Ohne Linker kein Programm

Linker sind für die Entwicklung nativer Anwendungen ein unverzichtbares Werkzeug. Sie fügen nicht nur einzelne Programmmodule zu einer ausführbaren Binärdatei zusammen, sondern können Speicherplatz und Zeit sparen.

Von Dr. Christoph Erhardt

Wäre die Werkzeugkette der Softwareentwicklung eine Rockband, dann wäre der Compiler ihr Frontmann: berühmt, berüchtigt und in aller Munde. Dem Linker fiele eher die Rolle des Bassisten zu: unauffällig bis unsichtbar und nur einschlägig Interessierten namentlich bekannt. Das geht so weit, dass manche sich fragen, ob der Linker mehr als ein arkanes Relikt aus grauer Vorzeit ist und ob außerhalb des althergebrachten C-/C++-Ökosystems überhaupt ein Linker zum Einsatz kommt. Die Antwort ist ein klares Ja. Egal, ob man Code in C++, Go, Rust oder Haskell schreibt: Wann immer am Ende des Bauvorgangs eine nativ ausführbare Binärdatei herauspurzelt, wurde sie durch einen Linker erzeugt.

Der erste Teil der Artikelreihe beleuchtet die technischen Grundlagen des Linkens und beschreibt, was unter der Haube passiert. Der technische Fokus liegt auf der Verarbeitung von ELF-Programmen (Executable and Linking Format) unter Linux; die Konzepte sind aber genauso auf andere gängige Binärformate wie PE (Windows) oder Mach-O (macOS, iOS) anwendbar. Der zweite Teil in einer der nächsten iX-Ausgaben wird sich mit der Praxis befassen und zeigen, welche Features ein moderner Linker beherrscht und welche Linker es im Open-Source-Umfeld gibt. Wer auf den richtigen Linker setzt, kann damit zeitkritische Teile des Bauprozesses um ein Vielfaches beschleunigen.

Inhalt der Artikelreihe

Teil 1: Technische Grundlagen: Aufgaben eines Linkers, Relokation, Symbole, Funktionen, Bibliotheken

Teil 2: Linker-Auswahl unter Linux, Leistung, erweiterte Features: inkrementelles Linken, Skripte, Identical COMDAT Folding, Linkzeitoptimierung

Linker im Detail: ein Blick unter die Haube

Für alle gängigen Programmiersprachen ist der Compiler nach wie vor so ausgelegt, dass er jedes Quellmodul separat verarbeitet. So erzeugt ein C-Compiler für jede .c-Quelldatei eine zugehörige .o-Objektdatei, die den Maschinencode für dieses Modul enthält. Dem Linker fällt nun die Aufgabe zu, diese Ansammlung von Einzelmodulen zu einer ausführbaren Binärdatei oder ladbaren Bibliothek zusammenzufügen (Abbildung 1). Die klare Arbeitsteilung trägt zur Wiederverwendbarkeit bei; so kann ein und derselbe Linker mit den verschiedensten Compilern zusammenarbeiten.

In aller Regel wird der Linker nicht von Hand oder durch das Build-System aufgerufen. Stattdessen findet ein Aufruf an das Compiler-Frontend statt, das dann den Linker beauftragt. So kann das Frontend nach oben eine Menge sprach- und plattformspezifischer Details verbergen, die es dem Linker per Befehlszeile mitteilen muss. Dazu zählen explizite Angaben zu Zielarchitektur und Typ der Ausgabedatei sowie unverzichtbare Abhängigkeiten wie Startup-Code und Standardbibliothek(en) der verwendeten Programmiersprache. Im Fall der GNU Compiler Collection ergibt sich die folgende Aufrufkette: `gcc` (Frontend) – `collect2` (Linker-Wrapper) – `ld` (der eigentliche Linker). Das Kürzel `ld` ist ein Relikt aus der Unix-Urzeit, als Linker und Lader noch ein und dasselbe Stück Software waren.

Symbole und Sektionen, die Grundbausteine des Linkens

Für den Einstieg ist es wichtig zu verstehen, in welcher Form der Linker seine Eingabedaten vorfindet. Jede globale respektive statische Variable und jede kompilierte Funktion ist nicht mehr als ein Array von Bytes mit fixer Größe und vorinitialisiertem Inhalt, entweder ein Bündel von Daten oder eine Sequenz von Maschineninstruktionen. Den Namen, über den ein solches Programmelement eindeutig identifiziert und adressiert wird, bezeichnet man als Symbol. Der Inhalt hinter jedem Symbol ist genau einer Sektion zugeordnet: einem zusammenhängenden Speicherbereich mit einheitlichen Attributen und Zugriffsrechten. So

Listing: Beispielprogramm hitchhiker mit Symboltabellen

```
// Inhalt von main.c
#include <stdio.h>
#include <stdlib.h>
#include "deepthought.h"

int main(void) {
    printf("The answer is: %u\n", compute_answer());
    return EXIT_SUCCESS;
}
```

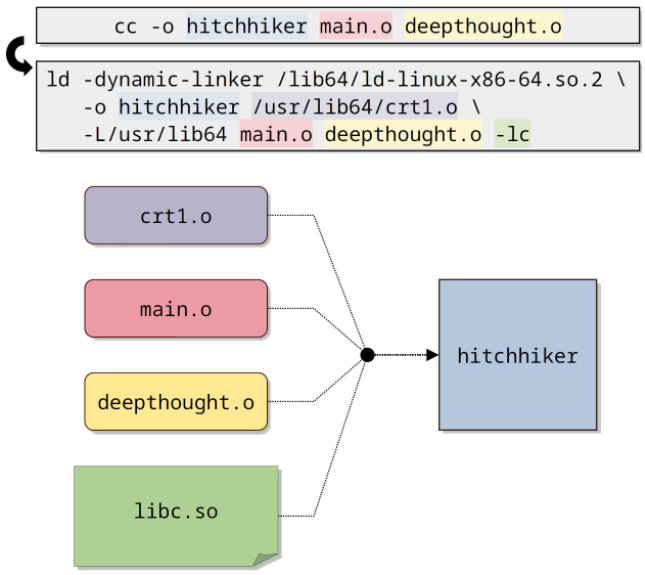
```
// Symboltabelle von main.o
                 U compute_answer
                 U printf
00000000 00000018 T main
```

```
// Inhalt von deepthought.c
#include "deepthought.h"
#include <unistd.h>

unsigned the_answer = 42;

unsigned compute_answer(void) {
    usleep(7500000);
    return the_answer;
}
```

```
// Symboltabelle von deepthought.o
                 U usleep
00000000 00000013 T compute_answer
00000000 00000004 D the_answer
```



Der Linker fügt die Arbeitsprodukte des Compilers – hier drei Objektdateien und eine dynamische Bibliothek – zusammen und erzeugt als Endprodukt eine ausführbare Binärdatei (Abb. 1).

bündelt die les- und ausführbare `.text`-Sektion den Programmcode von Funktionen, `.rodata` enthält Konstanten und ist nur lesbar; `.data` enthält les- und schreibbare globale Variablen.

In jeder erzeugten .o-Datei legt der Compiler neben den genannten Sektionen eine weitere Sektion namens `.symtab` an, die eine Symboltabelle enthält. Diese Tabelle ist ein Verzeichnis aller in diesem Modul definierten Symbole samt Verweis auf die zugeordnete Sektion und einer (vorläufigen) Adresse. Zusätzlich enthält sie Verweise auf alle externen Symbole, die aus diesem Modul heraus referenziert werden.

Das Listing enthält den Quellcode und die daraus generierten Symboltabellen des Beispielprogramms hitchhiker. Die Ausgabe im Format (Adresse, Größe, Sektion, Name) entstammt dem Unix-Kommando `nm -nS`, die Abkürzungen bedeuten: U = unaufgelöst, T = `.text`, D = `.data`.

Die erste Aufgabe des Linkers ist es, die Symboltabellen aller Einzelmodule und Bibliotheken in eine große Tabelle zusammenzuführen. Bleiben am Ende unaufgelöste Symbolreferenzen übrig, ist das Programm unvollständig. Die Folge ist die berüchtigte Fehlermeldung „undefined reference to foo“. Umgekehrt gilt: Definieren mehrere Module ein und dasselbe Symbol, dann weiß der Linker nicht, welches Exemplar er nehmen soll, und bricht mit der Meldung „multiple definition of foo“ ab.

Hat das Zusammenführen aller Symboltabellen konfliktfrei funktioniert, kann der Linker jedem Symbol eine Adresse zuweisen und die zugehörigen Inhalte – also Code und Daten – in ihre Ausgabesektionen kopieren. Mit einer naiven Eins-zu-eins-Kopie ist es hier allerdings nicht getan, weil in einem solchen Bytestrom häufig Verweise auf Symbole eingebettet sind: Der Programmcode enthält Funktionsaufrufe und Zugriffe auf globale Variablen und die globalen Variablen wiederum können mit Zeigern auf andere Variablen oder Funktionen vorinitialisiert sein. Der Linker muss alle symbolischen Verweise während des Kopierens durch konkrete Adressen ersetzen (Abbildung 2). Dieser Vorgang nennt sich Relozieren.

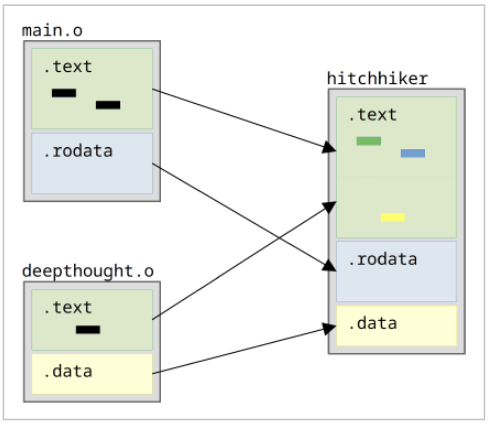
Ein Beispiel aus einer konkreten Objektdatei ist in Abbildung 3 zu sehen: Hier hat der Compiler vorläufige Platzhalterwerte in Form von Nullen in den Maschinencode eingefügt. Da die Adressierung stets relativ zum Programmzähler `%rip` erfolgt, beziehen sie sich scheinbar auf die jeweils nachfolgende Instruktion. Denn der Programmzähler `%rip` zeigt auf x86_64 immer auf die nächste Instruktion: Während `call` an Offset 0x6 ausgeführt wird, zeigt `%rip` bereits auf 0xb, und beim Ausführen von `mov` an Offset 0xb zeigt `%rip` auf 0x11.

Relokationen sind die Ausfüllhilfe für den Linker

Da ein Linker keinen Disassembler enthält, hat er keine Ahnung von Inhalt und interner Struktur der Sektionen, die er kopieren soll, und auch nicht, wie die darin enthaltenen Maschineninstruktionen zu interpretieren wären. Aus seiner Sicht sind alle Sektionen nichts weiter als semantikfreie Arrays von Bytes. Es stellt sich die Frage, woher der Linker dennoch

iX-TRACT

- Linker verbinden Programmmodule zu einem ausführbaren Programm oder zu einer Bibliothek.
- Vor allem dynamische Bibliotheken stellen Linker vor anspruchsvolle Aufgaben.
- Die verwendeten Beispiele demonstrieren die Verarbeitung von ELF-Programmen (Executable and Linking Format) unter Linux, die grundlegenden Konzepte gelten aber genauso für andere Binärformate auf anderen Plattformen.



Beim Zusammenkopieren der Sektionsinhalte muss der Linker Adressplatzhalter (schwarz) gegen echte Adressen (bunt) austauschen (Abb. 2).

weiß, an welchen Stellen in der Ausgabedatei er welche Adresswerte einfügen muss und in welchem Format.

Hier kommt die Relokationstabelle ins Spiel, die in jeder .o-Datei enthalten ist. In dieser Tabelle hat der Compiler für jeden Platzhalter eine Ausfüllvorschrift (Relokation) hinterlegt. In modernen ELF-Objektdateien wird sie üblicherweise im RELA-Format erzeugt (Relocation with explicit Addend) und in der Sektion `.rela.text` gespeichert. Eine RELA-Relokation ist ein Vierertupel aus einem Relokationstyp `T`, einem Symbol `S`, einem Offset `0` und einem Summanden `A`. Damit wird der folgende Code ausgedrückt:

$$C[0] := f_T(addr(S) + A)$$

Auf Deutsch bedeutet das: Ermittle die Adresse von `S`, addiere `A`, codiere die Summe in dem durch `T` vorgegebenen Format und schreibe das Ergebnis an Offset `0` in den Ausgabebytestrom.

Der Zoo verfügbarer Relokationstypen `T` ist spezifisch für die Zielarchitektur. Maßgeblich ist, welche möglichen Codierungen von Immediate-Werten in Maschineninstruktionen die Befehlssatzarchitektur vorsieht. Auf RISC-Architekturen wie ARM, wo Instruktionen eine fixe Länge haben, ist beispielsweise das Holen einer längeren Adresse häufig auf zwei Instruktionen aufzuteilen; beide erhalten dann jeweils eine eigene Relokation. Auf x86_64 mit seinen variablen Instruktionslängen passt dagegen selbst eine absolute 64-Bit-Adresse in eine Instruktion.

Für das Beispielmodul deepthought.o hat der Compiler die Relokationstabelle in Abbildung 4 erzeugt (ausgelesen mit `readelf -r deepthought.o`). Die Tabelle enthält zwei Tupel (`T`, `S`, `0`, `A`) mit den Werten `(PLT32, usleep, 0x7, -4)` und `(PC32, the_answer, 0xd, -4)`. Der Typ `PLT32` markiert den Aufruf einer Bibliotheksfunktion über die spezielle PLT-Indirektion (mehr dazu später); `PC32` bezeichnet eine gewöhnliche programmzählerrelative Adressierung. Der negative Summand –4 in beiden Fällen rührt daher, dass der Programmzähler auf x86 immer auf den Beginn der nächsten Instruktion zeigt. Diesen Versatz gilt es bei der Adressberechnung zu kompensieren.

Ein Blick in die Symboltabelle hilft beim Ausrechnen

Mit diesen Informationen ist der Linker nun in der Lage, die Werte für die Platzhalter auszurechnen, indem er in der Symboltabelle nachschlägt. Im Beispiel liegt der PLT-Einsprungpunkt der externen Bibliotheksfunktion `usleep()` an der Adresse 0x401020 und die globale Variable `the_answer` an 0x40402c. Abbildung 5 zeigt den fertig relozierten Maschinencode der Funktion `compute_answer()` in der ausführbaren Binärdatei hitchhiker.

Der Linker kann beim Relozieren auch tiefgreifendere Änderungen vornehmen als das bloße Befüllen von Adressplatzhaltern. In bestimmten Fällen kann es passieren, dass er eine ganze Instruktion in eine andere umschreibt. Ein Beispiel sind Funktionsaufrufe in der ARM-Architektur: Aufrufe, die den Befehlssatz wechseln – von ARM zu Thumb oder zurück –, werden durch eine andere Instruktion (`blx`) repräsentiert als gewöhnliche Funktionsaufrufe (`bl`). Da der Compiler für ein modulexternes Aufrufziel noch nicht wissen kann, ob es ARM- oder Thumb-Code enthält, emittiert er zunächst eine `bl`-Instruktion; erst der Linker hat dieses Wissen und muss sie dann gegebenenfalls in `blx` umschreiben.

Jetzt hat der Linker alle Informationen beisammen, die er zum Erstellen der fertigen ausführbaren Datei benötigt: Er hat die Symboltabellen zusammengeführt und ihre Konsistenz sichergestellt sowie die finalen Inhalte für Code und Daten errechnet, indem er die Relokationen anwendete. Die Inhalte schreibt er nach Sektionen gebündelt in die Ausgabedatei und klebt einen ELF- sowie einen Programmheader davor. Beim Ausführen des Programms wird der Lader für jede Sektion ein zugehöriges Segment im RAM anlegen und die jeweiligen Inhalte dorthin laden.

Bibliotheken: Puzzlestücke von Softwaresystemen

Um Code wiederverwendbar zu machen oder Softwareprojekte zu strukturieren, ist es ein gängiger Ansatz, logisch zusammengehörende Softwaremodule zu Bibliotheken zusammenzufassen. Dafür gibt es zwei Mechanismen, die sich

```
0000000000000000 <compute_answer>:
      0:  50                    push  %rax
      1:  bf e0 70 72 00        mov   $0x7270e0,%edi
      6:  e8 00 00 00 00        call  b <compute_answer+0xb>
      b:  8b 05 00 00 00 00     mov   0x0(%rip),%eax    # 11 <compute_answer+0x11>
     11:  5a                    pop   %rdx
     12:  c3                    ret
```

In der für x86_64 kompilierten Objektdatei deepthought.o sind die Adressen von usleep() (grün) und the_answer (blau) noch nicht aufgelöst (Abb. 3).

```
Relocation section '.rela.text' at offset 0x1a8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000400000004 R_X86_64_PLT32    0000000000000000 usleep - 4
00000000000d  000500000002 R_X86_64_PC32     0000000000000000 the_answer - 4
```

Die Relokationstabelle in deepthought.o teilt dem Linker mit, wo die Platzhalter aus Abbildung 3 liegen und wie und womit sie auszufüllen sind (Abb. 4).

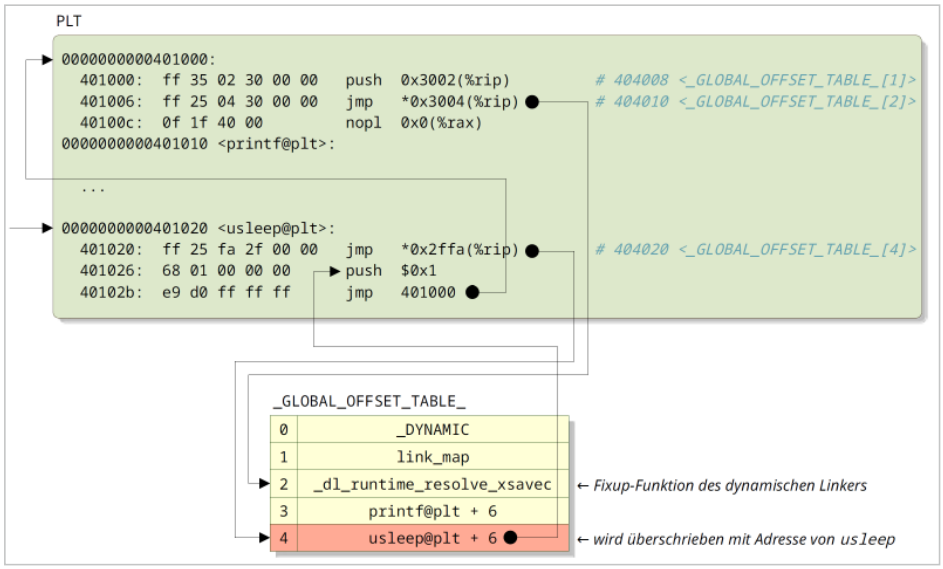
```
0000000000401085 <compute_answer>:
 401085:  50                    push  %rax
 401086:  bf e0 70 72 00        mov   $0x401090 - 0x70
 40108b:  e8 90 ff ff ff        call  401020 <usleep@plt>
 401090:  8b 05 96 2f 00 00     mov   0x2f96(%rip),%eax    # 40402c <the_answer>
 401096:  5a                    pop   %rdx
 401097:  c3                    ret
```

(= -0x70)

(= 0x401096 + 0x2f96)

In der Ausgabedatei hat der Linker die Leerstellen mit programmzählerrelativen Adressangaben (in Little-Endian-Zweierkomplementnotation) ausgefüllt (Abb. 5).

Beim ersten Aufruf von usleep@plt() lenkt die globale Offsettabelle den Kontrollfluss in den dynamischen Linker; er überschreibt den GOT-Eintrag mit der Adresse der Zielfunktion, sodass jeder nachfolgende Aufruf von usleep@plt() dorthin weiterspringt (Abb. 6).



grundlegend voneinander unterscheiden: statische und dynamische Bibliotheken. Beide benötigen spezielle Unterstützung, aber vor allem dynamische Bibliotheken stellen den Linker vor zusätzliche Herausforderungen.

Eine statische Bibliothek ist nichts weiter als eine simpel strukturierte unkomprimierte Archivdatei, in der eine oder mehrere .o-Dateien zusammengepackt sind; sie gehorcht üblicherweise dem Namensschema lib<name>.a. Statische Bibliotheken lassen sich dem Linker auf der Befehlszeile per `-l<name>` als Symbolquellen mitgeben. Dabei gelten aber andere Regeln als für gewöhnliche Objektdateien: Anstatt alle .o-Dateien aus der Bibliothek einzubinden, greift sich der Linker nur diejenigen Module heraus, die Definitionen für bis dato unaufgelöste Symbole enthalten. Für die Praxis bedeutet das, dass die Reihenfolge der Befehlszeilenargumente wichtig ist: Bibliotheken sollten nach allen .o-Dateien ganz am Ende stehen.

Wenn eine statische Bibliothek libA.a von einer anderen statischen Bibliothek libB.a abhängt, müssen Entwickler darauf achten, beide in der richtigen Reihenfolge einzubinden: erst `-lA`, dann `-lB`. Besonders vertrackt wird es bei zyklischen Abhängigkeiten. Hier kann es nötig sein, eine Bibliothek mehrfach anzugeben, also `-lA -lB -lA`. Genau wie Objektdateien sind statische Bibliotheken

Zwischenprodukte des Bauvorgangs, die man nach dem Linken entsorgen kann. Zum Ausführen des fertigen Programms sind sie überflüssig.

Wenn mehrere Programme ein und dieselbe statische Bibliothek einbinden, enthält jedes von ihnen eine Kopie des Bibliothekscodes. Das schlägt sich nicht nur auf belegten Festplattenplatz und Arbeitsspeicherbedarf nieder und bläht sich im Instruktionscache unnötig auf, sondern hat in der Praxis auch ganz konkrete Sicherheitsimplikationen: Wird in einer weithin verwendeten Bibliothek ein kritischer Bug entdeckt und gefixt, muss jedes einzelne Anwendungsprogramm, das diese Bibliothek einbindet, gegen die fehlerbereinigte Version neu gelinkt werden.

Abhilfe gegen diese Nachteile versprechen dynamische Bibliotheken. Eine dynamische Bibliothek, auch Shared Library oder Shared Object genannt, ist eine ladbare Funktionssammlung, die in Form einer lib<name>.so-Datei auf der Platte liegt. Der Maschinencode einer solchen Bibliothek ist nur einmal in den Arbeitsspeicher zu laden, danach können beliebig viele Prozesse ihn an einer jeweils frei wählbaren Basisadresse im eigenen Adressraum einblenden. Diese Deduplikation spart RAM.

Damit das funktioniert, muss der Quellcode der Bibliothek als positionsunabhängiger Code (Position-independent Code; Compileroption `-fPIC`) übersetzt worden sein. Im Maschinencode müssen dann sämtliche Adressbezüge relativ zum Programmzähler sein. Das ist großteils trivial, da heutige Befehlssatzarchitekturen ohnehin standardmäßig mit programmzählerrelativen Sprüngen und Funktionsaufrufen arbeiten. Es bleiben aber Fälle übrig, in denen Programmcode mit einer absoluten Symboladresse hantieren muss: wenn er einen Zeiger auf eine globale Variable oder eine Funktion holt, eine Bibliotheksfunktion aufruft und wenn er auf eine globale Variable einer Bibliothek zugreift.

Damit der Maschinencode selbst positionsunabhängig bleiben kann, muss er den Umgang mit solchen absoluten Adressen über eine Indirektion realisieren. Dafür braucht es eine globale Offsettabelle (GOT). Sie liegt an einem fixen Offset relativ zur `.text`-Sektion und enthält für jedes relevante Symbol einen Eintrag mit dessen absoluter Adresse. Für die genannten Fälle generiert der Compiler Maschinencode, der die GOT zum Nachschlagen nutzt.

Das Befüllen der GOT übernimmt der dynamische Lader (ld.so) zu dem Zeitpunkt, wenn er die Bibliothek lädt. Die Information, in welche GOT-Einträge er welche Adressen zu schreiben hat, bekommt er in Form dynamischer Relokationen mitgeteilt (eine pro GOT-Eintrag), die der Linker erzeugt und in die Sektion `.rela.dyn` der Bibliothek schreibt. Damit delegiert der Linker Aufgaben, die er statisch zum Linkzeitpunkt noch nicht erledigen kann, an den Lader. Der Lader wird so zu einem dynamischen Linker.

Effizientes Arbeiten mit Lazy Binding

Bei umfangreichen Frameworks wie Qt und bei der C-Standardbibliothek kann die GOT groß werden, was sich in der Ladezeit bemerkbar macht. Oft benutzt ein Programm aber nur einen Bruchteil der gebotenen Funktion. Sinnvoll wäre, nur diejenigen dynamischen Relokationen aufzulösen, die tatsächlich verwendet werden. Deshalb bieten heutige Systeme einen partiell trägen Ansatz (Lazy Binding): Der Lader schreibt den GOT-Eintrag einer Funktion nicht beim Laden der Bibliothek, sondern erst bei ihrem allerersten Aufruf. Dafür generiert der Linker eine zweite Tabelle, die Procedure Linkage Table (PLT), die für jede Bibliotheksfunktion ein Stück synthetisierten Trampolincode enthält. Je nach Initialisierungsstatus des zugehörigen GOT-Slots leitet das Trampolin den Kontrollfluss entweder zuerst in den dynamischen Linker oder direkt in die Zielfunktion. Der Assembly-Code in der PLT ist so geschickt gestrickt, dass der Ausführungspfad zwar verschlungen ist, aber ohne Verzweigungen und bedingte Sprünge auskommt.

Wie das beim Aufruf von `usleep@plt()` aussieht, skizziert Abbildung 6: Im Ausgangszustand verweist jeder Funktionseintrag in der GOT zunächst auf die zweite Instruktion des jeweils zugehörigen PLT-Slots. Der indirekte `jmp` am Anfang des PLT-Codes führt also unmittelbar zurück in den eigenen Code, der aus der GOT die Fix-up-Funktion des dynamischen Linkers heraussucht und anspringt. Die Fix-up-Funktion ermittelt dann die virtuelle Adresse der Zielfunktion `usleep()`, überschreibt den GOT-Eintrag an Index 4 mit dieser Adresse und springt dorthin. Ab diesem Zeitpunkt wird jeder nachfolgende Aufruf an `usleep@plt()` nahtlos nach `usleep()` weiterspringen.

Dank dieser Indirektion muss der Lader nicht jede Aufrufstelle einer Bibliotheksfunktion einzeln relozieren, sondern es braucht pro aufgerufener Funktion nur eine Relokation: die des zugehörigen GOT-Eintrags. Die zusätzlichen Tabellenzugriffe machen positionsunabhängigen Code gegenüber positionsabhängigem Code zwar minimal langsamer, in der Praxis fällt dieser Unterschied aber kaum ins Gewicht.

Leider ist der Lazy-Binding-Ansatz ein zweischneidiges Schwert, weil er eine sicherheitsrelevante Problematik aufwirft: Da die GOT zur Programmlaufzeit schreibbar bleibt und noch dazu an einer fixen Adresse liegt, ist es relativ einfach, GOT-Einträge gezielt zu überschreiben. Das ist ein beliebtes Mittel für Angreifer, einen einfachen Pufferüberlauf in kontrollierte Codeausführung umzumünzen. Moderne Linux-Distributionen härten deswegen ihre Softwarepakete, indem sie standardmäßig die Linker-Option `-z now` setzen. Sie zwingt den dynamischen Linker, alle Relokationen schon zum Ladezeitpunkt aufzulösen. In Kombination mit der Option `-z relro` führt das dazu, dass er unmittelbar danach die GOT (und weitere Datenstrukturen) nur lesbar macht. Damit ist dieser Angriffsvektor aus dem Spiel genommen.

Fazit

Der Artikel hat im Schnelldurchgang die Funktionen eines Linkers beleuchtet: Module zusammenfügen, Sektionen verschmelzen, Symbolbezüge auflösen und dynamische Programmbibliotheken zur Laufzeit ladbar machen. Ein moderner Linker beherrscht aber noch viel mehr. Der zweite Teil der Artikelreihe wird erweiterte Features vorstellen und erkunden, welche Linker in einer zeitgenössischen Linux-Distribution zur Verfügung stehen und worin sie sich unterscheiden. So viel sei bereits verraten: Gerade in Sachen Performanz gibt es große Unterschiede. (nb@ix.de)

Quellen

Beispiele, Linker-Blogartikelsammlung: ix.de/zksa

DR. CHRISTOPH ERHARDT

entwickelt als Senior Software Engineer bei Method Park by UL Softwareplattformen für Medizingeräte. Er steuert Patches zum mold-Linker bei und betreut das mold-Paket als Maintainer für Fedora Linux.