Basics zum Linux-Kernel, Teil 5

Sie müssen nicht gleich einen selbst gebauten Kernel nutzen, um Einfallstore zu verrammeln. Wenn Sie es doch tun, sollten Sie sich aber Gedanken über die eingesetzte Linux-Version machen und Zeit für die Pflege einplanen.

Von Thorsten Leemhuis

Sicherheitsgewinn ohne eigenen Kernel

Ich möchte ein möglichst sicheres Linux-System bauen. Dazu erhielt ich den Rat, dieses mit einem selbst kompilierten Kernel zu versorgen; bei dem solle ich alle nicht benötigten Funktionen deaktivieren, um Angriffspunkte zu reduzieren. Mich schreckt aber der Einrichtungs- und Pflegeaufwand, den Sie im dritten Teil ihrer Kernel-FAQ (siehe ct.de/y745) umrissen haben. Geht das nicht einfacher?

Sie können die angesprochene Angriffsfläche schon ein gutes Stück reduzieren, indem Sie dem Kernel Ihrer Distribution das Nachladen von Modulen verbieten:

```
echo 1 > ↵
↳/proc/sys/kernel/modules_disabled
```

Diese Blockade können Sie im Betrieb nicht widerrufen, daher müssen Sie neu starten, um wieder Module laden zu können. Sie sollten den Befehl daher erst ausführen, wenn das System wirklich alle im Betrieb erforderlichen Kernel-Module geladen hat. Bei einer selbst gebauten Firewall und simplen Servern ist das oft schon am Ende des Boot-Prozesses der Fall, daher kann es dort ausreichen, den Befehl über die /etc/rc.d/rc.local auszuführen.

Bei komplexeren Servern und insbesondere Desktops ist das nicht so einfach, denn diese laden Module für einige essenzielle Funktionen erst bei Bedarf nach – etwa beim Start der grafischen Bedienoberfläche, beim Aufbau der VPN-Verbindung oder beim ersten Mounten eines USB-Sticks. Wer das Nachladen von Modulen bei solchen Computern untersagen will, sollte diese zuerst ein paar Tage oder Wochen normal benutzen und vor dem Herunterfahren jeweils mit `lsmod` dokumentieren, welche Module das System im Betrieb so nachzieht. Erstellen Sie dann die Datei /etc/modules-load.d/mymodules.conf, in der Sie jedes tatsächlich erforderliche Modul in einer eigenen Zeile auflisten, damit es der Kernel gleich beim Booten lädt. Details dazu erläutert die via `man modules-load.d` abrufbare Dokumentation.

Wollen Sie darüber hinaus das Laden einiger automatisch eingebundener Module untersagen, legen Sie eine Datei wie /etc/modprobe.d/myblacklist.conf an; füllen Sie die mit Zeilen wie `blacklist firewire_core`, um etwa das Kernel-Modul für Firewire-Hardware zu blockieren. Ein `man modprobe.d` liefert weitere Details zum Verfahren. Vorsicht: Sollte das Modul schon im Initramfs geladen werden, müssen Sie dieses neu erzeugen; bei Ubuntu & Co. etwa mit `update-initramfs -u`, bei Fedora mit `dracut -f`.

Nach diesen Vorbereitungen können Sie das Nachladen weiter Module untersagen, indem Sie den oben erwähnten Befehl am Ende des Startvorgangs ausführen lassen. Behalten Sie bei Desktop-PCs aber stets im Hinterkopf, dass dort kaum jemand so etwas macht. Viele Programme sind daher nicht auf derlei ausgelegt: Einige zeigen verwirrende Fehlermeldungen, wenn etwas nicht funktioniert, weil der Kernel ein benötigtes Modul nicht automatisch nachlädt.

Bedenken Sie, dass diese Tricks nur einen Angriffsvektor angehen. Der Aufwand lohnt sich daher nur, wenn Sie die Sicherheitsmaßnahmen auch an anderen Stellen verstärken. Daher sollten Sie unbedingt eine Distribution einsetzen, die Sicherheitslücken zuverlässig und zügig stopft. Außerdem sollten Sie möglichst viele der Sicherheitstechniken verwenden, die Linux-Kernel, Compiler & Co, dieser Tage bieten. Dazu gehört insbesondere der Einsatz von Sicherheitstechniken wie AppArmor oder SELinux, auch wenn die manchmal Arbeit machen oder bocken. Das ist wie mit dem Sicherheitsgurt im Auto: manchmal vielleicht ein wenig unbequem, aber eine wichtige Schutzmaßnahme – allerdings nur eine von vielen, die für zeitgemäße Sicherheit nötig ist.

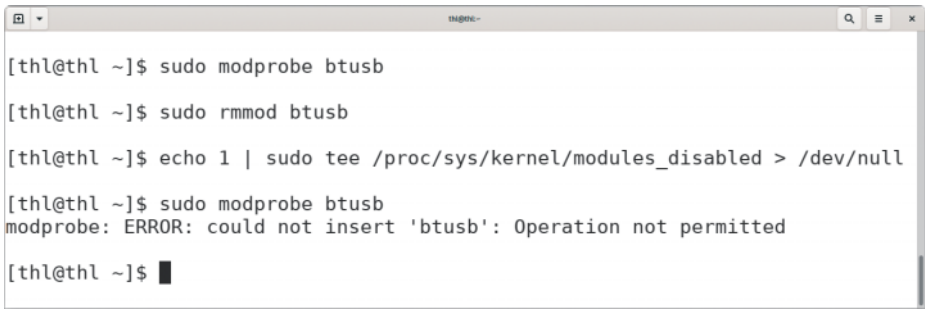
Basis für eigenen Kernel

Welche Linux-Version ist die beste Basis, wenn ich einen eigenen Kernel kompilieren will?

Nehmen Sie den aktuellen Stable-Kernel, wenn Sie möglichst neue Treiber brauchen und etwas über Linux lernen wollen. Die werden aber nur recht kurz gepflegt, daher müssen Sie bei solchen alle neun oder zehn Wochen auf eine neuere Versionslinie wechseln – etwa von 4.19 auf 4.20, 5.0, 5.1 und so weiter.

Wenn Sie solche Sprünge vermeiden wollen, sollten Sie den neuesten Longterm-Kernel einsetzen. Diese manchmal auch LTS-Kernel genannten Linux-Versionslinien versorgen die Entwickler mindestens zwei, oftmals sogar sechs Jahre lang mit Sicherheitskorrekturen. Meist wählen die Kernel-Entwickler rund um den Jahreswechsel eine Versionsline aus, die sie zum Longterm-Kernel erheben. Die zwei derzeit neuesten Longterm-Kernel sind Linux 4.19 und 5.4.

Der langen Supportzeit zum Trotz: Wenn Ihnen Sicherheit am Herzen liegt, sollten Sie einmal pro Jahr auf die jeweils neueste Longterm-Linie umsteigen. Korrekturen für manche Sicherheitslücken lassen sich nämlich nicht oder nur mühsam in ältere Kernel-Versionen zurück-



Sie können potenzielle Einfallstore verbarrikadieren, indem Sie das Nachladen von Kernel-Modulen untersagen.

portieren. Dadurch ist etwa der Schutz vor den Prozessorschwachstellen wie Spectre und Meltdown in Longterm-Kerneln wie 4.9 oder 4.4 seit jeher schlechter als bei jüngeren Versionslinien; außerdem dauert es oft länger, bis ältere Linien geschützt sind. Auf diese generellen Probleme wies Greg Koah-Hartman schon mehrfach deutlich hin (siehe ct.de/y745), der viele Stable- und Longterm-Kernel betreut und als zweitwichtigster Linux-Entwickler gilt.

Stable- und Longterm-Kernel haben einen Nachteil: Durchschnittlich erscheint in jeder Linie mindestens einmal pro Woche eine neue Version. Auf Sicherheit bedachte Nutzer müssen diese immer einspielen, denn jede stopft vermutlich Sicherheitslücken. Genau weiß man es aber nie, da die Linux-Entwickler nicht ausweisen, wenn sie welche korrigieren (siehe Teil 3 der FAQ: ct.de/y745).

Gemächlichere Kernel-Basis

Mir ist es zu aufwendig, meinen Kernel wöchentlich aktualisieren zu müssen. Kann ich das nicht irgendwie vermeiden?

Bauen Sie den Kernel aus dem Quellpaket, aus dem der Kernel Ihrer Distribution entsteht. Die Distributoren aktualisieren ihre Kernel nämlich seltener. Außerdem veröffentlichen sie Sicherheitshinweise, die gestopfte Lücken nennen. Sollte keine davon für Ihr System relevant sein, können Sie erwägen, sich das Kompilieren eines neueren Kernels zu sparen. Ganz sicher kann man sich aber auch nicht sein, denn Distributions-Entwickler erfahren manchmal erst nach Wochen von Lücken, die Stable- oder Longterm-Kernel bereits korrigiert haben.

Auch der Support-Zeitraum kann für die Kernel-Quellen des Distributors sprechen, schließlich pflegen einige ihre Kernel teilweise fünf bis zehn Jahre lang. Ein weiterer Grund: Manchmal enthalten die Quellen des Distributions-Kernels irgendwelche Treiber oder Korrekturen, die für Ihr System relevant sind, aber den bei Kernel.org erhältlichen Quellen fehlen.

Obacht: Ähnlich wie bei älteren Longterm-Versionslinien von Kernel.org sind Lücken in Distributionskerneln manchmal nicht so gut oder zügig gestopft wie in aktuellen Linux-Versionen.

Linux-Quellen besorgen

Ich baue meine Kernel-Images selbst, aber die Linux-Quellen über Tar-Archive oder Patches von kernel.org zu aktualisieren ist mühsam. Geht das nicht effizienter?

Nutzen Sie doch einfach direkt das Quellcodeverwaltungssystem, aus dem die Archive und Patches von Kernel.org entstehen. Falls Sie Mainline-Kernel bauen, legt der folgendem Git-Aufruf den aktuellen Stand im Verzeichnis `linux` ab:

```
git clone git://git.kernel.org/
↳pub/scm/linux/kernel/git/torvalds/
↳linux.git
```

So ein Klon belegt aber satte 2,5 GByte Platz, denn er umfasst die Quellen aller in den letzten 15 Jahren erschienenen Linux-Versionen. Wer sparen will, kann durch Angabe von `--depth 1` einen „Shallow"-Klon erzeugen, der mit dem aktuellen Stand beginnt. Beim Entstehen dieser Zeilen belegte so ein Klon circa 1,2 GByte.

Egal welchen der zwei Wege Sie wählen: Um den Quellcode auf den aktuellen Stand zu bringen, brauchen Sie in dem Verzeichnis nur `git pull` absetzen.

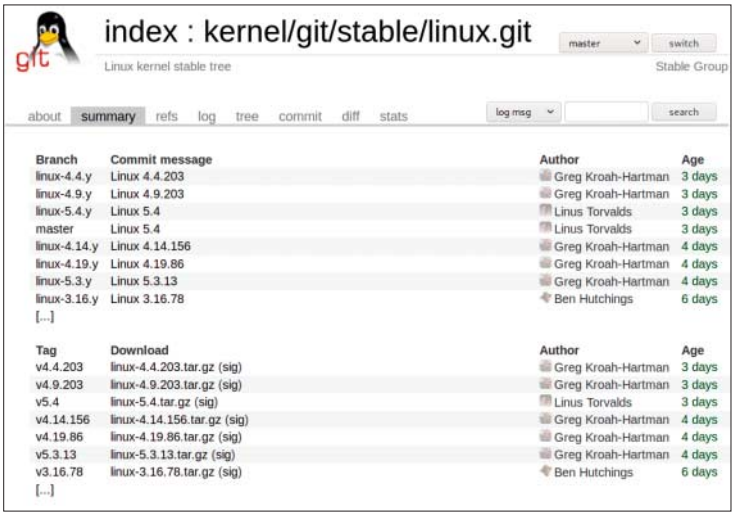
Das Ganze geht auch mit Stable- und Longterm-Kerneln, allerdings brauchen Sie für diese ein anderes Git-Repository. Die einzelnen Versionslinien finden sich dort in Zweigen (Git „Branches"), die man explizit auschecken muss:

```
git clone git://git.kernel.org/
↳pub/scm/linux/kernel/git/stable/
↳linux.git
git checkout linux-5.4.y
```

So ein Klon belegt sogar 2,7 GByte. Wer hier sparen will, muss nicht nur die Tiefe limitieren, sondern auch den gewünschten Zweig angeben:

```
git clone git://git.kernel.org/
↳pub/scm/linux/kernel/git/stable/
↳linux.git --depth 1
↳--branch linux-4.19.y
```

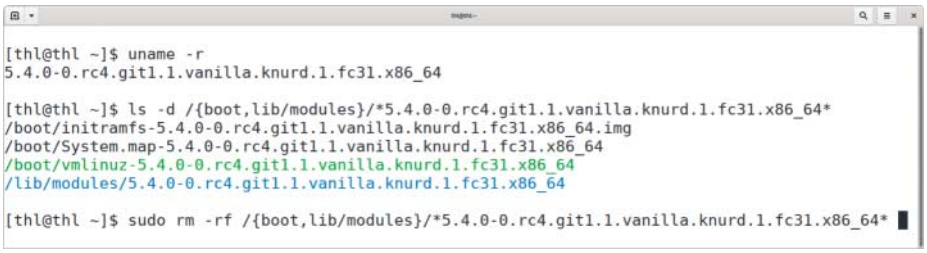
Diesen können Sie fortan via `git pull` aktualisieren. Wenn Sie allerdings irgendwann auf eine neuere Versionslinie wech-



Die einfachste Art, immer aktuelle Kernel-Quellen zu haben: die Git-Repositories der Entwickler nutzen.

seln wollen, müssen Sie den zuständigen Zweig mit einem Kommando wie dem folgenden nachholen:

```
git fetch --depth 1 origin ↵
↳linux-5.4.y:linux-5.4.y
git checkout linux-5.4.y
```



Ein manuell installiertes Kernel-Image und seine Module können Sie mit dem richtigen Kommando in Sekundenschnelle von der Platte putzen.

Linux-Kernel paketieren

Wie installiere ich einen selbst kompilierten Kernel, damit ich ihn später möglichst leicht und zugleich rückstandslos wieder loswerde?

Idealerweise bauen Sie dazu selbst Kernel-Pakete. Die dazu nötigen Befehle unterscheiden sich aber erheblich zwischen den Distributionen, daher können wir sie hier nur anreißen.

Bei Debian-basierten Distributionen empfiehlt sich der Bau, indem man in den Linux-Quellen ein `make deb-pkg` aufruft. Weitere Details zum Vorgehen erläutert ein Dokument der Debian-Entwickler (siehe ct.de/y745). Ähnlich können Sie über das Make-Target `binrpm-pkg` auch RPM-Pakete direkt aus den Linux-Quellen erzeugen. Sie sind aber nicht so gut auf die individuellen Gegebenheiten der verschiedenen Distributionen abgestimmt, die RPMs verwenden, daher kann bei der Deinstallation eher mal was zurückbleiben. Diese Gefahr kann man auf einem anderen Bauweg ausschließen: das Source-RPM mit dem Distributionskernel aktualisieren und das dann mit `rpmbuild` bauen.

Keiner dieser Wege lässt sich aber mal eben in einer Minute erklären, geschweige denn meistern. Wer die Einarbeitung scheut, installiert seinen Kernel einfach über das universelle `make modules_install install` und entfernt die installierten Dateien später manuell (siehe nächste Frage).

Kernel wieder loswerden

Wie werde ich einen Kernel wieder los, den ich manuell per `make modules_install install` installiert habe?

Dazu brauchen Sie dann die Versionsbezeichnung des zu entfernenden Kernels. Die erfahren Sie, indem sie den Kernel starten und darunter `uname -r` ausführen. Starten Sie dann noch mal mit einem anderen Kernel neu, um sich nicht den Ast abzusägen, auf dem Sie gerade sitzen.

Die beim Aufruf von `make` installierten Kernel-Module werden Sie dann schnell los, indem Sie das in /lib/modules/ zu findende Verzeichnis löschen, dessen Name identisch mit den Ausgaben des genannten `uname`-Befehls ist. Diese Versionsbezeichnung findet sich typischerweise auch in den Namen von drei anderen Dateien, die Sie loswerden wollen. Alle drei liegen in /boot/: das meist mit „vmlinuz-“ beginnenden Kernel-Image, das mit „initrd-“ oder „initramfs-“ beginnenden Initramfs und die Symbolinformationen, die in einer mit „System.map-“ beginnenden Datei liegen.

Nach dem Löschen dieses Trios verbleibt nur noch der Eintrag zum Start des Kernels in der Konfigurationsdatei des Boot-Managers. Den können Sie darin manuell entfernen; bei vielen Distributionen verschwindet der Eintrag aber auch automatisch, wenn Sie die Datei mit Skripten wie `update-grub` neu erzeugen.

All das ist nicht sonderlich schwer, birgt aber zahlreiche Stolperfallen. Beispielsweise ist /boot/vmlinuz bei manchen Distributionen ein symbolischer Link auf das zuletzt installierte Kernel-Image; wenn Sie Letzteres löschen, funktionieren womöglich einige Einträge im Boot-Menü nicht mehr. Wappnen Sie sich daher für solche Fälle und legen Sie ein Live-Linux zu Reparaturzwecken bereit!

Neues Linux für Raspi & Co

Ich nutze einen Einplatinencomputer (Single-Board Computer, SBC) mit der vom Hersteller bereitgestellten Distribution, die einen Linux-Kernel 4.19 verwendet. Jetzt habe ich aber einen WLAN-Stick gekauft, für den erst Version 5.4 einen Treiber mitbringt. Kann ich gefahrlos auf diese Version umsteigen?

Vermutlich nicht, denn Ihre Distribution verwendet wahrscheinlich gar nicht Linux 4.19 als Kernel, sondern nur einen darauf basierenden Kernel. Das klingt wie ein unwesentlicher, vernachlässigbarer Detailunterschied – aber das ist es keineswegs! Hersteller von SBC-spezifischen Distributionen verändern die als Basis dienenden Kernel-Quellen nämlich oft mit Hunderten oder Tausenden von Patches. Und die enthalten oft zahlreiche Treiber und Umbauten, ohne die der jeweilige Einplatinencomputer gar nicht oder nur rudimentär läuft.

Viele der Hersteller von SBCs kümmern sich nicht groß darum, diese Änderungen in den offiziellen Linux-Kernel zu integrieren. Man kann aber auch Glück haben, denn einige Hersteller oder Fans des SBC sorgen dafür, dass die Anpassungen in den offiziellen Kernel einfließen; oft dauert es aber nicht nur Monate, sondern Jahre, bis ein neu vorgestellter SBC im Upstream-Kernel so halbwegs alltagstauglich unterstützt wird, da die Änderungen der Hersteller vielfach die Qualitätsansprüche der Kernel-Entwickler nicht erfüllen und daher erst auf Vordermann gebracht werden müssen.

Außer diesem Problem gibt es noch ein weiteres: Ihre Distribution enthält womöglich proprietäre Userland-Treiber, die genau auf den mitlieferten Kernel abgestimmt sind. Vielfach laufen solche Treiber mit neueren Linux-Versionen gar nicht oder nur mit Mühen. Darüber hinaus sind Firmware, Boot-Loader und Kernel bei SBCs und den dort dominierenden Prozessoren mit ARM-Kern deutlich enger verzahnt als bei Computern mit x86-CPUs – das erschwert einen Wechsel auf eine neueren Linux-Kernel abermals.

(thl@ct.de)

Basics zum Linux-Kernel, Teil 1, 2, 3 und 4 sowie weitere Links: ct.de/y745