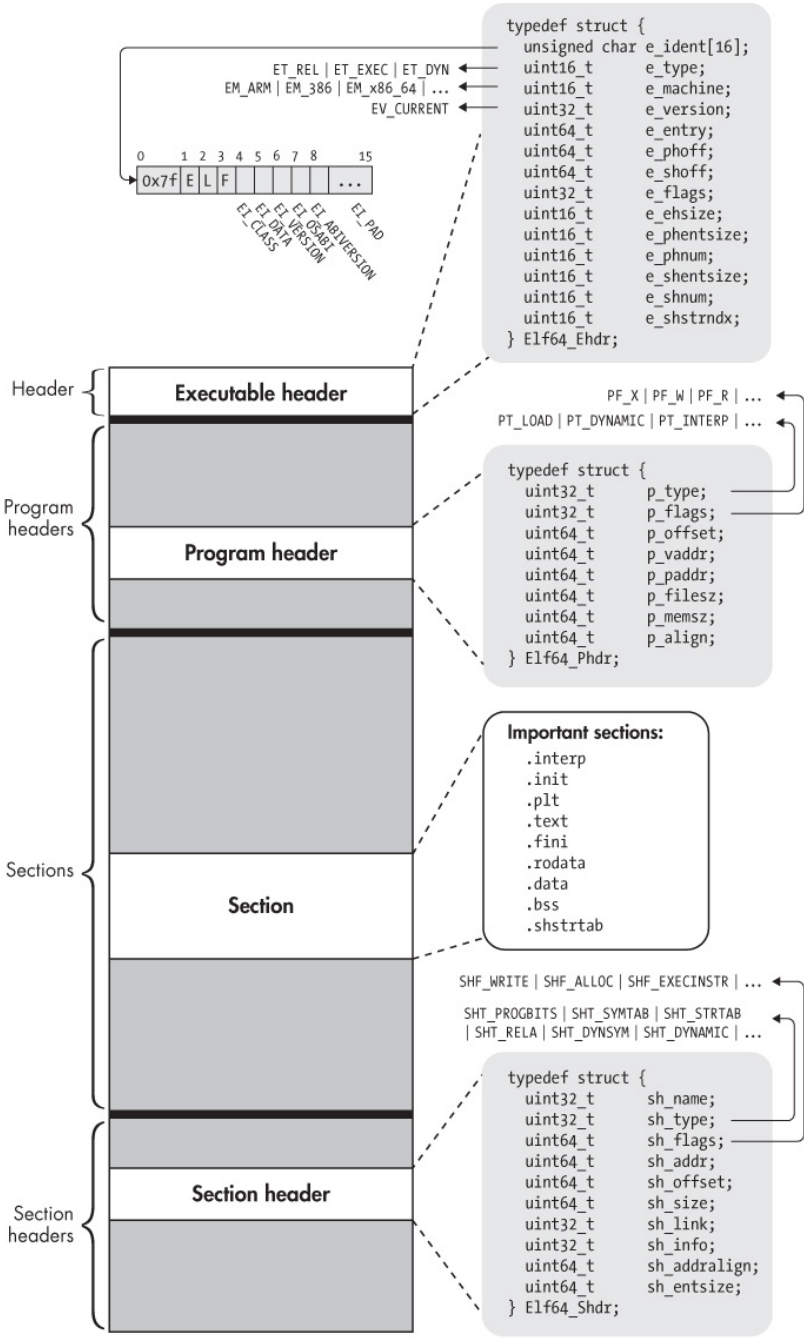
```
gcc <example.c> -o <example_bin>

# Давайте вызовем команду file к нашему двоичному исполняемому файлу
file <example_bin>

<example_bin>: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=1189a5182dc9274591321961fea250aa18893450, for GNU/Linux 3.2.0, with debug_info, not stripped
```

Утилита file сообщает, что мы имеем файл типа ELF 64-bit LSB pie executable. Добавилась аббревиатура PIE, что означает Position Independent Executable (исполняемый позиционно-независимый код). У вас может быть конечно и без PIE. Файл теперь исполняемый, а не перемещаемый, как после этапа ассемблирования. Важно также, что файл динамически скомпонован, то есть в нем используются библиотеки, не включенные в его состав, а разделенные с другими программами в системе. Наконец, слова interpreter /lib64/ld-linux-x86-64.so.2 говорит нам, какой *динамический компоновщик* будет использован для окончательного разрешения зависимостей от динамических библиотек на этапе загрузки исполняемого файла в память. Запустив двоичный файл, вы увидите, что он делает то, что и ожидалось. Рабочий двоичный файл! Какой огромный и сложный путь ради небольшой программы, верно?

Формат ELF

Имея общее представление о том, как выглядят и как работают двоичные файлы, мы можем перейти к деталям конкретного двоичного формата. В этой части статьи мы рассмотрим формат Executable And Linkable Format (ELF), подразумеваемый по умолчанию для двоичных файлов в Linux-системах. Именно с ним мы будем работать в этой книге.

Формат ELF используется для исполняемых файлов, объектных файлов, разделяемых библиотек и дампов памяти. Здесь я остановлюсь только на исполняемых ELF-файлах, но все те же концепции применимы и к другим файлам в этом формате.



Поскольку в этой статье мы будем иметь дело в основном с 64-разрядными двоичными файлами, в центре обсуждения будет 64-разрядный формат ELF. Впрочем, 32-разрядный формат похож и отличается главным образом

размером и порядком следования некоторых полей заголовков и других структур данных. Вам не составит труда перенести обсуждаемые здесь концепции на 32-разрядные двоичные ELF-файлы.

Когда впервые начинаешь подробно анализировать двоичный ELF-файл, эта сложность может показаться ошеломляющей. Но по существу ELF-файл содержит компоненты всего четырех типов: заголовок исполняемого файла, несколько необязательных заголовков программы, несколько секций и несколько необязательных заголовков секций, по одному на каждую секцию. Далее мы обсудим их по порядку.

Заголовок исполняемого файла в стандартном ELF-файле расположен первым, за ним идут заголовки программы, секции и заголовки секций.

Структура

В силу расширяемости ELF-файлов, структура может различаться для разных файлов. ELF-файл состоит из:

- заголовка ELF
- данных

Заголовок

Каждый ELF-файл начинается с заголовка исполняемого файла. Это всего лишь структурированная последовательность байтов, сообщающая нам, что это ELF-файл определенного типа и где искать все остальное содержимое. Формат заголовка исполняемого файла можно найти в определении типа в файле /usr/include/elf.h (там же определены другие относящиеся к ELF типы и константы) или в спецификации ELF1.

Как мы уже выяснили, любой ELF-файл начинается с заголовка, который представляет структуру, содержащую информацию о типе файла, его разрядности, типе архитектуры, версии ABI (Application Binary Interface), а также о том, где в файле искать все остальное. Формат структур заголовка как для 32-разрядных, так и для 64-разрядных ELF-файлов (как, впрочем, и форматы всех остальных структур ELF-файлов) можно посмотреть в файле `/usr/include/elf.h`.

Первые 16 байт заголовка (массив e_ident) служат для идентификации ELF-файла. Первые четыре байта — это магическая константа, состоящая из байта 0x7f, за которым идут ASCII-коды символов E, L и F. По наличию этих байтов загрузчик Linux (или, к примеру, утилита file) определяет, что перед ним именно ELF-файл (в PE-файлах Windows аналогичную функцию выполняет комбинация из ASCII-кодов символов M и Z). Следующие в этом массиве байты в файле elf.h обозначаются такими константами:

- EI_CLASS — байт определяет разрядность ELF-файла (значение 0x01 соответствует 32 разрядам, значение 0x02 — 64);
- EI_DATA — значение этого байта определяет порядок следования байтов данных при размещении их в памяти (Little Endian или Big Endian). Архитектура x86 использует размещение байтов Little Endian, поэтому значение этого байта будет равно 0x01;
- EI_VERSION — версия спецификации ELF-формата. Корректное значение в настоящее время — 0x01;
- EI_OSABI и EI_ABIVERSION определяют версию двоичного интерфейса и операционную систему, для которой откомпилирован файл. Для Linux первый из этих двух байтов обычно имеет значение 0x00 или 0x03, второй — 0x00.
- EI_PAD в настоящее время не несет никакой нагрузки и заполнено нулевыми значениями (в местах с индексами от 9 по 15).

Далее после массива e_ident расположены следующие поля:

- e_type — значение этого поля, как можно предположить, определяет тип ELF-файла (имеются в виду те типы, о которых мы говорили перед созданием примеров для нашего исследования). Некоторые интересующие нас в первую очередь значения этого поля:
- ET_EXEC — исполняемый файл (значение равно 0x02). Данное значение используется только для позиционно зависимых исполняемых ELF-файлов (например, тех, которые были скомпилированы GCC с опцией -no-pie);
- ET_REL — перемещаемый файл (значение в этом случае — 0x01);
- ET_DYN — разделяемый объектный файл (значение равно 0x03). Данное значение характерно как для динамически подключаемых библиотек (тех самых, которые обычно имеют расширение .so), так и для позиционно независимых исполняемых файлов. Как они различаются, мы обсудим чуть позже;
- e_machine — значением этого поля определяется архитектура, для которой собственно и создан ELF-файл. Поскольку мы в первую очередь говорим об архитектуре x86 в 64-разрядном исполнении, то значение этого поля будет EM_X86_64 (равно 0x42). Понятно, что можно встретить и другое значение, например EM_386 (для 32-разрядного случая архитектуры x86, равно 0x03) или, к примеру, EM_ARM (для процессоров ARM и равное 0x28);
- e_version — дублирует значение байта EI_VERSION из массива e_ident;
- e_entry — точка входа в ELF-файл, с которой начинается выполнение программы. Для позиционно зависимых файлов здесь лежит абсолютный виртуальный адрес начала выполнения программы, для позиционно независимого кода сюда пишется смещение относительно виртуального адреса начала образа ELF-файла, загруженного в память;
- e_phoff — смещение начала заголовков программ (обрати внимание: здесь, в отличие от точки входа, смещение относительно начала файла, а не виртуальный адрес);
- e_shoff — смещение начала заголовков секций (также относительно начала файла);
- e_flags — это поле содержит флаги, специфичные для конкретной архитектуры, для которой предназначен файл. В нашем случае (имеется в виду архитектура x86) поле имеет значение 0x00;
- e_ehsize — размер заголовка ELF-файла (для 32-разрядного он равен 52 байтам, для 64-разрядного — 64 байтам);

- e_phentsize — размер одной записи в разделе заголовков программ (размер одного заголовка);
- e_phnum — число записей в разделе заголовков программ (число заголовков программ);
- e_shentsize — размер одной записи в разделе заголовков секций (размер одного заголовка);
- e_shnum — число записей в разделе заголовков секций (число заголовков программ);
- e_shstrndx — это поле содержит индекс (то есть порядковый номер в разделе заголовков секций) заголовка одной из секций, которая называется .shstrtab. Эта секция содержит имена всех остальных секций ELF-файла в кодировке ASCII с завершающим нулем.

```
typedef struct {
     unsigned char e_ident[16];      /* Магическое число и другая информация */
     uint16_t e_type;                /* Тип объектного файла */
     uint16_t e_machine;             /* Архитектура */
     uint32_t e_version;             /* Версия объектного файла */
     uint64_t e_entry;               /* Виртуальный адрес точки входа */
     uint64_t e_phoff;               /* Смещение таблицы заголовков программы в файле */
     uint64_t e_shoff;               /* Смещение таблицы заголовков секций в файле */
     uint32_t e_flags;               /* Флаги, зависящие от процессора */
     uint16_t e_ehsize;              /* Размер заголовка ELF в байтах */
     uint16_t e_phentsize;           /* Размер записи таблицы заголовков программы */
     uint16_t e_phnum;               /* Количество записей в таблице заголовков программы */
     uint16_t e_shentsize;           /* Размер записи таблицы заголовков секций */
     uint16_t e_shnum;               /* Количество записей в таблице заголовков секций */
     uint16_t e_shstrndx;            /* Индекс таблицы строк в заголовке секции */
} Elf64_Ehdr;
```

Заголовок исполняемого файла представлен здесь в виде C-структуры (struct) Elf64_Ehdr. Заглянув в файл /usr/include/elf.h, вы увидите, что в определении структуры на самом деле фигурируют типы Elf64_Half и Elf64_Word. Это просто псевдонимы (typedef) целых типов uint16_t и uint32_t. Для простоты я раскрыл эти псевдонимы в коде выше.

Чтобы посмотреть на заголовок ELF-файла воочию, воспользуемся утилитой readelf (здесь опция -W указывает, что необходимо выводить полные строки, без ограничения в 80 символов в строке, а опция -h говорит, что вывести нужно именно заголовок):

```
readelf -W -h <example_elf_bin>
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1070
  Start of program headers:          64 (bytes into file)
  Start of section headers:          19664 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         38
  Section header string table index: 37
```

Как видно на листинге ниже, заголовок ELF начинается с «магического числа». Это «магическое число» даёт информацию о файле. Первые 4 байта определяют, что это ELF-файл (45=E,4c=L,46=F, перед ними стоит значение 7f).

Заголовок содержит следующие поля:

- Магическое число – байты 0–3 содержат магическое число 0x7F,'E','L','F', идентифицирующее файл как ELF.

- Данные — Далее идёт поле «данные», имеющее два варианта: 01 — LSB (Least Significant Bit), также известное как little-endian, либо 02 — MSB (Most Significant Bit, big-endian). Эти значения помогают интерпретировать остальные объекты в файле. Это важно, так как разные типы процессоров по разному обрабатывают структуры данных. В нашем случае используется LSB, так как процессор имеет архитектуру AMD64.

- Класс – байт 4 определяет размер слова архитектуры: ELFCLASS32 (1) для 32-битной версии или ELFCLASS64 (2) для 64-битной версии. После объявления типа ELF, следует поле класса. Это значение означает архитектуру,

для которой предназначен файл. Оно может равняться 01 (32-битная архитектура) или 02 (64-битная). Здесь мы видим 02, что переводится командой readelf как файл ELF64, то есть, другими словами, этот файл использует 64-битную архитектуру.

- Кодировка. Байт 5 определяет порядок байтов: ELFDATA2LSB (1) для прямого порядка байтов или ELFDATA2MSB (2) для прямого порядка байтов. Магической значение «01», представляющее собой номер версии. В настоящее время имеется только версия 01, поэтому это число не означает ничего интересного.

- Версия ELF. Байт 6 охватывает версию заголовка ELF. В настоящее время наиболее распространенным является EV_CURRENT (1).

- OS ABI — байт 7 идентифицирует целевой двоичный интерфейс приложения ОС (ABI). Общие значения: ELFOSABI_SYSV (0), ELFOSABI_LINUX (3) или ELFOSABI_NONE (255). Каждая операционная система имеет свой способ вызова функций, они имеют много общего, но, вдобавок, каждая система, имеет небольшие различия. Порядок вызова функции определяется «двоичным интерфейсом приложения» Application Binary Interface (ABI). Поля OS/ABI описывают, какой ABI используется, и его версию. В нашем случае, значение равно 00, это означает, что специфические расширения не используются. В выходных данных это показано как System V.

- Версия ABI — байт 8 указывает версию ABI, обычно 0. При необходимости, может быть указана версия ABI.

- Тип – полуслово в байтах 16–17, определяющее тип объектного файла – ET_EXEC (2) для исполняемых файлов, ET_DYN (3) для общих объектов, ET_REL (1) для перемещаемых файлов и т. д. Поле типа указывает, для чего предназначен файл. Вот несколько часто встречающихся типов файлов.

    - CORE (значение 4)

    - DYN (Shared object file), библиотека (значение 3)

    - EXEC (Executable file), исполняемый файл (значение 2)

    - REL (Relocatable file), файл до линковки (значение 1)

- Машина – полуслово в байтах 18–19, указывающее требуемую архитектуру ЦП. x86 — это EM_386 (3), x86_64 — это EM_X86_64 (62). Также в заголовке указывается ожидаемый тип машины (AMD64).

- Версия – слово в байтах 20–23 указывает версию формата ELF. Обычно EV_CURRENT (1).

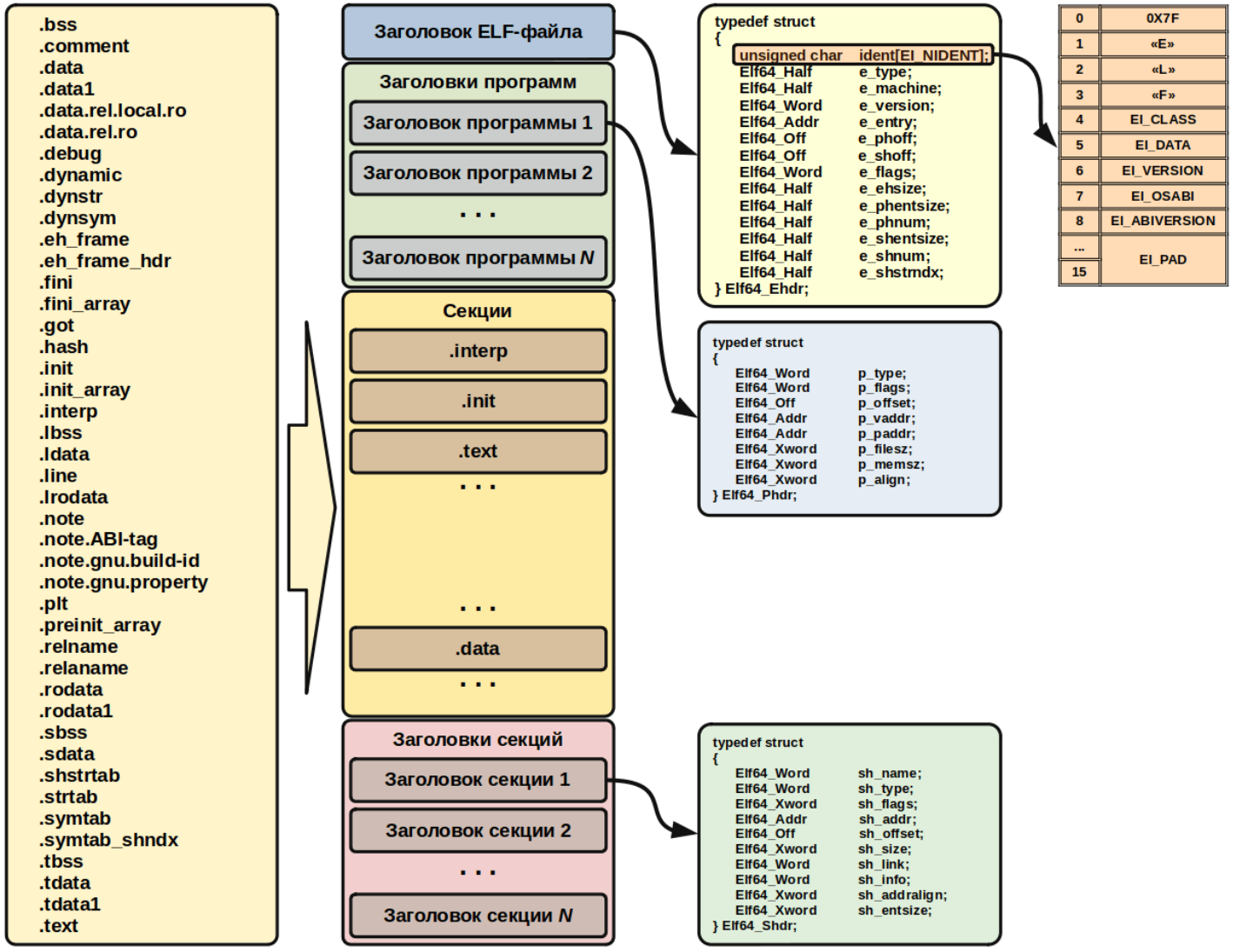
- Точка входа – слово в байтах 24–27 хранит адрес памяти для точки входа – начальной инструкции, выполняемой при загрузке исполняемого файла.

- Смещение/размер заголовка программы. Длинные слова в байтах 28–35 определяют смещение в байтах размер таблицы заголовка программы.

- Смещение/размер заголовка раздела. Длинные слова в байтах 40–47 определяют смещение в байтах размер таблицы заголовков разделов.

- Флаги. Слово в байтах 48–51 содержит флаги, специфичные для процессора. Общие значения — 0 или 1, что означает, что флаги не установлены.

Заголовок ELF является обязательным. Он нужен для того, чтобы данные корректно интерпретировались при линковке и исполнении. Для лучшего понимания внутренней работы ELF-файла, полезно знать, для чего используется эта информация.



Хотя некоторые поля могут быть просмотрены через readelf, их на самом деле больше. Например, можно узнать, для какого процессора предназначен файл. Используем hexdump, чтобы увидеть полный заголовок ELF и все значения.

```
hexdump -C -n 64 /bin/bash

00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  50 37 03 00 00 00 00 00  |..>.....P7......|
00000020  40 00 00 00 00 00 00 00  50 ed 14 00 00 00 00 00  |@.......P.......|
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1e 00 1d 00  |....@.8...@.....|
00000040
```

Хотя вы можете делать всё это в hex-дампе, имеет смысл использовать инструмент, который сделает работу за вас. Утилита dumpelf может быть полезна. Она показывает форматированный вывод, соответствующий заголовку ELF. Хорошо будет изучить, какие поля используются, и каковы их типичные значения.

Теперь, когда мы объяснили значения этих полей, время посмотреть на то, какая реальная магия за ними стоит, и перейти к следующим заголовкам!

Данные

Помимо заголовка, файлы ELF состоят из трёх частей.

- Программные заголовки или сегменты
- Заголовки секций или секции
- Данные

Перед тем, как мы погрузимся в эти заголовки, будет нелишним узнать, что файл ELF имеет два различных «вида». Один из них предназначен для линкера и разрешает исполнение кода (сегменты). Другой предназначен для команд и данных (секции). В зависимости от цели, используется соответствующий тип заголовка. Начнём с заголовка программы, который находится в исполняемых файлах ELF.

Заголовки программы

Файл ELF состоит из нуля или более сегментов, и описывает, как создать процесс, образ памяти для исполнения в рантайме. Когда ядро видит эти сегменты, оно размещает их в виртуальном адресном пространстве, используя системный вызов mmap(2). Другими словами, конвертирует заранее подготовленные инструкции в образ в памяти. Если ELF-файл является обычным бинарником, он требует эти программные заголовки, иначе он просто не будет работать. Эти заголовки используются, вместе с соответствующими структурами данных, для формирования процесса. Для разделяемых библиотек (shared libraries) процесс похож.

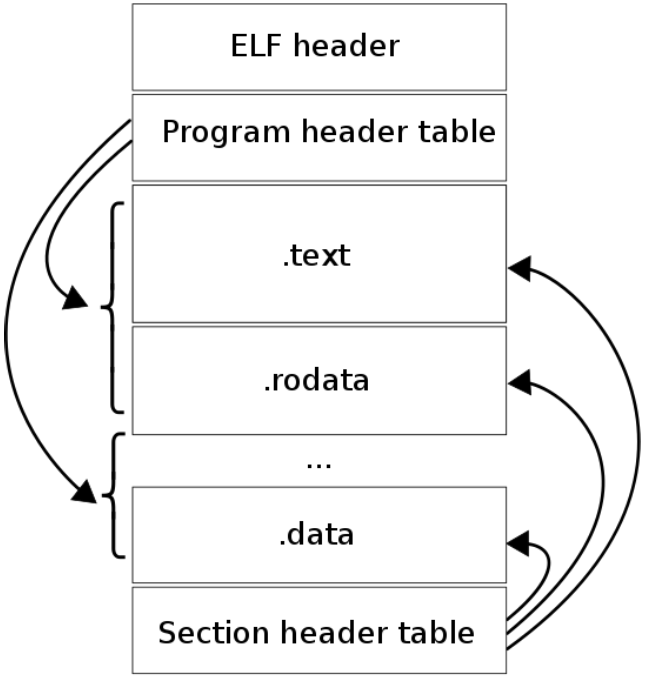


Таблица заголовков программы содержит один или несколько заголовков программы, которые описывают исполняемые сегменты, необходимые во время выполнения. Сегменты представляют собой смежные фрагменты кода и данных, которые будут отображены в адресное пространство процесса.

Типичные сегменты включают в себя:

- .text– Содержит исполняемый код, такой как функции и процедуры. Отображается как чтение/выполнение.
- .rodata– Данные только для чтения, такие как константы и строковые литералы. Сопоставлено только для чтения.
- .data/ .bss– Изменяемые переменные данных времени выполнения, которые инициализируются/обнуляются. Сопоставленное чтение/запись.
- .dynsym– Таблица символов динамического компоновщика, используемая для динамического связывания.
- .dynamic– Информация, необходимая для динамического связывания во время выполнения.
  Каждый заголовок программы имеет следующую структуру:
- Тип — тип сегмента, например PT_LOAD (загружаемый), PT_DYNAMIC (для динамического связывания), PT_INTERP (путь к программному интерпретатору) и т. д.

- Смещение – смещение в байтах от начала файла до первого байта сегмента.
- Виртуальный адрес — адрес, по которому этот сегмент должен быть сопоставлен во время выполнения.
- Физический адрес – адрес сегмента после загрузки в память (необязательно).
- Размер файла – размер сегмента в байтах в файле ELF.
- Размер памяти — необходимый размер памяти сегмента при загрузке, потенциально превышающий размер файла.
- Флаги — разрешения, такие как PF_X (выполнение), PF_W (запись), PF_R (чтение).
- Выравнивание – необходимое выравнивание для сегмента.

Вот пример заголовка программы, напечатанный readelf:

```
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000201000 0x0000000000201000
                 0x000000000001c864 0x000000000001c864  R E    200000
```

Здесь показан загружаемый сегмент со смещением файла 0, сопоставленный с виртуальным адресом 0x201000, с разрешениями RE (чтение + выполнение), размером 0x1c864 байта и выравниванием 200 000 байт.

Заголовки секции определяют все секции файла. Как уже было сказано, эта информация используется для линковки и релокации.

Секции появляются в ELF-файле после того, как компилятор GNU C преобразует код C в ассемблер, и ассемблер GNU создаёт объекты.

Как показано на рисунке вверху, сегмент может иметь 0 или более секций. Для исполняемых файлов существует четыре главных секций: .text, .data, .rodata, и .bss. Каждая из этих секций загружается с различными правами доступа, которые можно посмотреть с помощью readelf -S.

Секционное представление ELF-файлов

Код и данные в ELF-файле логически разделены на секции, которые представляют собой непересекающиеся смежные блоки, расположенные в файле друг за другом без промежутков. У секций нет определенной общей структуры: в каждой секции организация размещения данных или кода зависит от ее назначения. Более того, некоторые секции вообще могут не иметь какой-либо структуры, а представлять собой неструктурированный блок кода или данных. Каждая секция описывается своим заголовком, который хранится в таблице заголовков секций. В заголовке перечислены свойства секции, а также местонахождение содержимого самой секции в файле.

Вообще, если внимательно посмотреть спецификацию ELF-файла, то можно увидеть, что деление на секции предназначено для организации работы компоновщика, а с точки зрения исполнения файла секционная организация не несет никакой полезной информации. То есть не нуждающиеся в компоновке ELF-файлы (неперемещаемые исполняемые файлы) могут не иметь таблицы заголовков секций (и во многих случаях ее не имеют). Для загрузки в память процесса и выполнения ELF-файлов используется еще одна логическая организация — сегментная, о которой мы поговорим ниже. Если в ELF-файле нет таблицы заголовков секций, поле e_shoff в заголовке будет равно нулю.

Если мы введем команду `readelf -S -W <example_bin>`, мы можем увидеть:

```
There are 38 section headers, starting at offset 0x4cd0:
  [ 1] .interp           PROGBITS        0000000000000318 000318 00001c 00   A  0   0  1
  [ 2] .note.gnu.property NOTE            0000000000000338 000338 000040 00   A  0   0  8
  [ 3] .note.gnu.build-id NOTE            0000000000000378 000378 000024 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            000000000000039c 00039c 000020 00   A  0   0  4
```

```
  [ 5] .gnu.hash         GNU_HASH        00000000000003c0 0003c0 000024 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000000003e8 0003e8 0000a8 18   A  7   1  8
  [ 7] .dynstr           STRTAB          0000000000000490 000490 00008f 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          0000000000000520 000520 00000e 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0000000000000530 000530 000030 00   A  7   1  8
  [10] .rela.dyn         RELA            0000000000000560 000560 0000c0 18   A  6   0  8
  [11] .rela.plt         RELA            0000000000000620 000620 000018 18  AI  6  24  8
  [12] .init             PROGBITS        0000000000001000 001000 000017 00  AX  0   0  4
  [13] .plt              PROGBITS        0000000000001020 001020 000020 10  AX  0   0 16
  [14] .plt.got          PROGBITS        0000000000001040 001040 000008 08  AX  0   0  8
  [15] .text             PROGBITS        0000000000001050 001050 000109 00  AX  0   0 16
  [16] .fini             PROGBITS        000000000000115c 00115c 000009 00  AX  0   0  4
  [17] .rodata           PROGBITS        0000000000002000 002000 000013 00   A  0   0  4
  [18] .eh_frame_hdr     PROGBITS        0000000000002014 002014 00002c 00   A  0   0  4
  [19] .eh_frame         PROGBITS        0000000000002040 002040 0000a4 00   A  0   0  8
  [20] .init_array       INIT_ARRAY      0000000000003dd0 002dd0 000008 08  WA  0   0  8
  [21] .fini_array       FINI_ARRAY      0000000000003dd8 002dd8 000008 08  WA  0   0  8
  [22] .dynamic          DYNAMIC         0000000000003de0 002de0 0001e0 10  WA  7   0  8
  [23] .got              PROGBITS        0000000000003fc0 002fc0 000028 08  WA  0   0  8
  [24] .got.plt          PROGBITS        0000000000003fe8 002fe8 000020 08  WA  0   0  8
  [25] .data             PROGBITS        0000000000004008 003008 000010 00  WA  0   0  8
  [26] .bss              NOBITS          0000000000004018 003018 000008 00  WA  0   0  1
  [27] .comment          PROGBITS        0000000000000000 003018 000012 01  MS  0   0  1
  [28] .debug_aranges    PROGBITS        0000000000000000 003030 000120 00      0   0 16
  [29] .debug_info       PROGBITS        0000000000000000 003150 000643 00      0   0  1
  [30] .debug_abbrev     PROGBITS        0000000000000000 003793 00020f 00      0   0  1
  [31] .debug_line       PROGBITS        0000000000000000 0039a2 000236 00      0   0  1
  [32] .debug_str        PROGBITS        0000000000000000 003bd8 000863 01  MS  0   0  1
  [33] .debug_line_str   PROGBITS        0000000000000000 00443b 000155 01  MS  0   0  1
  [34] .debug_rnglists   PROGBITS        0000000000000000 004590 000059 00      0   0  1
  [35] .symtab           SYMTAB          0000000000000000 0045f0 000378 18     36  19  8
  [36] .strtab           STRTAB          0000000000000000 004968 0001e7 00      0   0  1
  [37] .shstrtab         STRTAB          0000000000000000 004b4f 00017a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

Заголовки секции

Заголовки секции определяют все секции файла. Как уже было сказано, эта информация используется для линковки и релокации.

Секции появляются в ELF-файле после того, как компилятор GNU C преобразует код C в ассемблер, и ассемблер GNU создаёт объекты.

Как показано на рисунке вверху, сегмент может иметь 0 или более секций. Для исполняемых файлов существует четыре главных секций: .text, .data, .rodata, и .bss. Каждая из этих секций загружается с различными правами доступа, которые можно посмотреть с помощью readelf -S.

Итак, как мы выяснили, в заголовках секций (если в ELF-файле есть раздел с таблицей заголовков секций) содержится информация о свойствах и местонахождении той или иной секции. Заголовки секций представляют собой структуру, описание которой можно найти в файле /usr/include/elf.h (там эти структуры носят имена Elf64_Shdr и Elf32_Shdr для 64- и 32-разрядных файлов соответственно).

Разберемся с назначением каждого из полей этой структуры:

- sh_name — содержит индекс имени секции (здесь имеется в виду номер байта, с которого начинается имя секции) в таблице строк, которая, в свою очередь, содержится в секции .shstrtab. Другими словами, все имена секций хранятся в специальной секции .shstrtab (о ней мы уже говорили, когда рассматривали заголовок ELF-файла), а в поле sh_name находится значение смещения начала строки с именем секции, к которой относится данный заголовок, от начала секции sh_strtab;
- sh_type — тип секции, который определяет ее содержимое. Из всех возможных типов (а они также определены в /usr/include/elf.h) наибольший интерес представляют следующие:
- SHT_NULL — неиспользуемая (пустая) секция. Согласно спецификации, первая запись в таблице заголовков секций должна быть именно такой (значение, как нетрудно догадаться, равно 0x00);
- SHT_PROGBITS — секция содержит данные или код для выполнения (значение равно 0x01);
- SHT_SYMTAB — секция содержит так называемую таблицу статических символов (под символами в данном случае понимаются имена функций или переменных). Каждая запись в этой секции представляет собой структуру Elf64_Sym (или Elf32_Sym для 32-разрядных файлов), которая определена в usr/include/elf.h. Как правило, секция с таблицей статических символов носит имя .symtab, каждая запись в этой секции нужна для сопоставления того или иного символа с местонахождением функции или переменной, имя которой и определено данным символом. Все это в подавляющем большинстве случаев нужно, чтобы облегчить отладку программы, а непосредственно для выполнения ELF-файла эта секция не используется (и во многих случаях после отладки программы ее из файла удаляют с помощью утилиты strip). Значение равно 0x02;
- SHT_DYNSYM — таблица символов, используемая динамическим компоновщиком при компоновке программы (числовое значение равно 0x0b). Каждая запись этой секции также представляет собой структуру Elf64_Sym (или Elf32_Sym). Как правило, секция с таблицей динамических символов носит имя .dynsym. Более подробно о секциях .symtab и .dynsym поговорим чуть позже;
- SHT_STRTAB — в секциях такого типа хранятся строки в кодировке ASCII с завершающим нулем (в частности, уже знакомая нам секция .shstrtab имеет именно такой тип). Значение равно 0x03;
- SHT_REL, SHT_RELA — секции этого типа содержат записи о перемещениях, причем формат каждой записи определен структурами Elf64_Rel (Elf32_Rel) и Elf64_Rela (Elf32_Rela), опять же описанными в elf.h. Непосредственно с организацией перемещений мы разберемся чуть позже;

- SHT_DYNAMIC — секция этого типа хранит информацию, необходимую для динамической компоновки (числовое значение — 0x06). Формат одной записи в такой секции описывается структурой Elf64_Dyn (Elf32_Dyn) в файле elf.h;
- SHT_NOBITS — секция такого типа не занимает места в файле. По сути, наличие такой секции является директивой о выделении необходимого количества памяти для неинициализированных переменных на этапе загрузки ELF-файла в память и подготовки его к выполнению (обычно такая секция носит имя .bss). Числовое значение данной константы равно 0x08;
- sh_flags — содержит дополнительную информацию о секции. Из всего многообразия значений в первую очередь интересны флаги:
- SHF_WRITE — флаг, имеющий значение 0x01 и говорящий о том, что в секцию возможна запись данных (здесь необходимо учитывать возможность объединить несколько флагов посредством операции «или»);
- SHF_ALLOC — наличие этого флага говорит о том, что содержимое секции должно быть загружено в виртуальную память при подготовке программы к выполнению (хотя нужно помнить, что ELF-файл загружается в память с применением сегментного представления файла, ну а сегмент есть не что иное, как объединение нескольких секций). Числовое значение равно 0x02;
- SHF_EXECINSTR — такой флаг показывает, что секция содержит исполняемый код (значение равно 0x04);
- SHF_STRINGS — элементы данных в секции с таким флагом состоят из символьных строк, завершающихся нулевым символом (значение — 0x20);
- sh_addr — виртуальный адрес секции. Хоть мы и говорили, что секционное представление не используется при размещении файла в памяти, в некоторых случаях компоновщику необходимо знать, по каким адресам будут размещены те или иные участки кода или данных на этапе выполнения, чтобы обеспечивать перемещения. Для этого и предусмотрено данное поле. Если секция не должна грузиться в память при выполнении файла, то значение этого поля равно нулю;
- sh_offset — смещение секции в файле (относительно его начала);
- sh_size — размер секции в байтах;
- sh_linc — содержит индекс (в таблице заголовков секций) секции, с которой связана данная секция;
- sh_info — дополнительная информация о секции (значение зависит от типа секции);
- sh_addralign — здесь задаются требования к выравниванию секции в памяти (если содержит значения 0x00 или 0x01, то выравнивание не требуется);
- sh_entsize — в некоторых секциях, например таблице символов .symtab, лежат записи определенных структур (к примеру, Elf64_Sym), и для таких секций поле содержит размер одной записи секции в байтах.

Если секция не относится к такому виду, то поле содержит нулевое значение.

.text

Содержит исполняемый код. Он будет упакован в сегмент с правами на чтение и на исполнение. Он загружается один раз, и его содержание не изменяется. Это можно увидеть с помощью утилиты objdump.

Здесь как раз и находится весь тот код, ради выполнения которого и была написана программа и на который указывает поле e_entry заголовка ELF-файла. Однако если посмотреть дизассемблированный листинг этой секции, то вопреки ожиданиям мы увидим, что по адресу, на который указывает точка входа, лежит не функция main(), а некая функция `_start`, после которой присутствует еще несколько функций, выполняющих подготовку к запуску программы (например, deregister_tm_clones, register_tm_clones и frame_dummy).

Функция `_start` считывает параметры командной строки (если они есть) и вызывает функцию `__libc_start_main`. И уже эта функция вызывает на выполнение функцию main(), где содержится основной код программы.

```
12 .text 0000a3e9 0000000000402120 0000000000402120 00002120 2**4
CONTENTS, ALLOC, LOAD, READONLY, CODE
```

.data

Инициализированные данные, с правами на чтение и запись. Секция для хранения инициализированных переменных, изменение которых возможно в ходе выполнения программы (соответственно, эта секция имеет флаг SHF_WRITE).

.rodata

Инициализированные данные, с правами только на чтение. (=A). В этой секции хранятся константные значения, то есть значения, которые не подлежат изменению в ходе выполнения программы.

.bss

Неинициализированные данные, с правами на чтение/запись. (=WA). Секция .bss предназначена для неинициализированных переменных. Если секции .data и .rodata имеют тип SHT_PROGBITS, эта секция, как мы уже отмечали выше, имеет тип SHT_NOBITS. Данная секция не занимает место в ELF-файле, поскольку и так понятно, что неинициализированные переменные равны нулю, а хранить эти нули в ELF-файле нет никакого смысла.

```
[24] .data PROGBITS 00000000006172e0 000172e0
0000000000000100 0000000000000000 WA 0 0 8
[25] .bss NOBITS 00000000006173e0 000173e0
```

```
0000000000021110 0000000000000000 WA 0 0 32
```

Команды для просмотра секций и заголовков:

- dumpelf
- elfls -p <example_bin>
- eu-readelf –section-headers <example_bin>
- readelf -S <example_bin>
- objdump -h <example_bin>

Группы секций

Некоторые секции могут быть сгруппированы, как если бы они формировали единое целое. Новые линкеры поддерживают такую функциональность. Но пока такое встречается не часто.

```
readelf -g /bin/ps
```

```
There are no section groups in this file.
```

Хотя это может показаться не слишком интересным, большие преимущества даёт знание инструментов анализа ELF-файлов.

Чтобы определить, какие внешние библиотеки использованы, просто используйте ldd на том же бинарнике.

Для сбора дополнительной информации мы можем использовать отличный скрипт под названием ldd (List Dynamic Dependencies). Этот сценарий перечисляет все динамические библиотеки, необходимые для конкретного исполняемого файла. И, в отличие от методов статического перечисления, этот инструмент описывает полный путь (!) Библиотеки, которую мы собираемся использовать. Например, вывод для нашей программы "Hello, Habr"

```
ldd <example_bin>
```

```
linux-vdso.so.1 (0x00007ffc799ab000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f65bf000000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f65bf2f2000)
```

Сегментное представление ELF-файлов

Как мы уже говорили, сегментное представление используется компоновщиком при загрузке ELF-файла в процесс для выполнения. Этот тип представления дает таблица заголовков программы (повторюсь, если исследуемый файл не предназначен для выполнения, то эта таблица может отсутствовать). Таблица заголовков программы описывается структурой Elf32_Phdr или Elf64_Phdr из уже знакомого нам файла /usr/include/elf.h.

В целом сегмент может включать в себя ноль и более секций, то есть объединяет секции в один блок. Здесь может возникнуть справедливый вопрос: почему же тогда в сегменте может быть ноль секций?

Дело в том, что некоторые типы сегментов при описании их в ELF-файле не имеют внутри себя никаких секций (то есть они пустые). К примеру, пустые секции имеет заголовок, с которого начинается таблица заголовков программы (он самый первый в таблице и как раз и сигнализирует о том, что с этого места начинается таблица заголовков), или сегмент, хранящий информацию о стеке (имеет тип заголовка PT_GNU_STACK).

Вывести информацию о сегментах можно следующим образом:

```
readelf -W --segments bin/hello
```

```
Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1070
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0002d8 0x0002d8 R   0x8
  INTERP         0x000318 0x0000000000000318 0x0000000000000318 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x000638 0x000638 R   0x1000
  LOAD           0x001000 0x0000000000001000 0x0000000000001000 0x000165 0x000165 R E 0x1000
  LOAD           0x002000 0x0000000000002000 0x0000000000002000 0x0000e4 0x0000e4 R   0x1000
  LOAD           0x002dd0 0x0000000000003dd0 0x0000000000003dd0 0x000248 0x000250 RW  0x1000
  DYNAMIC        0x002de0 0x0000000000003de0 0x0000000000003de0 0x0001e0 0x0001e0 RW  0x8
```

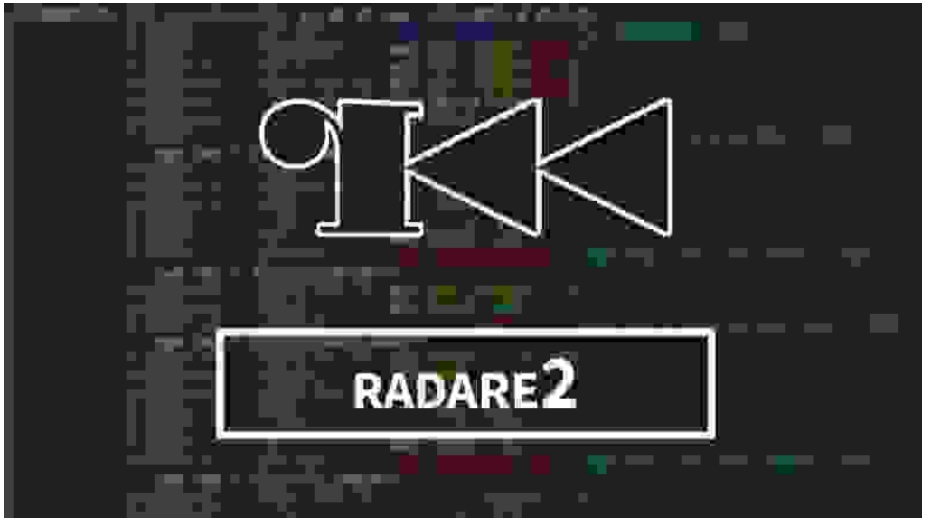
```
  NOTE           0x000338 0x0000000000000338 0x0000000000000338 0x000040 0x000040 R   0x8
  NOTE           0x000378 0x0000000000000378 0x0000000000000378 0x000044 0x000044 R   0x4
  GNU_PROPERTY   0x000338 0x0000000000000338 0x0000000000000338 0x000040 0x000040 R   0x8
  GNU_EH_FRAME   0x002014 0x0000000000002014 0x0000000000002014 0x00002c 0x00002c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x002dd0 0x0000000000003dd0 0x0000000000003dd0 0x000230 0x000230 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .got.plt .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
   10     .eh_frame_hdr
   11
   12     .init_array .fini_array .dynamic .got
```

Итак, основные поля заголовка программы таковы:

- p_type — это поле определяет тип сегмента. Возможные значения также представлены в файле /usr/include/elf.h. Наиболее важные значения:
- PT_HDR — с данного заголовка начинается таблица заголовков программы (как мы уже говорили, описанный этим заголовком сегмент пустой);
- PT_LOAD — сегмент этого типа предназначен для загрузки в память на этапе подготовки процесса к выполнению;
- PT_INTERP — этот сегмент содержит секцию .interp, в которой лежит имя интерпретатора, используемого при загрузке ELF-файла;
- PT_DYNAMIC — в этом сегменте содержится секция .dynamic с информацией о динамических связях, которая говорит, как разбирать и подготавливать ELF-файл к выполнению (более подробно об этой секции поговорим в следующий раз);
- PT_GNU_STACK — здесь хранится информация о стеке, которая определяется значением флага pt_flags и показывает, что стек не должен исполняться (значение pt_flags равно PF_R и PF_W). Отсутствие данного сегмента говорит о том, что содержимое стека может быть исполнено (что, как ты наверняка знаешь, не есть хорошо);
- p_flags — определяет права доступа к сегменту во время выполнения ELF-файла (самые главные значения этого поля: PF_R — чтение, PF_W — запись, PF_X — выполнение);
- p_offset, p_vaddr и p_filesz — значения этих полей аналогичны значениям полей sh_offset, sh_addr и sh_size в заголовке секции;
- p_addr — обычно значение этого поля для современных версий Linux равно нулю (хотя изначально здесь хранился адрес в физической памяти, по которому должен быть загружен сегмент);
- p_memsz — размер сегмента в памяти (если вспомнить о наличии секции .bss, которая может входить в состав сегмента, то станет понятно, почему размер сегмента в файле может быть меньше, чем размер сегмента в памяти);
- p_align — это поле аналогично полю sh_addralign в заголовке секции.

Инструменты анализа двоичных файлов

radare2



Небольшой экскурс в историю. Проект radare начал разрабатывать хакер с ником pancake в 2006 году, и долгое время, по сути, он был единственным разработчиком. Созданный фреймворк обладал простым консольным интерфейсом для работы как шестнадцатеричный редактор, поддерживающий 64-битную архитектуру. Это позволяло находить и восстанавливать данные с жестких дисков. Поэтому его еще называли инструментом для компьютерной криминалистической экспертизы. Но в 2010 год произошел «редизайн» фреймворка, после чего проект стал разрастаться и пополняться новым функционалом, позволяющим использовать его не только как редактор, но и как дизассемблер, анализатор и кода, и шелл-кодов. На данный момент этот фреймворк используют знаменитые CTF-команды (Dragon Sector) и вирусные аналитики (MalwareMustDie и AlienVault), причем последние представляют его на своем воркшопе на Black Hat.

Radare2 это фреймворк для анализа бинарных файлов. Он включает в себя большое количество утилит. Изначально он развивался как шестнадцатеричный редактор для поиска и восстановления данных, затем он обрастал функционалом и на текущий момент стал мощным фреймворком для анализа данных. В этой статье я расскажу как с помощью фреймворка Radare2 произвести анализ логики работы программы, а также опишу основные элементы языка ассемблера, которые необходимы для проведения реверс инжиниринга.

Тулкит Radare2 создан Серджи Альваресом (Sergi Alvarez). Цифра 2 подразумевает, что код был полностью переписан по сравнению с первой версией. Сейчас он используется многими исследователями, для изучения работы кода.

Бесплатная и открытая цепочка инструментов для упрощения выполнения некоторых низкоуровневых задач, таких как криминалистика, реверс-инжиниринг программного обеспечения, эксплуатация, отладка и т. д.

Он состоит из множества библиотек (которые расширяются плагинами) и программ, которые можно автоматизировать практически с помощью любого языка программирования.

r2 — это полная переработка radare. Он предоставляет набор библиотек, инструментов и плагинов для упрощения задач обратного проектирования. Распространяется в основном под лицензией LGPLv3, каждый плагин может иметь разные лицензии (см. r2 -L, rasm2 -L, ...).

Проект radare2 начинался как простой шестнадцатеричный редактор с командной строкой, ориентированный на судебную экспертизу. Сегодня r2 — это многофункциональный низкоуровневый инструмент командной строки с поддержкой сценариев с помощью встроенного интерпретатора Javascript или через r2pipe.

r2 может редактировать файлы на локальных жестких дисках, просматривать память ядра и отлаживать программы локально или через удаленный сервер GDB. Широкая поддержка архитектуры r2 позволяет анализировать, эмулировать, отлаживать, изменять и дизассемблировать любые двоичные файлы.

Скорее всего, он есть в репозиториях вашего дистрибутива.

Функции radare2:

- Пакетный режим, командная строка, визуальный режим и интерактивные панели
- Встроенный веб-сервер со сценариями js и webui
- Сборка и дизассемблирование большого списка процессоров
- Работает в Windows и любой другой версии UNIX
- Анализ и эмуляция кода с помощью ESIL
- Собственный отладчик и GDB, WINDBG, QNX и FRIDA
- Навигация по графикам потока управления ascii-art
- Возможность исправлять двоичные файлы, изменять код или данные

- Поиск шаблонов, магических заголовков, сигнатур функций
- Легко расширять и изменять
- Командная строка, C API, скрипт с r2pipe в любой язык

Radare2 представляет собой комплект из нескольких утилит:

- radare2 (r2) — Шестнадцатеричный редактор, дизассемблер и отладчик с расширенным интерфейсом командной строки. Позволяет работать с различными устройствами ввода вывода, такими как диски, удаленные устройства, отлаживаемые процессы и др., а также работать с ними как с простыми файлами.
- rabin2 — Используется для получения информации об исполняемых бинарных файлах.
- rasm2 — Позволяет производить преобразования из опкода в машинный код и обратно. Поддерживает большое количество архитектур.
- rahash2 — Утилита, предназначенная для расчета контрольных сумм. Поддерживает множество алгоритмов, позволяет получить контрольную сумму целого файла, его части или произвольной строки.
- radiff2 — Утилита, для сравнения бинарных файлов, поддерживает множество алгоритмов, умеет сравнивать блоки кода исполняемых файлов.
- rafind2 — Утилита для поиска последовательности байт.
- ragg2 — Утилита для компиляции небольших программ.
- rarun2 — Утилита, способная запускать анализируемую программу с различными настройками окружения.
- rax2 — Небольшой калькулятор, позволяющий производить простые вычисления в различных системах счисления.

Основным недостатком, который препятствует распространенности фреймворка, является отсутствие качественного GUI. Имеются сторонние реализации, но сожалению они не слишком удобные. Также стоит отметить наличие встроенного веб интерфейса.

Radare2 чаще всего применяется как инструмент реверс инжиниринга, в качестве продвинутого дизассемблера. Рассматривать Radare2 мы будем именно как дизассембрер и произведем анализ простого crackme.

Официальный сайт, Дистрибутив

binutils

Набор инструментального ПО для обращения с объектным кодом в объектных файтах различного формата.

Современные версии были изначально написаны программистами из Cygnus Solutions, используя библиотеку libbfd (Binary File Descriptor). Эти утилиты обычно используются в сочетании с GCC, make и отладчиком GNU.

Изначально пакет состоял только из небольших утилит, но позже в релизы были включены GNU Assembler (GAS) и GNU linker (GLD), так как их функциональные назначения достаточно сильно связаны. Большая часть утилит — довольно простые программы. Основные сложные части вынесены в общие библиотеки: libbfd и libopcodes.

Программные утилиты, включающие в себя разные программы. Большинство Linux-систем имеют установленный пакет binutils. Другие пакеты могут помочь вам увидеть больше информации. Правильный тулкит упростит вашу работу, особенно если вы занимаетесь анализом ELF-файлов. Я собрал здесь список пакетов и утилит для анализа ELF-файлов.

GNU Binutils — это набор бинарных инструментов. Основные из них:

- ld — компоновщик GNU.
- as — ассемблер GNU.
- gold — новый, более быстрый компоновщик только для ELF.

Но они также включают в себя:

- addr2line — преобразует адреса в имена файлов и номера строк.
- ar — Утилита для создания, изменения и извлечения из архивов.
- c++filt — Фильтр для распутывания закодированных символов C++.
- dlltool — Создает файлы для создания и использования DLL.
- elfedit — позволяет изменять файлы формата ELF.
- gprof — отображает информацию о профилировании.
- gprofng — собирает и отображает данные о производительности приложения.
- nlmconv — преобразует объектный код в NLM.
- nm — список символов из объектных файлов.
- objcopy — копирует и переводит объектные файлы.
- objdump — отображает информацию из объектных файлов.
- ranlib — генерирует индекс содержимого архива.

- readelf — отображает информацию из любого объектного файла формата ELF.
- size — отображает размеры разделов объекта или архивного файла.
- strings — выводит список печатаемых строк из файлов.
- strip — отбрасывает символы.
- Windmc — компилятор сообщений, совместимый с Windows.
- Windres — компилятор файлов ресурсов Windows.

А также некоторые библиотеки:

- libbfd — библиотека для работы с двоичными файлами различных форматов.
- libctf — библиотека для управления форматом отладки CTF.
- libopcodes — библиотека для ассемблирования и дизассемблирования различных языков ассемблера.
- libsframe — библиотека для управления форматом отладки SFRAME.
  Большинство этих программ используют BFD, библиотеку дескрипторов двоичных файлов, для выполнения низкоуровневых манипуляций. Многие из них также используют библиотеку кодов операций для сборки и дизассемблирования машинных инструкций.

Программы binutils были портированы на большинство основных вариантов Unix, а также на системы Wintel, и основная причина их существования — предоставить системе GNU (и GNU/Linux) возможность компилировать и связывать программы.

Заключение

Формат ELF очень сложен и интересный. И если вы хотите изучать информационную безопасность, системное программирование и администрирование, то вам следует знать его устройство.

Полезные ссылки:

- OSDev Wiki — ELF
- ELF Format Cheatsheet / шпаргалка по ELF
- Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification v1.2 / PDF
- Строение GNU/Linux
- Openbsd man — elf
- ELF standart / pdf