Linux-Kurs - Kernel

https://linuxkurs.ch

Wahlweise konntest Du dein System auf Slackware -current aktualisieren oder die stabile Version beibehalten. Nun ist es an der Zeit, dass wir uns das Herzstück deines Linux-Systems etwas genauer anschauen, dem Kernel.

Hinweis: Sofern nicht anders erwähnt, müssen alle folgenden Kommandos mit Root-Rechten ausgeführt werden.

initrd

Bisher verwenden wir den monolithischen *huge* Kernel. Dabei handelt es sich um eine grosse Datei, die alle wichtigen Treiber bereits enthält. Eine Alternative ist die Nutzung des sogenannten *generic* Kernels. Dieser enthält im Kernel-Image nur einen Bruchteil der Treiber. Um das System dennoch erfolgreich starten zu können, benötigst Du zusätzlich eine sogenannte *initrd*, was für *initial ramdisk* steht.

Diese initrd ist dynamisch und wird auf dein System angepasst. Beim Startvorgang lädt der Kernel die initrd nach und fährt dann mit dem init Prozess fort.

Slackware enthält ein Script mit dem Namen *mkinitrd_command_generator.sh*, welches dich bei der Erstellung der initrd unterstützt. Es überprüft welche Kernel-Module auf deinem System benötigt werden und gibt dir einen Befehl aus der zur Erstellung der Ramdisk ausgeführt werden kann:

`/usr/share/mkinitrd/mkinitrd_command_generator.sh`

```
root@darkstar:~# /usr/share/mkinitrd/mkinitrd_command_generator.sh
#
# mkinitrd_command_generator.sh revision 1.45
#
# This script will now make a recommendation about the command to use
# in case you require an initrd image to boot a kernel that does not
# have support for your storage or root filesystem built in
# (such as the Slackware 'generic' kernels').
# A suitable 'mkinitrd' command will be:

mkinitrd -c -k 4.19.80 -f ext4 -r /dev/sda2 -m jbd2:mbcache:crc32c-intel:ext4 -u -o /boot/initrd.gz
root@darkstar:~#
```

Ohne weitere Parameter bezieht sich das Script auf den aktuell laufenden Kernel. Es wird standardmässig keine Ramdisk erstellt sondern nur der Befehl ausgegeben, mit dem diese generiert werden kann.

Markiere und kopiere die letzte Zeile und füge Sie in der Shell ein. Mit Enter kannst Du den Befehl auslösen. Dadurch wird eine Ramdisk im Verzeichnis /boot mit dem Namen initrd.gz erstellt.

Um den Kernel mit dieser initrd zu starten, müssen entsprechende Einträge in der Bootloader-Konfiguration hinterlegt werden.

Füge dazu folgende Zeilen am Ende der */etc/lilo.conf* vor dem Hinweis *# Linux bootable partition config ends* hinzu:

```
image = /boot/vmlinuz-generic
  initrd = /boot/initrd.gz
  root = /dev/sda2
  label = Linux-Generic
  read-only
```

Führe `lilo` aus und versuche die Kernel-Versionen mit der Ramdisk zu starten. Der Startvorgang sollte ein wenig schneller gehen als mit dem *huge* Kernel.

Die initrd muss nach jedem Kernel-Update neu erstellt werden und der Bootloader lilo muss ausgeführt werden damit die Änderungen aktiv werden. Um diesen Prozess zu vereinfachen hat der Slackware-Benutzer *zerouno* ein Script erstellt, welches nach einem Kernel-Upgrade mittels slackpkg die Möglichkeit bietet die initrd neu zu erstellen und lilo auszuführen. Du kannst es wie folgt herunterladen und aktivieren:

```
wget https://linuxkurs.ch/kurs/zlookkernel.sh -O
/usr/libexec/slackpkg/functions.d/zlookkernel.sh
chmod +x /usr/libexec/slackpkg/functions.d/zlookkernel.sh
```

Beim nächsten Kernel-Update mittels slackpkg wirst Du abschliessend gefragt ob die initrd neu erstellt werden soll und lilo ausgeführt werden soll. Sofern Du dies mit **Y** beantwortest erfolgt der Prozess automatisch.

Eigenen Kernel erstellen

Wir konnten die Kernel Varianten *huge* und *generic* der Distribution kennenlernen. Es kann allerdings vorkommen, dass eine neuere Kernelversion benötigt wird. Grund dafür kann zum Beispiel fehlende Hardwareunterstützung in der Distributions-Version sein. In solchen Fällen kann es sinnvoll sein einen einen eigenen Kernel zu compilieren.

Wir gehen davon aus das Du den *generic* Kernel gestartet hast. Sollte dies nicht der Fall sein, starte dein System neu und wähle im lilo Bootmenü den Punkt: *Linux-Generic*.

Um einen eigenen Kernel compilieren zu können, benötigen wir die entsprechenden Quellen. Die offiziellen Kernel-Quellen findest du auf: https://www.kernel.org

Kopiere dir beispielsweise im Firefox den Link auf die aktuelle stabile Kernelversion indem du mit der rechten Maustaste auf *[tarball]* klickst und **Link-Adresse kopieren** wählst. Öffne daraufhin ein Terminal und wechsle in das Verzeichnis */usr/src* in dem sich üblicherweise die Kernel-Quellen befinden:

`cd /usr/src`

Mit Hilfe des **wget** Befehls kannst Du das Kernel-Archiv herunterladen. Gebe dazu `wget` gefolgt von einem Leerzeichen ein und füge den zuvor kopierten Link mit `Ctrl + Shift + V` ein.

Zum Beispiel:

`wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.4.1.tar.xz`

Entpacken kannst Du das Archiv mit folgendem Kommando:

```
tar -xvpf linux-5.4.1.tar.xz
```

Passe dabei den Dateinamen entsprechend an und wechsle daraufhin in den neu erstellten Pfad, in unserem Beispiel mit Hilfe von:

```
cd linux-5.4.1
```

Als Basis für den neuen Kernel kannst Du die Konfiguration des aktiven *Generic* Kernels verwenden. Mit folgendem Befehl kannst Du die bisherige Konfiguration übernehmen:

```
zcat /proc/config.gz > .config
```

Der Befehl `zcat` verhält sich ähnlich wie der bereits bekannte `cat` Befehl, ermöglicht aber auch das Anzeigen von gzip-komprimierten Dateien. Das `>` Zeichen leitet die Ausgabe um, in diesem Falle in die Datei *.config* im aktuellen Verzeichnis.

Mit Hilfe von `make olddefconfig` kann daraufhin die alte Konfiguration adaptiert werden:

```
make olddefconfig
```

Nun ist es möglich Anpassungen an der Kernelkonfiguration vorzunehmen. Dabei hilft dir der Befehl `make menuconfig`. Alternativ gibt es eine grafische Konfigurationsoberfläche die Du mit `make xconfig` aufrufen kannst.

Anpassungen am Kernel sind optional. Als Beispiel möchten wir die Unterstützung für das Sicherheitsframework AppArmor aktivieren. Dieses bietet zusätzlichen Schutz auch vor noch nicht öffentlich bekannten Sicherheitslücken, sogenannten Zero-Day-Exploits.

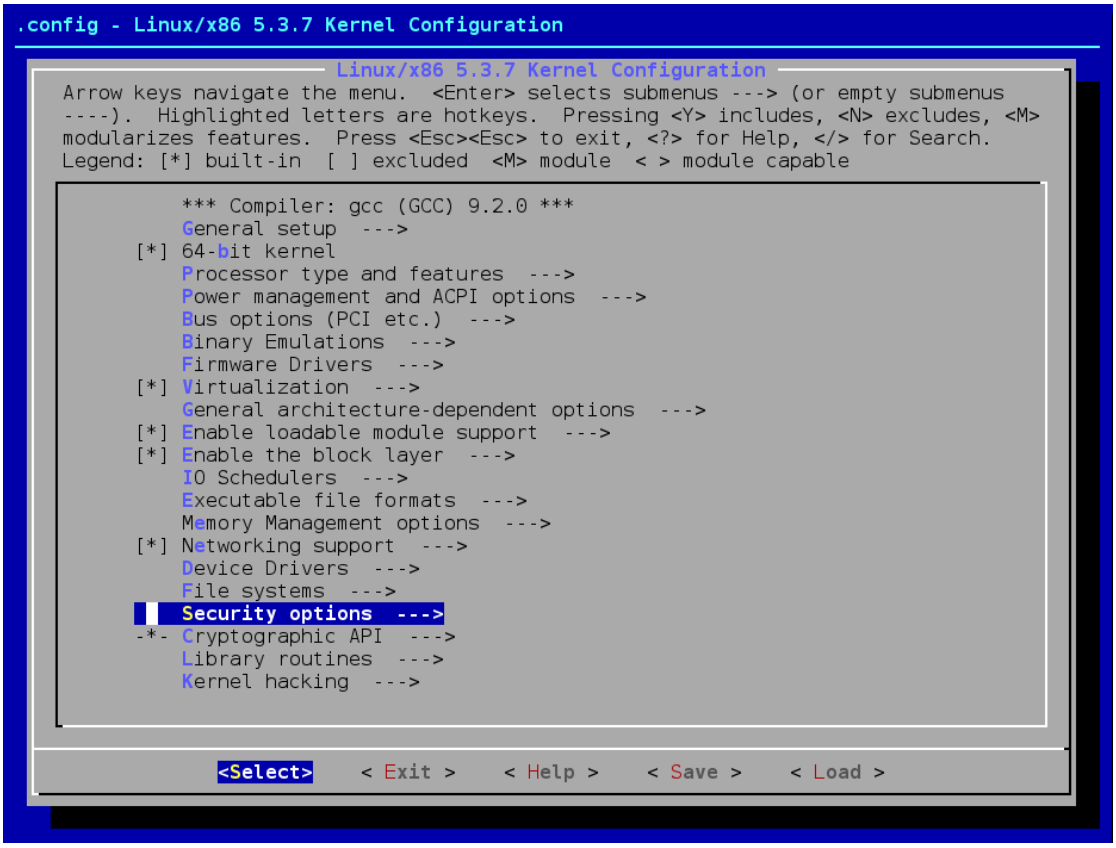
Starte im Verzeichnis in dem sich die Kernel-Sourcen befinden den folgenden Befehl:

```
make menuconfig
```

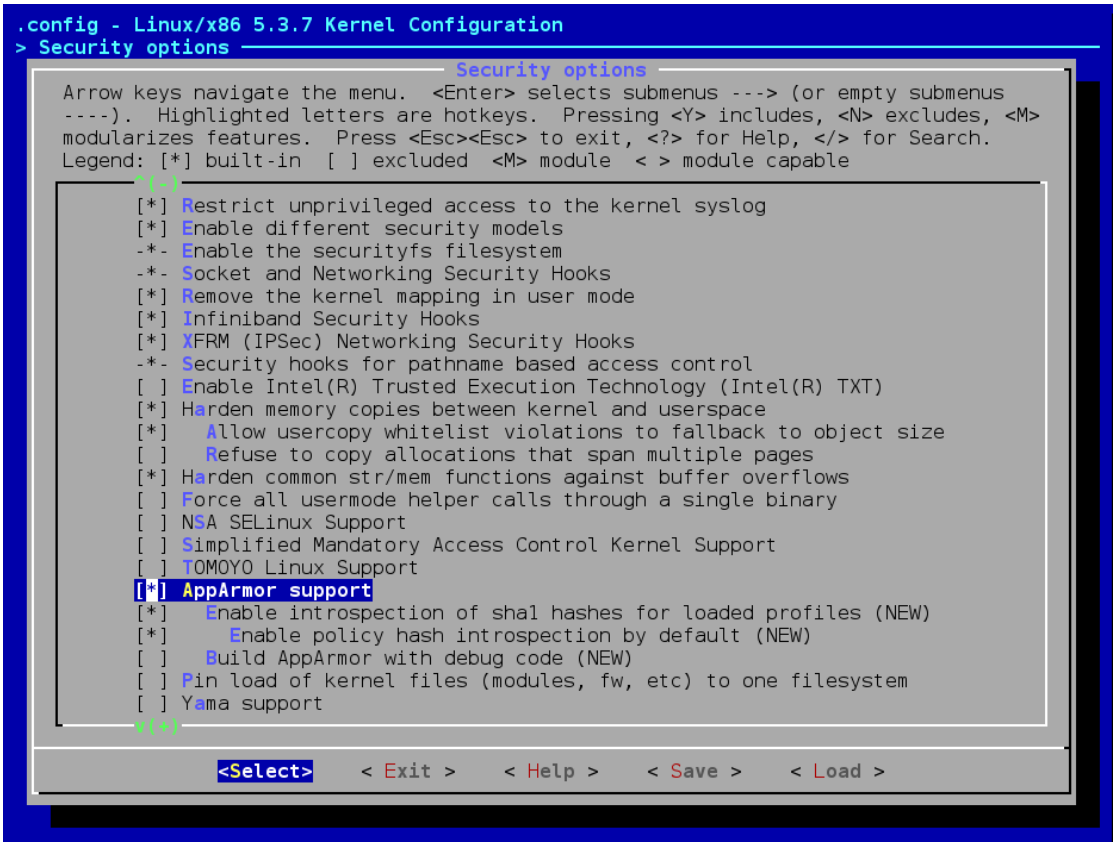
Es begrüsst dich eine TUI-Oberfläche in der Du wie gewohnt navigieren kannst. Wenn Du ein Untermenü öffnest, kannst Du über den **Exit** Befehl eine Ebene zurückspringen.

Hinweis: Im Hauptmenü beendet **Exit** die Anwendung.

Navigiere mit den Cursortasten oder durch Aufruf des Kurzbefehls zum Punkt *Security Options* und öffne das Untermenü durch Eingabe von Enter, sofern der Punkt **Select** aktiviert ist.



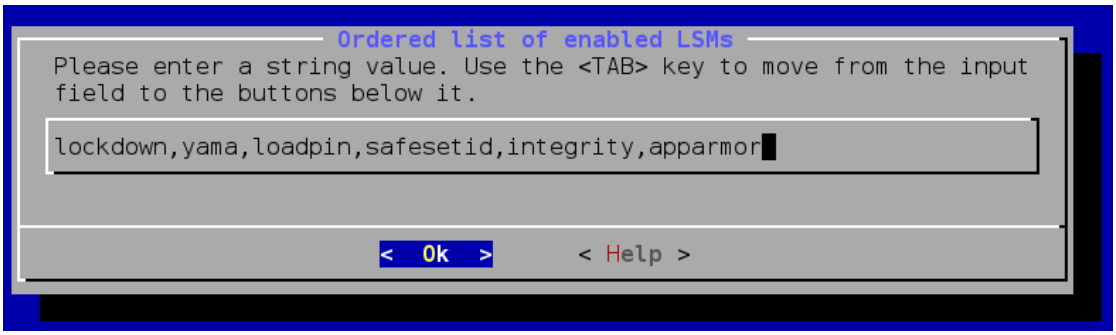
Scrolle in der sich öffnenden Parameterliste bis zu den Punkt AppArmor Support und aktiviere diesen mit der Space-Taste.



Des weiteren muss das AppArmor *lockdown* Modul zur Liste der *LSMs* (Linux Security Modules) hinzugefügt werden. Wähle dazu weiter unten den Punkt **Ordered list of enabled LMSs** aus:

```
[ ] TOMOYO Linux Support
[*] AppArmor support
[*]   Enable introspection of sha1 hashes for loaded profiles
[*]     Enable policy hash introspection by default
[ ]   Build AppArmor with debug code
[ ] Pin load of kernel files (modules, fw, etc) to one filesystem
[ ] Yama support
[ ] Gate setid transitions to limit CAP_SET{U/G}ID capabilities
[ ] Integrity subsystem
    First legacy 'major LSM' to be initialized (Unix Discretionary Access Control
(lockdown,yama,loadpin,safesetid,integrity) Ordered list of enabled LSMs
    Kernel hardening options  --->
```

Es öffnet sich eine Liste der aktivierten Security Module. Füge dort **,apparmor** hinzu:



Du kannst bei Bedarf weiter Änderungen vornehmen. Speichere deine Änderungen ab, indem Du mit der Tab-Taste den Punkt **Save** anwählst und mit Enter bestätigst.

Du wirst nach einem Dateinamen für die Kernel-Konfigurationsdatei gefragt. Standardmässig lautet dieser *.config*. Wir gehen davon aus, dass Du die Einstellung unverändert beibehältst.

Als nächstes kannst Du die Anwendung mit Hilfe des **Exit** Befehls beenden. Du musst ihn möglicherweise mehrmals anwählen, falls Du dich in einem Untermenü befindest.

Mit folgendem Befehl startest Du den Compiliervorgang:

```
make bzImage modules
```

Es werden das Kernel-Image (*bzImage*) sowie die Kernelmodule (*modules*) compiliert. Dieser Vorgang kann einige Zeit in Anspruch nehmen.

Die compilierten Kernelmodule müssen daraufhin in den entsprechenden Pfad kopiert werden. Standardmässig wäre dies */lib/modules/* gefolgt von der Kernelversion. Dabei hilft dir das folgende Kommando:

```
make modules_install
```

Der Befehl kopiert nicht nur die Kernelmodule an den richtigen Ort sondern führt abschliessend auch das `depmod` Kommando aus. Einige Kernelmodule haben Abhängigkeiten, die mit Hilfe von **depmod** aufgelöst und in der Datei */lib/modules/KERNELVERSION/modules.dep* hinterlegt werden.

Das eigentliche Kernel-Image muss nun in den */boot* Pfad kopiert werden:

```
cp arch/x86/boot/bzImage /boot/vmlinuz-5.4.1
```

Ersetze die Kernel-Version (in diesem Beispiel *5.4.1*) durch diejenige des Kernels den Du compiliert hast.

Es ist sinnvoll auch die *System.map* sowie die Kernel-Konfiguration in den */boot* Pfad zu kopieren:

```
cp System.map /boot/System.map-5.4.1
cp .config /boot/config-5.4.1
```

Auch hier muss der Dateiname entsprechend angepasst werden, so dass er der compilierten Kernelversion entspricht. Die *System.map* wird im Fehlerfalle zum Debugging genutzt. Ausführliche Informationen dazu findest du auf folgender Webseite: https://rlworkman.net/system.map/

Ramdisk

Da wir den neuen Kernel auf der Basis des *generic* Kernels erstellt haben, ist eine Ramdisk zwingend notwendig. Zur Erstellung kannst Du das *mkinitrd_command_generator.sh* Script zur Hilfe nehmen.

`$(/usr/share/mkinitrd/mkinitrd_command_generator.sh -k 5.4.1 -a "-o /boot/initrd-5.4.1.gz" --run)`

Ersetze hierbei die Kernelversion durch die deines Kernels. Der Parameter *--run* gibt nur das Kommando zur Erstellung der initrd ohne weitere Bemerkungen aus. Da wir die gesamte Zeichenkette in *$()* stellen, wird diese von der BASH als Befehl interpretiert, also direkt ausgeführt und die initrd erzeugt.

Jedes Mal wenn Du einen neuen Kernel compilierst und verwenden möchtest musst Du initrd erneut erzeugen.

Bootloader

Abschliessend wird der Bootloader angepasst, so dass Du in der Lage bist deinen neuen Kernel zu starten.

Füge dazu einen entsprechenden Eintrag vor dem Hinweis *# Linux bootable partition config ends* zu der */etc/lilo.conf* Konfigurationsdatei hinzu:

```
image = /boot/vmlinuz-5.4.1
  initrd = /boot/initrd-5.4.1.gz
  root = /dev/sda2
  label = Linux-5.4.1
  read-only
```

Wobei die Versionsnummer entsprechend angepasst werden muss, so dass sie deinem Kernel entspricht.

Führe `lilo` aus und versuche deine erste selbstcompilierte Kernel-Versionen zu starten.

AppArmor

Wir haben den Kernel nun um AppArmor-Unterstützung erweitert. Nun fehlt nur noch die passende Anwendung. Diese lässt sich mit Hilfe des SlackBuild-Scriptes erstellen:

`sbopkg -i apparmor`

Damit AppArmor beim Systemstart automatisch geladen wird, kann ein entsprechender Eintrag zur Datei */etc/rc.d/rc.local* hinzugefügt werden:

`vi /etc/rc.d/rc.local`

```
# Start apparmor
if [ -x /etc/rc.d/rc.apparmor ]; then
  /etc/rc.d/rc.apparmor start
fi
```

Starte dein System neu und untersuche ob AppArmor aktiv ist:

`aa-status`

Damit hast du erfolgreich deinen eigenen Kernel compiliert.
