Executable and Linkable Format im Detail

Nachdrückliche Vorladung

ELF ist das Standardformat für Programme und dynamische Bibliotheken. Unter Linux sorgen ausgeklügelte Mechanismen dafür, dass solche Binaries korrekt im Speicher landen und starten. Per Prelink lässt sich dieser Vorgang oft sogar beschleunigen. Frank Peters

Bei modernen Betriebssystemen mit ihren virtuellen Speicherverwaltungen ist das Laden der Programme nicht trivial. Das System benötigt Information, wie es mit Binärcode und Daten umgehen soll. Linux benutzt dazu das offene Executable and Linkable Format (ELF, [1]), das ursprünglich die Unix System Laboratories entwickelt haben. ELF hat das frühere Standardformat A.out verdrängt. In allen modernen Linux-Distributionen sind nicht nur ausführbare Dateien, Bibliotheken und Objektfiles in ELF kodiert, sondern auch der Kernel und dessen ladbare Module.

Alle Programme bestehen aus mehreren Segmenten. Die wichtigsten sind das Text-(Code-)Segment mit dem ausführbaren Programmcode und das Daten-Segment, das die Programmdaten wie etwa Variablen aufnimmt. Startet das Programm, kümmert sich das Betriebssystem um das Laden nebst Adresszuordnung der Segmente. ELF unterscheidet drei Typen von Dateien:

Relozierbare Binaries

Eine relozierbare Datei (relocatable, verschiebbar) enthält Code und Daten, die für das Binden mit anderen Dateien geeignet sind. Sie besitzen noch keine festen Adressen für Code und Daten und dürfen daher beliebig im Adressraum wandern. Die Positionen, an denen zur Laufzeit eine feste Adresse stehen muss, sind speziell gekennzeichnet. Das Kennzeichnen findet nicht im Code statt, sondern über einen Eintrag in der Relokationstabelle.

Übersetzungsprograme wie der C-Compiler aus der GNU Compiler Collection erzeugen typisch solche Dateien, auch als Objektdateien oder -module (»*.obj«) bezeichnet. Der Compiler erzeugt jedes Objektmodul so, als besäße es einen eigenen Adressraum. Benutzt eine Objektdatei eine Methode aus einer anderen Objektdatei, wird ein Relokationseintrag erstellt. Er signalisiert, dass hier später die wirkliche Adresse der Methode einzutragen ist.

Objektmodule lassen sich in Bibliotheken zusammenfassen, wobei zwischen statischen und dynamischen Bibliotheken zu unterscheiden ist. Eine statische ist nur ein Archiv von Objektmodulen. Das so genannte statische Binden kopiert alle benötigten Objektmodule (auch aus statischen Bibliotheken) zusammen in eine ausführbare Datei. Der Binder hat die Aufgabe, aus den einzelnen Adressräumen einen gemeinsamen aufzubauen. Funktionen aus dynamischen Bibliotheken beziehungsweise Objektmodulen kopiert der Binder hingegen nicht in die ausführbare Datei, er legt nur einen Verweis auf die zu benutzende dynamische Bibliothek und die benötigte Funktion an.

Ausführbare Binaries

Eine ausführbare (executable) Datei enthält Code und Daten, die ein Prozessor auszuführen vermag. Sie bestimmt maßgeblich das Speicherlayout des Prozesses. Der GNU Binder »ld« erzeugt zum Beispiel diese Art Dateien. Typisch bestehen ausführbare Dateien nicht nur aus Objektmodulen, sondern auch aus

Informationen aus dynamischen Bibliotheken, die vor der Ausführung benötigt werden. Der dynamische Binder (auch Laufzeitbinder genannt) übernimmt diese Aufgabe. Er sucht die benötigten Bibliotheken und blendet sie in den Adressraum des Prozesses ein, um die vorhandenen Verweise auf die richtigen Adressen zu lenken.

Dynamische Bibliotheken

Die Dynamic shared Libraries/Objects (DSOs) sind eine Mischung aus relozierbaren und ausführbaren Binaries. Unter Linux sind sie an der Endung »*.so« zu erkennen. Auch dieser Dateityp enthält Code und Daten, die der Binder verwendet, um Referenzen aufzulösen. Der dynamische Binder nutzt diese Informationen, um ein Prozessabbild mit einer ausführbaren Datei herzustellen. Dazu müssen Code und Daten positionsunabhängig abgelegt sein, da im Voraus nicht bekannt ist, an welcher Speicheradresse die Bibliothek eingeblendet wird. Falls dynamische Bibliotheken wiederum dynamische Bibliotheken benötigen, werden Verweise auf sie angelegt.

Durch dynamische Bibliotheken lassen sich Programme besser warten. Bei einem Fehler in den Funktionen der Bibliothek ist nur diese Bibliothek auszutauschen. Programme, die die Bibliothek benutzen, müssen nicht geändert oder neu kompiliert zu werden. Außerdem sparen dynamische Bibliotheken Festplattenplatz, da der Entwickler den Funktionscode nicht statisch in seine Programme einkompilieren muss.

Der virtuelle Speichermechanismus ergibt einen weiteren Vorteil: Der in der Bibliothek enthaltene Code braucht das Betriebssystem nur einmal in den physikalischen Speicher zu laden und kann ihn dann in den virtuellen Adressraum anderer Prozesse einblenden.

Zum Vergleich: Das ursprüngliche A.out-Format kennt keine dynamischen Bibliotheken. Außerdem mussten die erzeugten Bibliotheken an eine feste Adresse gebunden sein. Um unter Linux trotzdem dynamische A.out-Libraries nutzen zu können, wurden in der Vor-ELF-Zeit einige Eigenschaften des Dateiformats missbräuchlich verändert.

Listing 1: Beispielprogramm »elf.c«

```
01 #include <stdio.h>
02 extern _start;
03
04 const int readOnly=42;
05 int data=42;
06 int bss=0;
07
08 int main() {
09    int stack=42;
10    printf("main= %p\n",&main);
11    printf("readOnly= %p\n",&readOnly);
12    printf("data= %p\n",&data);
13    printf("bss= %p\n",&bss);
14    printf("stack= %p\n",&stack);
15    printf("entry point= %p\n",&_start);
16    return 0;
17 }
```

Listing 2: ELF-Header

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement,
                                     little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80482a0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          7524 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         7
  Size of section headers:           40 (bytes)
  Number of section headers:         33
  Section header string table index: 30
```

Den Anfang bildet der ELF-Header

Jede ELF-Datei beginnt mit einem Header, der den Datei-Aufbau beschreibt. Seine Spezifikation findet sich in den Kernelquellen unter »include/linux/elf.h«. Um den Aufbau des Headers und der anderen Komponenten des ELF-Formats zu verstehen, muss hier das kleine C-Programm aus Listing 1 herhalten. »gcc elf.c« übersetzt es in eine ausführbare Datei mit dem Namen »a.out«, mit »gcc -c elf.c« entstünde eine verschiebbare Objektdatei. Der Dateiname »a.out« täuscht übrigens: Das Binary ist ein ELF-Format. Das Tool »readelf« ist sehr nützlich, um die Struktur von ELF-Dateien zu analysieren. Der Aufruf »readelf -h a.out« bringt den ELF-Header der Datei zum Vorschein (Listing 2).

Auf den ELF-Header folgt die Programm-Headertabelle. In der Regel fehlt sie bei relozierbaren Dateien. Danach kommen die Sektionen, sie nehmen die eigentlichen Daten auf, beispielsweise Programmdaten, Symbolnamen, Code und Informationen für die Relokation. Den Schluss bildet die Sektionen-Headertabelle, die die einzelnen Sektionen nach außen identifiziert.

Sind mehrere Objektmodule zu einem Programm verbunden, findet der Binder über die Sektionen-Headertabelle alle Informationen. Diese werden zum Beispiel benötigt, um die Sektionen, die den Programmcode enthalten, zu identifizieren und in die ausführbare Datei zu kopieren. Die ELF-Spezifikation spricht bei relozierbaren Dateien von der Linkansicht und bei dynamischen Bibliotheken und ausführbaren Dateien von der Ausführungsansicht (Abbildung 1).

Magische Zahl

Der ELF-Header beginnt mit 16 Identifikationsbytes. An den ersten vier Bytes (0x7F, »ELF« – die so genannte Magic Number) erkennen verarbeitende Programme, dass eine Datei im ELF-Format vorliegt. Die anderen Bytes geben unter anderem die verwendete Architektur vor, die Wortlänge und die Kodierung (Most Significant Byte (MSB) oder Least Significant Byte (LSB)). Da die Daten Byte-weise kodiert sind, interpretieren verarbeitende Programme sie auch ohne Kenntnisse über Wortlänge und Kodierung und entscheiden dann, ob sie überhaupt für die vorliegende Architektur verwendbar sind.

Der Typ »EXEC« kennzeichnet eine ausführbare Datei, hier für die x86-Architektur (»Machine: Intel80386«). Gültige Typen sind auch »REL« für relozierbare Dateien und »DYN« für dynamische Bibliotheken. Der Einstiegspunkt benennt die Adresse, bei der die Ausführung beginnen soll. Angegeben werden auch die Positionen der Programm- und der Sektionen-Headertabelle. Damit sind beide nicht an feste Positionen in der Datei gebunden. Damit den einzelnen Sektionen

ein Name zugeordnet werden kann, ist im ELF-Header noch ein Verweis auf die entsprechende Sektion enthalten (»Section header string table index«). In diesem Fall ist das die Sektion 30. Sie enthält eine Liste mit Ascii-Zeichen, die eine Null beschließt.

Die Sektionen-Headertabelle

Die Sektionen-Headertabelle ist eine Tabelle mit Einträgen fester Größe, die der »Size of section headers«-Eintrag im ELF-Header vorgibt. Die Einträge definieren die enthaltenen Sektionen, »readelf -S a.out« listet sie alle auf (Listing 3). Die einzelnen Namen der Sektionen löst ein Index in einer Stringtabelle auf, die der ELF-Header nennt (»Section header string table index«). Der Typ bezeichnet den Verwendungszweck der Sektion, wie ihn Tabelle 1 zeigt.

Der Typ einer Sektion bestimmt auch ihren internen Aufbau (siehe unten). Die Adresse »Addr« gibt die Position im virtuellen Adressraum an, an der die Sektion einzublenden ist. Der Offset »Off« legt die Anfangsposition innerhalb der Datei fest. Die Größe »Size« bezeichnet die Anzahl der Bytes in einer Sektion. Besitzen die Einträge in einer Sektion eine konstante Größe – beispielsweise bei einer Symboltabelle – gibt das »ES«-Attribut die Größe eines solchen Eintrags in Bytes an.

Alle Sektionen, bei denen das Flags-Attribut »Flg« auf »A« steht, lädt der Kernel in den Speicher. »X« kennzeichnet sie als ausführbar und »W« als schreibbar. Sind »X« oder »W« oder beide gesetzt, muss die Sektion auch als allozierbar (»A«) gekennzeichnet sein. Das Link-Attribut »Lk« verweist auf eine andere Sektion, deren Interpretation sich je Typ unterscheidet und die zusammen mit dem Info-Attribut »Inf« verwendet wird. Beispielsweise bekommt die Sektion vier (».dynsym«) auf diese Weise mitgeteilt, auf welche Stringtabelle (».dynstr«) sich die Einträge beziehen.

Die Vorschrift »Align« legt fest, wie die Daten ausgerichtet sein sollen. Obwohl Sektionen beliebige Namen tragen dürfen, erwartet Linux einige Standardsektionen, teilweise zwingend (Tabelle 2). Die Sektionen ».fini« und ».init« haben eine spezielle Bedeutung: Wenn eine Funktion in diesen Sektionen platziert wird, führt sie das Betriebssystem vor beziehungsweise nach dem eigentlichen Programm aus. Dieses Charakteristikum benutzen beispielsweise Compiler, um globale Konstruktoren und Destruktoren in C++ zu implementieren.

Im Gegensatz zum starren A.out-Format erlaubt das ELF-Format beliebige Sektionen. Gerade für die Abbildung globaler Konstruktoren und Destruktoren ist das wertvoll. A.out-Dateien dulden nur die vorab bekannten Sektionen: ».text«, ».data«, ».bss« sowie die Symbol- und die Stringtabelle.

Die Programm-Headertabelle

Ein Programm teilt sich in mehrere Segmente auf, die wiederum aus Sektionen bestehen. Der Kernel kümmert sich anhand der Programm-Headertabelle um

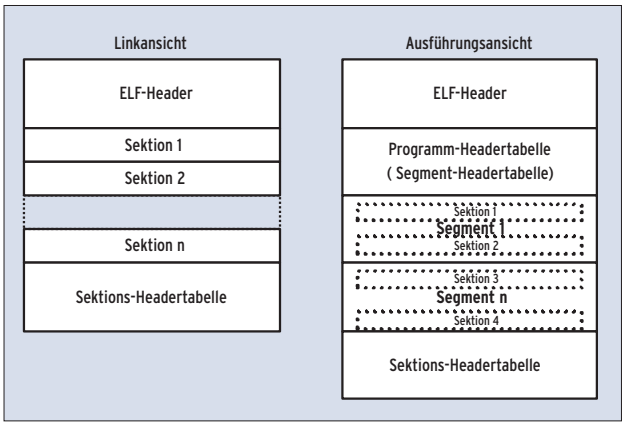

Abbildung 1: Aufbau von ELF-Dateien. Die ELF-Spezifikation spricht bei relozierbaren Dateien von der Linkansicht und bei dynamischen Bibliotheken sowie ausführbaren Dateien von der Ausführungsansicht.

Tabelle 1: Sektionstypen

Typ	Bedeutung
NULL	Nicht verwendete Sektion, deren Daten ignoriert werden
PROGBITS	Informationen, die zum Programm gehören und nur dort Bedeutung haben
DYNSYM	Symboltabelle für externe Referenzen
SYMTAB	Symboltabelle
HASH	Hashtabelle für das schnelle Finden von Einträgen in der Symboltabelle
DYNAMIC	Informationen für den dynamischen Binder, beispielsweise benötigte Bibliotheken
STRTAB	Stringtabelle, abschließende Nullen trennen die Zeichen

Listing 3: Enthaltene Sektionen

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048148 000148 000028 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          08048170 000170 000050 10   A  5   1  4
  [ 5] .dynstr           STRTAB          080481c0 0001c0 00004c 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          0804820c 00020c 00000a 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         08048218 000218 000020 00   A  5   1  4
  [ 8] .rel.dyn          REL             08048238 000238 000008 08   A  4   0  4
  [ 9] .rel.plt          REL             08048240 000240 000010 08   A  4   b  4
  [10] .init             PROGBITS        08048250 000250 000017 00  AX  0   0  4
  [11] .plt              PROGBITS        08048268 000268 000030 04  AX  0   0  4
  [12] .text             PROGBITS        080482a0 0002a0 000250 00  AX  0   0 16
  [13] .fini             PROGBITS        080484f0 0004f0 00001b 00  AX  0   0  4
  [14] .rodata           PROGBITS        0804850c 00050c 000053 00   A  0   0  4
  [15] .data             PROGBITS        08049560 000560 000010 00  WA  0   0  4
  [16] .eh_frame         PROGBITS        08049570 000570 000004 00   A  0   0  4
  [17] .dynamic          DYNAMIC         08049574 000574 0000c8 08  WA  5   0  4
  [18] .ctors            PROGBITS        0804963c 00063c 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049644 000644 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        0804964c 00064c 000004 00  WA  0   0  4
  [21] .got              PROGBITS        08049650 000650 000018 04  WA  0   0  4
  [22] .bss              NOBITS          08049668 000668 000008 00  WA  0   0  4
  [23] .comment          PROGBITS        00000000 000668 0000bd 00      0   0  1
  ...
  [30] .shstrtab         STRTAB          00000000 001c43 00011e 00      0   0  1
  [31] .symtab           SYMTAB          00000000 00228c 0006e0 10     32  52  4
  [32] .strtab           STRTAB          00000000 00296c 0003a2 00      0   0  1
```

das Einladen der Segmente. »readelf -l a.out« gibt die Segmente der ausführbaren Datei aus (Listing 4).

Die ausführbare Datei »a.out« umfasst sechs Segmente, aus denen sich das Programm im Speicher zusammensetzt. Für jedes Segment sind Typ, Größe, Position im virtuellen Adressraum, Flags und Ausrichtungsvorschrift (Alignment) angegeben. Die Angabe der physikalischen Adresse wird unter Linux ignoriert. Die in dem Beispielprogramm definierten Typen haben die in Tabelle 3 beschriebene Bedeutung.

Der Typ »LOAD« ist für den Kernel das wichtigste Segment. Er erkennt daran, was er wohin und mit welchen Zugriffsrechten (»Flg«) einblenden muss. Es fällt auf, dass der Speicher (»MemSize«) des zweiten Segments »LOAD« etwas größer ist als die Größe innerhalb der Datei (»FileSize«). Grund: Das Segment enthält auch die ».bss«-Sektion – also den mit einer Null initialisierten Heap. Er ist nicht in der Datei abgebildet, was viel Plattenplatz spart. Die zwei Segmente vom Typ »LOAD« müssen sich an 0x1000 ausrichten, was der Seitengröße entspricht. Außerdem ist unter »Section to Segment mapping« die Gruppierung der Sektionen zu den Segmenten zu sehen. Abbildung 2 zeigt die Zusammenhänge.

Dynamisches Laden

Das Laden und Starten einer ausführbaren Datei erfolgt in zwei Schritten. Ein Aufruf des »execve()«-Systemrufs veranlasst den Linux-Kernel, den aktuellen Prozess mit den Daten aus der Datei zu ersetzen. Angeleitet durch die Programm-Headertabelle bindet er die nötigen Segmente der Datei – sie enthält die einzelnen Sektionen – per Memory-Mapped-Files-Mechanismus in den virtuellen Adressraum des Prozesses ein. Das bedeutet, dass der Inhalt eines Segments erst beim ersten Zugriff in den physikalischen Speicher geladen wird.

Dieser Mechanismus bietet – neben der Performance – zwei Vorteile: Erstens muss der Kernel für nur lesbare Segmente (beispielsweise das Code-Segment, erkennbar am Ausführungsflag des ersten »LOAD«-Segments) keinen Platz auf dem Swap-Device reservieren, da es jederzeit wieder aus der Datei geladen werden kann. Zweitens kann er ein in den physikalischen Speicher geladenes Segment in den virtuellen Adressraum vieler Prozesse einblenden, was den Hauptspeicher schont.

Ist der Prozess so vorbereitet, geht die Kontrolle nicht etwa an den im ELF-Header angegebenen Einstiegspunkt über, sondern an den im »INTERP«-Segment angegebenen dynamischen Binder. Alle für den Binder nötigen Informationen legt der Kernel auf dem Stack des Prozesses ab. Der Befehl »LD_SHOW_AUXV = 1 ./a.out« gibt dann die Werte aus.

Dynamische Bibliotheken (DSOs) sind in der Regel positionsunabhängig (PIC, Position Independent Code), sie sind an beliebiger Stelle in den Speicher ladbar, ohne den Code durch Relokationen an die Adresse anpassen zu müssen. Darum sind die Adressen der Funktionen des DSO zur Kompilierungszeit nicht bekannt, Linux muss sie zur Laufzeit erst finden.

Die PLT liegt im schreibgeschützten Speicher

Dazu leitet es die Sprungziele im Programm über die Procedure Linkage Table (PLT) aus der Sektion ».plt« ab. Da die PLT im schreibgeschützten Speicher liegt, sind die Sprungziele nicht direkt modifizierbar. Hier kommt die Global Offset Table (GOT, abgebildet in der ».got«-Sektion) ins Spiel. Sie enthält alle änderbaren absoluten Adressen und wird in den beschreibbaren Speicherbereich eingeblendet. Die PLT benutzt die GOT also als Indirektionsstufe. In ausführbaren Programmen ist sie an eine feste Adresse gebunden, die über das globale Symbol »_GLOBAL_OFFSET_

Listing 4: Enthaltene Segmente

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0055f 0x0055f R E 0x1000
  LOAD           0x000560 0x08049560 0x08049560 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000574 0x08049574 0x08049574 0x000c8 0x000c8 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
          .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
   04     .dynamic
```

Tabelle 2: Bedeutung der Sektionen

Name	Beschreibung
.fini	Enthält Instruktionen, die ausgeführt werden, wenn der Prozess terminiert
.init	Enthält Instruktionen, die vor dem eigentlichen Hauptprogramm ausgeführt werden
.data	Enthält initialisierte Daten; die Variable »data« im Beispielprogramm liegt hier
.rodata	Enthält nur lesbare, konstante Daten; die Variable »readOnly« aus dem Beispiel ist hier platziert
.text	Hier ist der Code abgelegt, den der Prozessor ausführen wird
.hash	Eine Hashtabelle für das schnelle Finden von Symbolen
.dynsym	Symboltabelle für den dynamischen Binder
.dynstr	Ascii-kodierte und mit Null (»\0«) abgeschlossene Zeichenketten, die der dynamische Binder benötigt
.symtab	Die Symboltabelle ist hier gespeichert; sie entspricht dem Format aus Abbildung 4
.shstrtab	Enthält die Namen der Sektionen selbst, etwa ».text«, ».strtab« und so weiter; die Sektion referenziert der ELF-Header unter »Section header string table index«
.dynamic	Enthält eine Tabelle mit Informationen für den dynamischen Binder, beispielsweise die Namen von zu ladenden Bibliotheken oder die Position der verwendeten Stringtabelle
.strtab	Enthält die Namen der Symbole als fortlaufende Null-terminierte Ascii-Zeichenketten
.rel.dyn, .rel.plt	Relokations-Einträge: Eine Tabelle mit einer Beschreibung, welche Positionen zu ändern sind, wenn die Datei im Speicher verschoben wird
.interp	Enthält den als Ascii-Zeichenkette kodierten Namen des Interpreters

TABLE_« referenzierbar ist. Bei DSOs nimmt ein Register (EBX bei Intel) die Adresse der GOT auf.

Ein Aufruf von »objdump -j .got a.out -d« zeigt den Inhalt der GOT an (Tabelle 4). Eintrag »0« enthält die Adresse der ».dynamic«-Sektion, die man über das Symbol »_DYNAMIC« referenzieren kann. Das ist für den dynamischen Binder wichtig, weil er sich initialisieren muss, ohne vorher Relokationen durchgeführt zu haben – dazu müsste er sich selbst verwenden. Der dynamische Binder setzt die Einträge »1« und »2«, bevor das Programm startet. Position »5« wird anhand eines Relokationseintrags aufgelöst. Die Positionen »3« und »4« sind hier etwas ungewöhnlich, da sie schon Adressen enthalten.

Die PLT besteht aus einer Tabelle zu je 16 Bytes mit kleinen Codefragmenten. Sie dient als eine Art Sprungtabelle, um externe Funktionen (aus DSOs) aufzurufen. Ein Aufruf von »objdump -j .plt -d a.out« erzeugt den in Listing 5 angegebenen Inhalt.

Jede extern benutzte Funktion bekommt einen Eintrag

Der erste Eintrag »(0)« springt in den dynamischen Binder. Die Operanden der »push«-Anweisungen sind die Offsets in die Relokationstabelle. In ihr lagern das entsprechende Symbol und die Adresse des zu verändernden Werts (Adresse des GOT-Eintrags). »R_386_JMP_SLOT« ist der Typ des Eintrags. Jede extern benutzte Funktion enthält einen Eintrag in der GOT und der PLT. Der erste Aufruf von »printf« (»(2)«) führt indirekt über den vierten Eintrag in der GOT und damit zur »push«-Anweisung (»(2.1)«).

Die letzte Anweisung »(2.2)« macht einen Sprung zum ersten PLT-Eintrag »(0)«. Der dynamische Binder sucht die Adresse der Funktion und trägt sie an der vierten Position in der GOT ein. Der nächsten Aufruf springt somit direkt die »printf«-Funktion an. Dieser Mechanismus ermöglicht es, PIC-Code zu erzeugen, der trotzdem mit festen Adressen arbeitet. Weil die Einträge nicht direkt in der PLT geändert werden, lässt sich das Codesegment in viele Prozesse einblen-

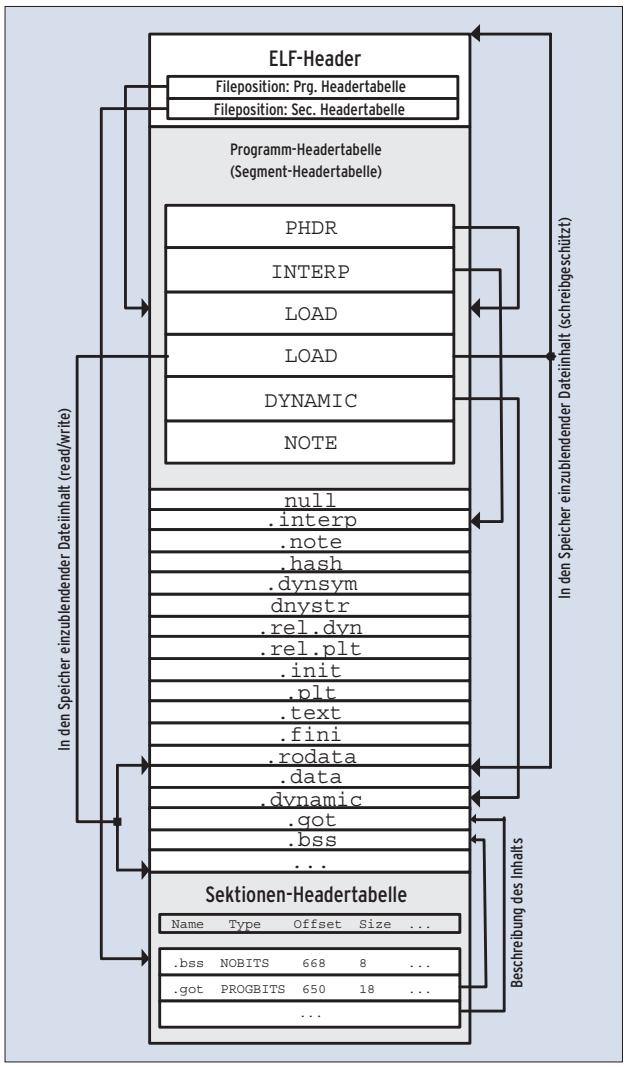

Abbildung 2: Am Typ »LOAD« erkennt der Kernel, was er wohin und mit welchen Zugriffsrechten (»Flg«) einblenden muss.

Tabelle 3: Segmenttypen

Typ	Bedeutung
PHDR	Die Position der Programm-Headertabelle (ihre eigene)
INTERP	Beschreibt den Namen eines externen Programms, das nach dem Kernel die Kontrolle erhält
LOAD	Segmente mit diesem Typ werden vom Kernel an die angegebenen Adressen eingeblendet
DYNAMIC	Enthalten Informationen für den dynamischen Binder

Tabelle 4: Global Offset Table

Adresse	Eintrag	Inhalt	Beschreibung
8049650	0	0x08049574	».dynamic«
8049654	1	0x00000000	Identifikations-Information
8049658	2	0x00000000	Einstiegspunkt in den dynamischen Binder
804965c	3	0x0804827e	»_libc_start_main« (»PLT[1]«)
8049660	4	0x0804828e	»printf« (»PLT[2]«)
8049664	5	0x00000000	»__gnom_start__«

Listing 5: Procedure Linkage Table

```
08048268 <.plt>:
(0)    8048268:       ff 35 54 96 04 08       pushl  0x8049654   ; GOT[1]
       804826e:       ff 25 58 96 04 08       jmp    *0x8049658  ; GOT[2]
       8048274:       00 00                   add    %al,(%eax)  ; Füllbytes
       8048276:       00 00                   add    %al,(%eax)  ; Füllbytes
(1)    8048278:       ff 25 5c 96 04 08       jmp    *0x804965c  ; GOT[3]
       804827e:       68 00 00 00 00          push   $0x0        ; ID _start_main
       8048283:       e9 e0 ff ff ff          jmp    8048268     ; PLT[0]
(2)    8048288:       ff 25 60 96 04 08       jmp    *0x8049660  ; GOT[4]
(2.1) 804828e:       68 08 00 00 00          push   $0x8        ; ID printf
(2.2) 8048293:       e9 d0 ff ff ff          jmp    8048268     ; PLT[0]
```

Listing 6: Sektionen nach Prelink

```
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 4] .dynsym           DYNSYM          08048170 000170 000050 10   A 23   1  4
[ 5] .gnu.liblist      GNU_LIBLIST     080481c0 0001c0 000028 14   A 23   0  4
[23] .dynstr           STRTAB          08049670 000670 00005f 00   A  0   0  1
[32] .gnu.prelink_undo PROGBITS        00000000 001cf4 000614 01      0   0  4
[24] .gnu.conflict     RELA            080496d0 0006d0 000048 0c   A  4   0  4
```

den, ohne es mehrfach im physikalischen Speicher zu halten.
Das Prinzip der Relokation kann man vereinfachend als Verschiebung bezeichnen. Hierbei findet die Verknüpfung von Symbolverweisen mit Symboldefinitionen statt, also die Zuordnung zu Werten. Jedes Objektmodul (Obj-Datei) erzeugt der Compiler so, als besäße es seinen eigenen Adressraum. Darum haben relozierbare Dateien so genannte Relocation Entries (Abbildung 3), durch die beim Zusammenfügen ein eigener Adressraum entsteht.

Relokationseinträge

Die Sektionen »[8]« und »[9]« im Beispiel (Tabelle 3) sind Relokationseinträge, die am Typ »REL« in der Sektionen-Headertabelle zu erkennen sind. Sie sind notwendig, weil sie ausführbare Dateien und DSOs darüber informieren, wie der Inhalt einer Sektion zu ändern ist, damit sie Programme verbinden können. Ein Relokationseintrag zeigt genau auf die zu ändernde Adresse (Offset). Die Symboltabelle hat auch einen Index, auf den der Offset anzuwenden ist.
Die Symboltabelle ordnet Bezeichner und Adressen einander zu, Abbildung 4 zeigt ihren Aufbau. In der Sektionen-Headertabelle ist sie als Typ »SYMTAB« gekennzeichnet. Das Link-Attribut »Lk« zeigt auf die korrespondierende Stringtabelle vom Typ »STRINGTAB«. Der Name steht in der Stringtabelle, in der Headertabelle nur der Offset. »Wert« ist eine Adresse, die man relativ zu einer Sektion oder absolut angeben darf.
»Info« beschreibt das Binding des Symbols, also seine Sichtbarkeit. Sie kann lokal, global oder schwach (weak) sein. Lokale Symbole sind außerhalb ihrer Objektdatei nicht sichtbar. Global definierte Symbole sind für alle Objektdateien sichtbar. Da Symbole immer eindeutig sein müssen, ist bei globalen Symbolen also besondere Vorsicht geboten. »Info« gibt zusätzlich die Klassifikation des Symbols an, etwa Objekt (Variablen, Arrays et cetera), Funktion, Sektion oder Datei. Symbole sind immer in Verbindung zu einer Sektion definiert, wofür der Index in der Sektionen-Headertabelle (Adresse der Sektion + Wert = Adresse) verantwortlich zeichnet.

Listing 7: Messung Xterm ohne Prelink

```
frank@debian:~/elf$ time xterm
real    0m0.013s
user    0m0.010s
sys     0m0.000s

frank@debian:~/elf$ LD_DEBUG=statistics xterm
      1380:                    number of relocations: 744
      1380:          number of relocations from cache: 1454
      1380:
      1380:    runtime linker statistics:
      1380:               final number of relocations: 853
      1380:    final number of relocations from cache: 1454
```

Listing 8: Messung Xterm mit Prelink

```
frank@debian:~/elf$ time xterm
real    0m0.004s
user    0m0.000s
sys     0m0.000s

frank@debian:~/elf$ LD_DEBUG=statistics xterm
      1426:                    number of relocations: 0
      1426:          number of relocations from cache: 1367
      1426:
      1426:    runtime linker statistics:
      1426:               final number of relocations: 0
      1426:    final number of relocations from cache: 1367
```

Aufwändiges Lazy Binding

Fast alle Programme benutzen DSOs, die zur Laufzeit geladen und deren Symbolreferenzen aufgelöst werden müssen. Der dynamische Binder benutzt ein so genanntes Lazy Binding. Das heißt, er führt nicht alle Relokationen auf ein Mal durch, sondern nur dann, wenn ein Symbol tatsächlich verwendet wird. Das Setzen der Umgebungsvariable »LD_BIND_NOW = 1« deaktiviert das Lazy Binding bei Bedarf, sodass alle Relokationen schon beim Laden des Programms erfolgen. Bei Programmen mit sehr vielen dynamischen Bibliotheken dauert das Auflösen sehr lange – und zwar bei jedem Programmstart.
Das Prelink-Tool [2] verspricht Abhilfe. Es erfordert eine Glibc 2.3.1-r2 oder neuer. Prelink sammelt alle Bibliotheken, die ein Programm benötigt, und weist ihnen einen eindeutigen virtuellen Adressraum zu, um sie gegen die Symbole zu binden. Die ausführbare Datei kann jetzt mit den festen Adressen in ihrer GOT bestückt werden. »prelink -vm ./a.out« startet den Vorgang, »readelf -e a.out« in Listing 6 zeigt die Änderungen im Vergleich zu Listing 3.

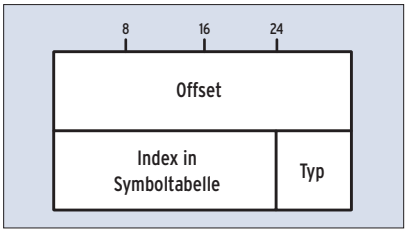

Abbildung 3: Relokationseintrag (Relocation Entry) ohne Zusatz für einen eigenen Adressraum.

Neue Sektionen machen sich breit

Im Wesentlichen gibt es drei neue Sektionen. Dazu wurden die Sektionen-Headertabelle sowie der ELF-Header verändert. Die Sektion ».gnu.liblist« enthält eine Liste abhängiger Bibliotheken und ihrer Namen (als Index in die per »Linkflag(23)« angegebene Sektion), Checksummen und Zeitstempel. ».gnu.prelink_undo« enthält Informationen zum Rückgängigmachen der Änderungen. Die Sektion ».gnu.conflict« listet Konflikte, etwa Symbole mit gleichen Namen, die sich an Stellen mit unterschiedlichen Sichtbarkeiten befinden.
Hier wird auch die Glibc geändert und im Prinzip an eine feste Adresse gebunden. Auch die exportierten Symbole erhalten absolute Adressen. »prelink« patcht das Programm dann, indem es die referenzierten Symbole sucht und mit ihren festen Adressen im Programm einträgt. Die Relokation ist vorgezogen und muss nicht mehr zur Laufzeit geschehen. »prelink« benutzt den dynami-

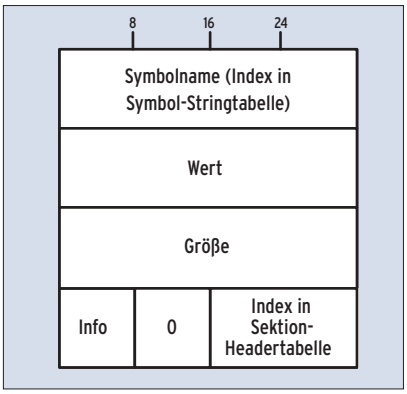

Abbildung 4: Die Symboltabelle ordnet Bezeichner und Adressen einander zu.

schen Binder zum Suchen und Auflösen der Symbole. Wegen der Umgebungsvariablen »LD_TRACE_PRELINKING = 1« gibt der Binder alle verwendeten Symbole aus. Prelink liest sie durch eine Pipe ein und verarbeitet sie. Ändern sich in der Zwischenzeit die Bibliotheken, ist der dynamische Binder klug genug, das anhand der ».gnu.liblist« zu erkennen – er schaltet dann auf sein Standardverhalten zurück und lädt die nötigen Bibliotheken erst zur Laufzeit.

Den größten Performancegewinn durch Prelinking erfahren dynamisch gelinkte C++-GUI-Programme, weil sie meist die Dienste sehr vieler Bibliotheken in Anspruch nehmen und ihre Symbolnamen sehr lang werden können. Das Vorbinden hat noch einen weiteren Vorteil: Linux arbeitet nach dem Copy-on-Write-Mechanismus (COW). Das ist eine wichtige Technik zum Kopieren jener Speicherseiten, die von mehreren Prozessen benutzt werden. Im lesenden Zugriff teilen die Prozesse sich die Seite. Erst wenn einer von ihnen versucht den Seiteninhalt, etwa eine Variable, zu ändern, erzeugt das Betriebssystem für diesen Prozess eine neue Speicherseite.

Durch das Relozieren zur Laufzeit, also ohne Prelink, können Prozesse solche geänderten Speicherseiten nicht mehr miteinander teilen. Das Vorbinden macht diese Relokationen aber unnötig, was weniger physikalischen Hauptspeicher verbraucht. Zudem wird der Kernel entlastet, da er weniger Seiten im Speicher kopieren muss. Die Listings 7 und 8 skizzieren den Unterschied anhand des »xterm«-Programms: Das Starten eines mit Prelink veränderten Programms geht tendenziell schneller.

ELF bringt Fortschritt und Tuning-Chancen

Das ELF-Format ist viel variabler als das zu Recht ausgemusterte A.out-Format, aber komplex ist es schon. Die Struktur eines ELF-Binaries ist nicht unergründlich, doch das Schreiben eines dynamischen Binders erfordert Expertenwissen. Um die Startgeschwindigkeit von Anwendungen – vor allem von grafischen Oberflächen wie KDE – zu erhöhen bietet sich das Tool »prelink« an, das alle neueren Distributionen mitbringen. Besonders dynamisch gelinkte GUI-Programme profitieren davon. *(jk)* ■

Infos

[1] ELF-Spezifikation: [http://www.linuxbase.org/spec/refspecs/elf/elf.pdf]

[2] Prelink: [http://packages.debian.org/unstable/admin/prelink.html]

Der Autor

Frank Peters arbeitet als Software-Consultant bei der Firma Intralab. Arbeitsschwerpunkte sind das Architekturdesign von komplexen verteilten Systemen und die Performance-Optimierung von J2EE-Anwendungen. Sein Interesse gilt der Systemprogrammierung und dem Verständnis der Funktionsweise von Betriebssystemen.