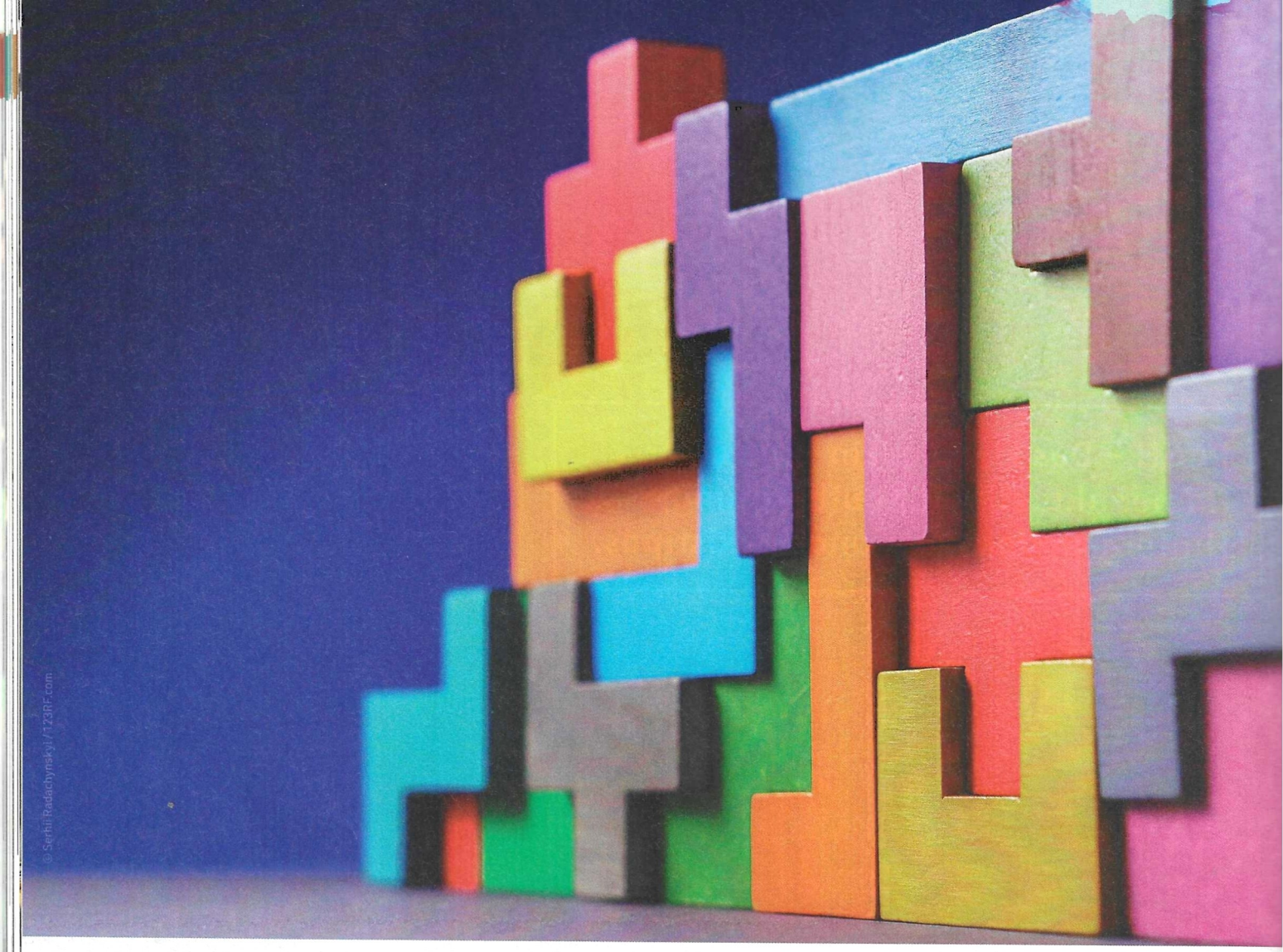


Kernel- und Treiberprogrammierung mit Linux – Folge 130

Speicherpuzzle

Die 64-Bit-Architektur verspricht Arbeitsspeicher ohne Ende, bürdet damit aber dem Kernel eine komplexe Verwaltung auf. Ein Blick in das Proc-Verzeichnis sorgt für mehr Klarheit.

Eva-Katharina Kunst, Jürgen Quade

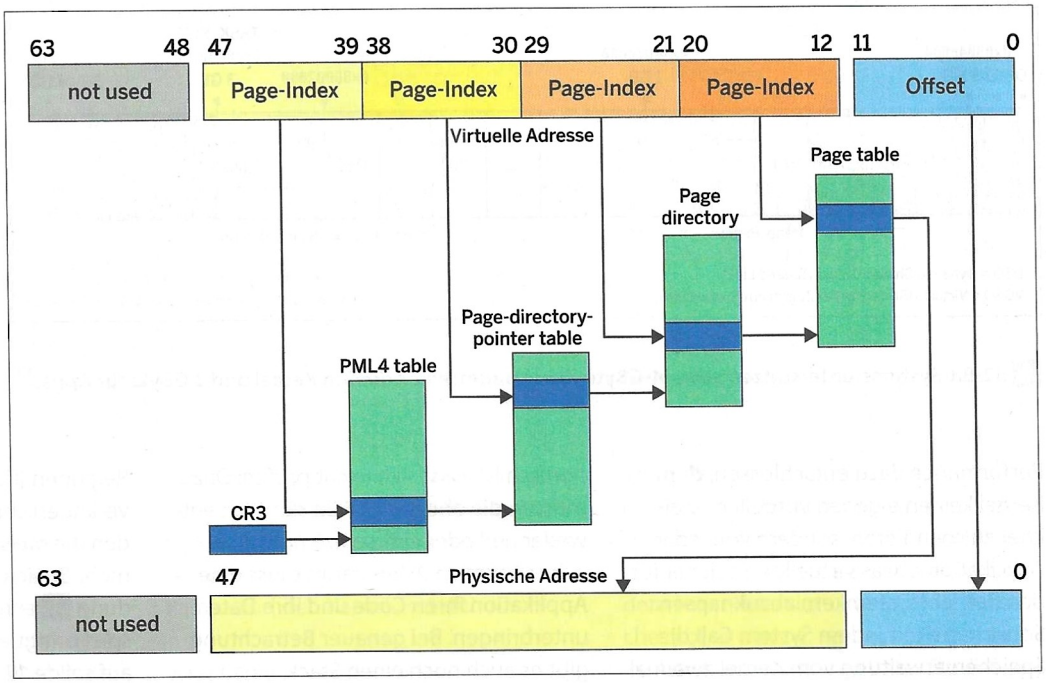
Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. **Jürgen Quade**, Professor an der Hochschule Niederrhein, gibt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux.

64-Bit-Systeme versprechen rechnerisch einen Adressraum von 16 Exabyte, also 16 Millionen Terabyte. Das ist viel, sehr viel. Aber vor allem bleibt es bei der Theorie. Tatsächlich hat AMD als Entwicklungsstätte der x86_64-Architektur den Adressraum auf 48 Bit reduziert, also auf 256 TByte. Gute 15 Millionen TByte bleiben damit ungenutzt. Aber bei aller Enttäuschung: Das genügt derzeit locker selbst für komplexeste Anwendungen. Wer damit nicht auskommt, dem bleibt noch die Intel-Variante mit einem Adressraum von 57 Bit (128 Petabyte). Allerdings ist die aktuelle Limitierung des Adressraums ohnehin selbst gemacht und keineswegs immanent. Sie kann also in Zukunft gelockert werden.

Zur Klarstellung: Der Adressraum gibt die Menge an Speicher an, die eine CPU grundsätzlich ansprechen kann. Dessen Umfang bestimmt die Breite der CPU-Register beziehungsweise hardwaretechnisch der sogenannte Adressbus, also die Anzahl der Adressleitungen zwischen CPU und physischem Speicher. Bei einem 64-Bit-System sind die Register 64 Bit breit, und der Adressbus könnte theoretisch dieselbe Breite aufweisen. Das

ergibt schließlich die theoretische Speichermenge von 2^{64} Bit, sprich 16 Exabyte. Faktisch gibt es maximal 48 Adressleitungen für 256 TByte physisches Memory.



1 Komplex, aber effektiv: das Memory Management per Four-Level-Paging.

Klotzen

Den physischen Speicher müssen sich alle Rechenprozesse teilen. Das bedingt sofort und berechtigterweise Konflikte, denen der Betriebssystemkern geschickt aus dem Weg geht. Dazu gaukelt er jedem einzelnen Prozess vor, er hätte mehr oder minder den kompletten Speicher für sich allein zur Verfügung, und zwar nicht nur den physisch tatsächlich verbauten, sondern die maximalen 256 TByte.

Dieses Täuschungsmanöver erleichtert die Programmerstellung kolossal. Da sich in der Praxis die einzelnen Jobs ohnehin mit signifikant weniger Memory begnügen, kommt es nur selten zu ernst zu nehmenden Konflikten. Wird in Summe mehr Speicher benötigt, als physisch vorhanden ist, lagert der Kernel intelligent Speicherinhalte auf die SSD oder Festplatte in den Swap-Speicher aus. Gibt es auch hier keinen Platz mehr, rückt der OOM-Killer aus und erledigt hinterlistig ein paar Speicherfresser.

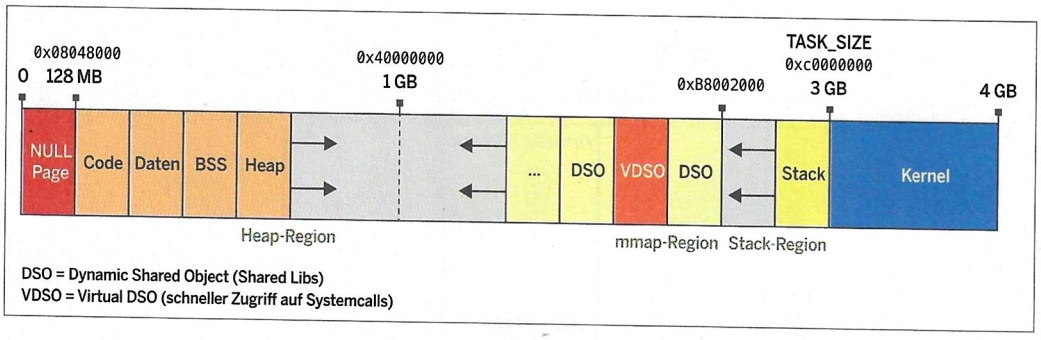
Den kompletten Adressraum und damit den Speicher, der einer Applikation zur Verfügung steht, nennt man virtuellen Speicher. Für das Aufteilen des physischen Speichers zwischen den Applikationen, dem Betriebssystemkern und dem Swapping zeichnet im Kernel die Speicherverwaltung verantwortlich. Sie nimmt über Tabellen und ein vier- oder – in der Intel-Variante – fünfstufiges Paging die Umsetzung von virtuellen auf physische Adressen vor (siehe Kasten Four-Level-Paging).

Halbe-halbe

Doch bleiben wir zunächst beim virtuellen Speicher. Bereits bei der 32-Bit-Architektur hatte man sich aus Gründen der

Four-Level-Paging

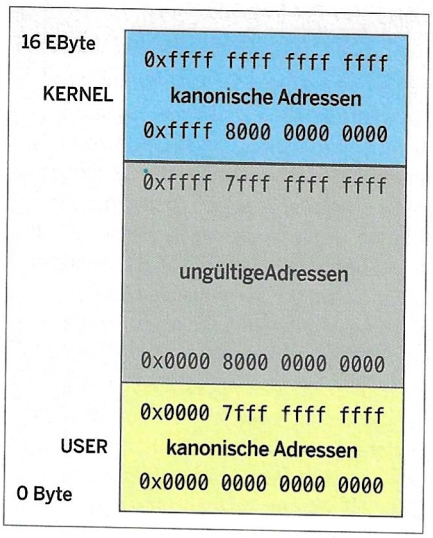
Ungeachtet dessen, was das Betriebssystem den Applikationen vorspiegelt, muss es den aktuellen Speicherbedarf letztlich auf den limitierten, physischen Speicher abbilden. Zu diesem Zweck teilt Linux den Speicher in 4-KByte-Blöcke ein, die sogenannten Pages. Es bildet dementsprechend einen 4-KByte-Block einer Applikation auf einen 4-KByte-Block physischen Speicher ab, indem es die zur Auswahl des Blocks notwendigen Adressbits austauscht.
Im einfachsten Fall könnte das für jede Applikation eine Tabelle übernehmen, in der die physischen Adressen abgelegt sind. Die oberen 36 Bit der virtuellen Adresse (der Page-Auswahl) wählen den Eintrag in der Tabelle aus, der Inhalt besteht dann aus 36 Bit der physischen Adressen. Das würde eine Tabelle mit 2^{36} Elementen zu je 36 Bit erfordern – also Gigabytes für jede einzelne Applikation. Das ist definitiv zu viel.
Deshalb verwendet Linux ein mehrstufiges Umsetzungsverfahren, bei dem man sich ähnlich wie bei einem Baum von Ast zu Ast bis zur eigentlichen Information hangelt. Abbildung 1 zeigt das Prinzip: Die 36 Blockauswahl-Bits der virtuellen Adresse teilt man in vier 9-Bit-Blöcke ein, von denen jeder eine aus 512 Speicherzellen mit je 64 Bit auswählt. 512 mal 64 Bit (also 8 Byte) ergeben nicht ganz zufällig 4 KByte, also eine Page. Die 9 Bits wählen also jeweils eine Speicherzelle innerhalb einer vom Kernel konfigurierten Page aus. In dieser Speicherzelle befindet sich unter anderem die Adresse einer nächsten Page mit 512 Speicherzellen. Dort wählen die nächsten 9 Bits wiederum eine Speicherzelle aus. Bei Stufe vier schließlich findet sich endlich die physische Blockadresse. In Kombination mit den 12 übrigen Bits – dem Offset – ergibt sie die physische Adresse. Da die Blockadresse von den 64 Bit einer Speicherzelle lediglich 36 Bit benötigen, lassen sich in den verbleibenden Bits zusätzliche Informationen ablegen, beispielsweise Zugriffsrechte.
Für eine superkleine Applikation benötigt man so nur noch vier Tabellen, also 16 KByte Speicher – eine saftige Reduktion. Den Anfang in diesem Umsetzungsbaum macht auf einer x86_64-Architektur übrigens das für jeden Job spezifische CPU-Register *CR3*, das das Betriebssystem bei jedem Prozesswechsel (Scheduling) neu setzt.



2 32-Bit-Systeme unterstützen einen 4-GByte-Adressraum: eines für den Kernel und 3 GByte für Apps.

Performance dazu entschlossen, dem Kernel keinen eigenen virtuellen Speicher zu spendieren, sondern von jeder Applikation etwas virtuellen Speicher für den Betriebssystemkern abzuknapsen. Sonst hätte mit jedem System Call die Speicherverwaltung vom Kernel zweimal umgebaut werden müssen. Von 4 GByte virtuellem Speicher auf einem 32-Bit-System verbleiben im Normalfall für die Applikation 3 GByte; das letzte GByte bleibt für den Kernel reserviert 2.

Auf der 64-Bit-Maschine hat AMD die 256 TByte mittig geteilt. Der Speicher für den Kernel liegt oben im virtuellen Adressraum, der für die Applikation unten. Dazwischen klafft eine riesige Lücke ungenutzter Adressen. Das Unternehmen hat auch eine neue Bezeichnung eingeführt, die kanonische Adresse („canonical address" 3). Kanonisch steht hier für legal oder nutzbar. Ob eine Adresse kanonisch ist, lässt sich leicht prüfen: Dazu müssen die obersten 17 Bit sämtlich entweder null oder eins sein.



3 64-Bit-Systeme nutzen nur einen kleinen Teil des Adressraums.

Im virtuellen Adressraum muss eine Applikation ihren Code und ihre Daten unterbringen. Bei genauer Betrachtung gibt es auch noch einen Stack, einen Heap (für per `malloc()` oder `new` dynamisch reservierten Speicher) und Shared Libs. Zudem gilt es aus Gründen der Kompatibilität, im Adressraum Schnittstellen zur Kommunikation mit dem Kernel unterzubringen. Langer Rede kurzer Sinn: Der Adressraum der Applikation und auch des Kernels muss organisiert werden. Damit lassen sich dann auch die Sicherheitsfeatures einer Speicherverwaltung nutzen. So kann man hardwareseitig das Verändern statischer Daten oder das Ausführen von Code vom Stack aus verhindern. Die Speicherbereiche der Applikation bezeichnet man als Regionen oder Segmente.

Fein säuberlich

Abbildung 4 zeigt die Struktur, wie Linux sie in einem 32-Bit-System realisiert. Auf ein 128 MByte großes Null-Segment folgt der Code der Applikation. Das Null-Segment lässt bewusst Programme abstürzen, die einen nicht initialisierten Adresszeiger (Pointer) verwenden. Auf den Code folgen die Daten, statisch, initialisiert und uninitialisiert. Der darauf anschließende Speicher lässt sich je nach Bedarf verwenden, also zum Beispiel als Heap. Auf der anderen Seite, an den hohen Adressen des virtuellen Adressraums, liegen die Bibliotheken und der Stack. Auf einem 64-Bit-System ergibt sich abgesehen vom erheblich größeren Adressraum eine ähnliche Struktur. Wie Abbildung 5 verdeutlicht, bleiben dabei Teile des Adressraums ungenutzt.

Da etwa Bibliotheken auch aus verschiedenen Regionen (Code, unveränderliche Daten, veränderliche Daten) bestehen, verwenden die meisten Applikationen weitaus mehr Speicherregionen als in der Abbildung 4 erkennbar. LibreOffice zum Beispiel bringt es auf einem 64-Bit-System auf solide 124 Segmente. Das können Sie leicht nachprüfen, da Linux weitreichende Informationen über Rechenprozesse im Verzeichnis `/proc/<pid>/` zur Verfügung stellt. Zu jeder Applikation lassen sich dort eine Reihe unterschiedlicher Dateien abrufen. Die Datei `maps` liefert dabei viele Informationen zu den Segmenten.

Trickreich lernen

Um ein erstes Gefühl für diese Informationen zu bekommen, geben Sie auf einer Konsole das Kommando `cat /proc/self/maps` ein. Das `self` im Pfadnamen zeigt als Alternative zur Angabe der PID als symbolischer Link auf das eigene Verzeichnis. Das Programm Cat gibt den Inhalt von Dateien auf dem Bildschirm aus, hier also – etwas von hinten durch die Brust geschossen – den Inhalt seiner eigenen Speicherorganisation.

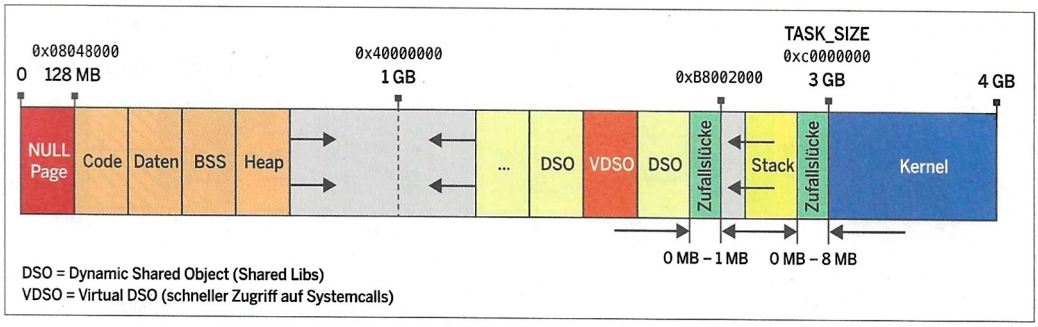
Etwas übersichtlicher wird es, sobald man das Auslesen von `/proc/self/maps` in ein kleines C-Programm gießt, das man ohne Bibliotheken als besonders schlanke Software generiert. Abbildung 6 zeigt das Übersetzen und den Aufruf des Programms.

Jede Zeile der Ausgabe steht für eine Region. Die erste Spalte gibt Start und Ende der virtuellen Adresse der Region

Listing 1: ASLR deaktivieren

```
# echo 0 >/proc/sys/kernel/randomize_va_space
```

an. Danach folgen vier Zugriffsrechte: r steht für Lesen, w für Schreiben, und x symbolisiert ein Codesegment. Das p bedeutet privat, also den exklusiven Zugriff nur für die Applikation selbst. Bei Segmenten, auf die mehrere Jobs zugreifen, ersetzt ein s (shared) das p.



4 In einer 32-Bit-Architektur folgt auf ein 12 MByte großes Null-Segment der Code der Applikation.

In dem Beispiel aus Abbildung 6 startet das Codesegment, zu erkennen am x bei den Zugriffsrechten des Programms, ab der Adresse `0x0000000000401000`. Steht an der Stelle ein Minus (–) anstelle des x, handelt es sich um ein Datensegment. Nach den Zugriffsrechten folgen die Angabe eines Offsets innerhalb des Segments, der zugehörige Gerätetreiber mit seiner Major- und Minor-Nummer und schließlich der Inode, also die Nummer, unter der man die Datei auf dem Speichermedium (SSD, Festplatte) findet.

An der Liste lässt sich sehr gut ablesen, dass Code, konstante Daten und veränderliche Daten am Anfang des Adressbereichs liegen (ab `0x401000`). Das Segment davor enthält statische, vom Dateiformat ELF abgelegte Informationen. Der Heap logiert ebenfalls im unteren Bereich. Der Stack und die mit `[vvar]` und `[vdso]` bezeichneten Segmente liegen an den oberen Adressen des virtuellen Applikationsadressraums, also knapp unterhalb von 128 TByte. Zur Erinnerung: AMD hat ja den virtuellen Adressraum auf 256 TByte reduziert und je zur Hälfte auf die Applikation und den Kernel verteilt.

Altlasten

Die Segmente `[vvar]` und `[vdso]` ermöglichen eine pfiffige Technik zum effizienten und kompatiblen Aufrufen von System Calls. Das Segment `[vsyscall]` dient ebenfalls dem Aufruf von Syscalls, ist aber faktisch ein Relikt vergangener Zeiten, das wohl nur aus Gründen der Kompatibilität in den Adressraum eingeblendet wird. Wie Sie an der Adresse erkennen können, zählt es eigentlich zum virtuellen Adressraum des Kernels.

Der von Applikationen dynamisch angeforderte Speicher (Heap) liegt in Abgrenzung zum Stack am Ende der eigentlichen Programmsegmente. Allerdings hat sich auch hier inzwischen eine Differenzierung eingeschlichen. Tatsächlich werden nur kleinere Datenbereiche auf dem Heap reserviert. Für größere Bereiche gibt es eigene, anonyme Regionen im Adressraum unterhalb des Stack-Segments. In unserem Beispiel findet sich kein solches anonymes Segment.

Der Zufall hilft

Beim Mehrfachstart eines Programms und gleichzeitiger Analyse der Speicherregionen fällt zudem auf, dass identische Segmente immer an unterschiedlichen Adressen liegen.

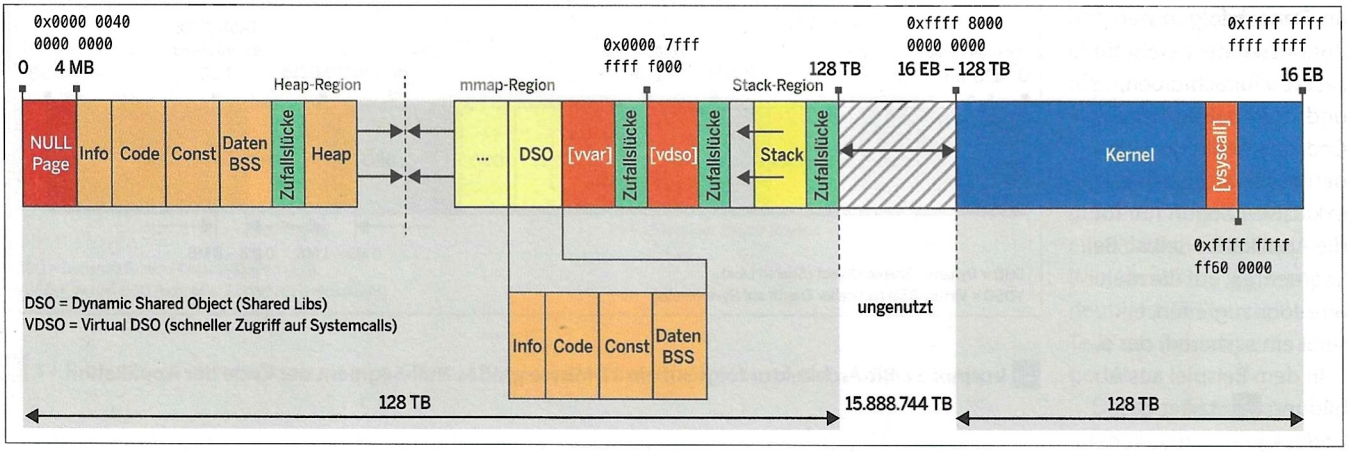
Hier greift eine Technik mit dem Namen Address Space Layout Randomization (ASLR), die die genaue Adresslage nach dem Zufallsprinzip bestimmt. Damit machen es moderne Betriebs-

Listing 2: `maps.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
//#include <fcntl.h>
#define HEAP_SIZE (256*1024*256)

int global_var = 99;
char *const_data = "program start";

int main( int argc, char **argv, char **envp )
{
  int fd, count;
  char buffer[256];
  char *heap_var;
  heap_var = (char *)sbrk(HEAP_SIZE); // alloc heap memory
  if (heap_var == NULL) {
    perror("sbrk");
    exit( -2 );
  }
  heap_var = heap_var - HEAP_SIZE;
  printf("code: %p, const_data: %p, var_data: %p, heap_data: %p ", main, const_data, &global_var, heap_var);
  printf("stack_data: %p\n\n", &fd);
  fd = open("/proc/self/maps", O_RDONLY);
  if (fd<0) {
      printf("open failed\n");
      exit( -1 );
  }
  while ((count=read( fd, buffer, sizeof(buffer)))>0) {
    if (write(1, buffer, count)<0) break;
  }
  close(fd);
  printf("\n");
  return 0;
}
```



5 Im virtuellen Adressraum einer 64-Bit-Architektur liegen manche Teile vollkommen brach.

systeme Angreifern schwer, die genauen Adresslagen vorherzusagen und sie dann für ihre bösartigen Zwecke zu nutzen. Je mehr Variabilität in den Adressen steckt, desto schwieriger wird eine Attacke.

Linux gibt sich übrigens in diesem Kontext von den gängigen Betriebssystemen am spendabelsten und weist im Gegensatz zu den Konkurrenten auch keine Anomalien auf. Sie können ASLR als Superuser auch über das Kommando aus Listing 1 ausschalten.

Speicherabfrage

Um auf der eigenen Maschine vergleichbare Ergebnisse zu produzieren, generieren Sie das C-Programm `maps` (Listing 2) mithilfe des in Listing 3 gezeigten Kommandos aus dem Quellcode. Allerdings gilt es, zuvor unter `/usr/src/linux/` den Linux-Quellcode mit den Header-Dateien der Nolibc abzulegen. Gegebenenfalls passen Sie stattdessen den Pfad zu den Nolibc-Headern im Kommando an.

Um das Programm normal mit der C-Standardbibliothek zu übersetzen, müssen Sie in Zeile 4 noch die Header-Datei `fcntl.h` einkommentieren. Außerdem sollten Sie im Hinterkopf behalten, dass der eingesetzte Syscall `sbrk()`, der Platz auf dem Heap schafft, sich bei Verwendung der C-Standardbibliothek anders verhält und den Start des neuen Bereichs zurückgibt, nicht dessen Ende.

Infos abgreifen

Im Übrigen erschöpft sich das Mitteilungsbedürfnis des Kernels bezüglich der Rechenprozesse nicht mit den gezeigten Informationen. Im Verzeichnis `/proc/PID/` findet sich beispielsweise die Datei `smaps`. Sie hält zu jedem Segment weitere, detaillierte Informationen bereit, die insbesondere bei der Verwaltung des Speichers anfallen. Hier können Sie zum Beispiel recherchieren, wie viel physischen Speicher (`Rss`) jedes einzelne Segment nutzt und welche der über 30 Zugriffs-Flags (`VmFlags`) im Detail gesetzt sind. Die Bedeutung dieser und auch anderer Informationen aus dem Verzeichnis `proc/` erläutert die Linux-Kernel-Dokumentation erfreulich ausführlich.

Kenntnisse über die Speicher- und Prozessverwaltung sind nicht nur für Nerds oder Betriebssystementhusiasten interessant. Bei der Entwicklung und der Administration lassen sich darüber Bottlenecks in Applikationen und im System aufspüren, und mit Kenntnis von Prozessen und deren Adresslagen extrahieren Forensiker bei einer Live-Analyse relevante Daten. Über weitere, manipulative Möglichkeiten wollen wir hier lieber nicht fachsimpeln. (*jlu*)

```
quade@ezs-cocka:~/linux-magazin$ cc -isystem /usr/src/linux/tools/include/nolibc/ -nostdlib -nolibc -no-pi
e     -fno-asynchronous-unwind-tables -Os     maps.c -o maps
quade@ezs-cocka:~/linux-magazin$ ./maps
code: 0x401000, const_data: 0x40207a, var_data: 0x403088, heap_data: 0x404000 stack_data: 0x7fffffffdf7c

00400000-00401000 r--p 00000000 103:04 19009103                          /home/quade/linux-magazin/maps
00401000-00402000 r-xp 00001000 103:04 19009103                          /home/quade/linux-magazin/maps
00402000-00403000 r--p 00002000 103:04 19009103                          /home/quade/linux-magazin/maps
00403000-00404000 rw-p 00002000 103:04 19009103                          /home/quade/linux-magazin/maps
00404000-04404000 rw-p 00000000 00:00 0                                  [heap]
7ffff7ff9000-7ffff7ffd000 r--p 00000000 00:00 0                          [vvar]
7ffff7ffd000-7ffff7fff000 r-xp 00000000 00:00 0                          [vdso]
7fffffffde000-7ffffffff000 rw-p 00000000 00:00 0                         [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]

quade@ezs-cocka:~/linux-magazin$
```

6 Die Datei `/proc/self/maps` liefert zahlreiche Infos zur Speicheraufteilung.

Listing 3: Programm kompilieren

```
$ c -isystem /usr/src/linux/tools/include/nolibc/ -nostdlib -nolibc \
    -no-pie -fno-asynchronous-unwind-tables -Os maps.c -o maps
```

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/48907

Weitere Infos und interessante Links
www.lm-online.de/qr/48907