ICT: Linux §6
User, File ACL, Bash, ...

Harald Schilly
harald@schil.ly
http://harald.schil.ly/

050126

Benutzer- und Gruppenverwaltung/1

Um auf einem Linux System arbeiten zu können, muss man sich als Benutzer einloggen. Dies wird über zwei zentrale Dateien verwaltet:

- `/etc/passwd + /etc/groups`

Da diese Dateien für den Benutzer lesbar sein müssen, dürfen keine Passwörter (auch wenn verschlüsselt) darin abgelegt werden. Diese befinden sich in "Shadow" Dateien: (siehe `man shadow`)

- `/etc/shadow + /etc/gshadow`

Ein Benutzer kann in mehreren Gruppen Mitglied sein, er kann seine momentane Haupt-Gruppe mit dem Befehl "`newgrp`" wechseln. Das ist der einzige Grund warum Gruppen ein Passwort haben können.

Beim Bearbeiten dieser Dateien dürfen keine gleichzeitigen Veränderungen passieren und den Syntax darf nicht zu zerstört werden. Wird die Datei ungültig, kann man sich das nächste mal nicht mehr einloggen.
Verwende immer **`vipw`** und **`vigr`** !!!

Benutzer- und Gruppenverwaltung/1

Ein Benutzer kann in mehreren Gruppen Mitglied sein, er kann seine momentane Haupt-Gruppe mit dem Befehl "**`newgrp`**" wechseln. Das ist der einzige Grund warum Gruppen ein Passwort haben können.

Frage: Wofür kann das gut sein?

Alle Gruppen eines Users sieht man mit: **`$ groups <name>`**

Aktuelle ID und Gruppe mit: **`$ id`**

Benutzer- und Gruppenverwaltung/2

Neben dem Einloggen gibt es auch ein Home-Verzeichnis und eine zugewiesene default Shell. Das wird in `/etc/passwd` angegeben.

Die Einträge können auch mit diversen Hilfsprogrammen vorgenommen werden:

- `useradd, userdel, usermod`
- `groupadd, groupdel, groupmod`

Freundlicher und (Debian/Ubuntu)-System angepasst ist

- **`adduser`**, `deluser`, `addgroup`, `delgroup`

... insbesondere wird hier das HOME Verzeichnis angelegt.
Für mehr Details siehe `/etc/adduser.conf` !

Passwörter mit `passwd` und `gpasswd` setzen. Benutzer mit `adduser user group` zu einer Gruppe hinzufügen. Standardmäßig wird das HOME-Verzeichnis aus dem in `/etc/skel` generiert. Alle Dateien werden auf den neuen Benutzer als Besitzer umgeschrieben.

Natürlich gibt es auch GUI Tools die das erledigen. In Gnome: Administration > Benutzer + Gruppen [Freischalten mit Adminrechten] und konfigurieren.

Benutzer- und Gruppenverwaltung/2

Grundlegend:

- `useradd, userdel, usermod`
- `groupadd, groupdel, groupmod`

Freundlicher und (Debian/Ubuntu)-System angepasst ist

- `adduser`, `deluser`, `addgroup`, `delgroup`

Frage:

- Wieso gibt es beide Varianten?
- Was macht add... was ...add nicht kann und umgekehrt?

Benutzer- und Gruppenverwaltung/3

Der Administrator in Linux Systemen heißt "root". Normalerweise hat nur dieser Account zusätzliche Fähigkeiten.
z.B:

- Auf jede Datei zugreifen können, lesen können, löschen, usw.
- Dateisysteme mounten
- Programme installieren
- Benutzerverwaltung
- …

Soll dieselbe Maschine von mehreren Admins verwaltet werden, gibt es ein Problem. Welches?

Benutzer- und Gruppenverwaltung/3

Füher gab es einen einzigen Administrator namens "root". Will man mehreren Benutzern Administrationsrechte geben, hatten sie alle dasselbe Passwort - das ist schlecht :-(
Daher gibt es üblicherweise folgenden Mechanismus: Gehört ein Benutzer zu einer besonderen Gruppe ("admin", ...) so kann er mittels 'sudo' und seinem eigenen Passwort vorübergehend Administrator werden.
-> siehe `/etc/sudoers` und `man sudoers`

Zwei Beispiele:

```
sudo apt-get install softwarepaket
```

Öffnen einer Bash Shell als Benutzer "anderer" und stelle das Home-Verzeichnis auf dessen Home-Verzeichnis um.

```
sudo -H -u anderer bash
```

Dateiberechtigungen setzen / 1

chmod ändert die Berechtigungen

- `chmod [-R] NNN <Datei(en)>`
  -R rekursiv, NNN ist die Oktalzahl, eine oder mehrere Dateien
- `chmod [-R] [ugoa][+-=][rwx] <Datei(en)>`
  - u = "user", g = "group", o = "other", a = "all"
  - + = setzen, - löschen (falls gesetzt), = genau darauf setze

Beispiel:

```
$ chmod a=rwx brokenlinks.py
$ ls -lah brokenlinks.py
-rwxrwxrwx 1 schilly cma 2.8K 2009-10-21 19:40 brokenlinks.py*
$ chmod o-rwx brokenlinks.py
$ ls -lah brokenlinks.py
-rwxrwx--- 1 schilly cma 2.8K 2009-10-21 19:40 brokenlinks.py*
$ chmod a-x brokenlinks.py
$ ls -lah brokenlinks.py
-rw-rw---- 1 schilly cma 2.8K 2009-10-21 19:40 brokenlinks.py
```

Dateiberechtigungen setzen / 2

`chgrp` gruppe <file> ... ändert die Gruppenzugehörigkeit

`chown` ... ändert den Eigentümer

- **`chown user:gruppe <file>`** ändert beides

Bemerkung: Ein normaler Benutzer kann das üblicherweise aus Sicherheitsgründen nicht - "`sudo`" gibt dem Befehl Administrationsrechte.

Beispiel:

```
$ ls -lah file
-rw-r--r-- 1 harri harri 0 2009-10-21 21:31 file
$ sudo chown linux file
[sudo] password for harri:
$ ls -lah file
-rw-r--r-- 1 linux harri 0 2009-10-21 21:31 file
$ sudo chown linux:linux file
$ ls -lah file
-rw-r--r-- 1 linux linux 0 2009-10-21 21:31 file
```

ACL - access control lists /1

ACLs regeln Zugriffsrechte die über das Benutzer/Gruppe/Welt Schema hinausgehen. Grundsätzlich funktioniert es nur mit Dateisystemen die es auch unterstützen. *Welche sind das?*

ACL muss beim Mounten explizit aktiviert werden:

- Option '`acl`' in `/etc/fstab` (siehe `man mount`)

Nun kann man zu jeder Datei und jedem Verzeichnis eine längere Liste angeben wo jeder Eintrag zusätzliche Regeln festlegt:

- Einen Benutzer explizit ausschließen, der sonst Zugriff hätte.
- Eine Gruppe zugreifen lassen, die keinen Zugriff hätte.

Für Details und Entscheidungsalgorithmus: `man acl`
Befehle: **`setfacl`** / **`getfacl`** ("+" im `ls -l` Listing)

ACL - access control lists /2

setfacl/getfacl Beispiele

Granting an additional user/group read/rwx access

```
setfacl -m u:lisa:r file
setfacl -m g:grp1:rwx file
```

Revoking write access from all groups and all named users (using the *effective rights* mask)

```
setfacl -m m::rx file
```

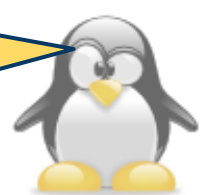
Removing a named group entry from a file's ACL

```
setfacl -x g:staff file
```

Copying the ACL of one file to another

```
getfacl file1 | setfacl --set-file=- file2
```

ACL aktivieren ohne Neustart:
1. /etc/fstab anpassen (",acl" bei den Optionen anhängen)
2. sudo mount -o remount / [bzw. /home]



Programmieren/Hintergrund

- Kompiliert: C, C++, fortran, lisp, go ...
  - GCC (Gnu C Compiler Suite) + `gdb` debugger
  - `objdump` (dis-assembler)
- Interpretiert: Bash, Python + Python-Libs, Lua, Ruby, ...
- Mischung: Java, Scala, ...

Hilfsmittel:

- Versionskontrolle & Patches
  - `diff` und `patch`
  - mercurial (hg), git, svn, perforce, ...

Projektmanagement:

- trac, bugzilla, Jira

Toolchain C

1. Schreibe C Datei: `file.c` (ev. mit Header: `file.h`)
2. Kompiliere: **`cc -o file -I<headerdir> file.c`**
3. Bibliothek: `cc -shared -o libBibliothek.so bibliothek.c`
   dann: `cc -o file -lBibliothek -L<pfad> file.c`
4. führe Programm normal aus: `./file`
5. Debuggen: Kompiliere mit option "`-g`" -> starte Prog. mit `gdb`

Beispiel:

```
#include <stdio.h>
int main() {
 int i;
 for (i = 1; i<=10; i++) {
   printf("hallo %d\n" , i);
 }
return 0;
}
```

C++ wird ganz ähnlich mit g++ kompiliert.

Toolchain C / 2

Inhalt der kompilierten Datei (jetzt Programm!):

```
$ objdump -d [-S] file
```

```
...
int main() {
 int i;
 for (i = 1; i<=10; i++) {
 804840c: 83 44 24 1c 01          add    $0x1,0x1c(%esp)
 8048411: 83 7c 24 1c 0a          cmpl   $0xa,0x1c(%esp)
 8048416: 7e df                   jle    80483f7 <main+0x13>
   printf("hallo %d\n" , i);
 }
 return 0;
 8048418: b8 00 00 00 00          mov    $0x0,%eax
}
 804841d: c9                      leave
 804841e: c3                      ret
 804841f: 90                      nop
...
```

Die CPU bekommt diese Bit's

Bash Wiederholung 1

Bash-Skript fängt mit einem Shabang `#!/usr/bin/env bash` an.

Variablen ohne Leerzeichen definieren: `i=4` oder `y="ein wort"`

Zuweisungen von Berechnungen (expr, bc): `x=`expr $i + 1``

- `$()` oder `` `...` `` für sub-shells
- `$(( ... ))` ... shellinterne Berechnung

Ausgabe mit `echo`, bzw. pipe `|` oder `>` file

Befehle mit Argumenten aufrufen: `befehl arg1 arg2`

- built-in Befehl der Shell
- selbstdefinierte Funktion
- Programm im `$PATH`

Laden eines Skripts (Bibliothek von Funktionen) mit
`$ source file.sh` bzw. `$ . file`

Bash WH 2: Kontrollstrukturen

```
if BEDINGUNG
then
   BLOCK
else [ elif BEDINGUNG; then ]
   BLOCK
fi
```

Bedingungen:
[-a FILE] .. Datei existiert
[-s FILE] .. Größe > 0
[-x FILE] .. ausführbar
[STRING1 == STRING2]
  [[STRING1 == STRING2]] (!!!)
[A1 OP A2] .. Vergleich
  -lt (less than), -le (less/equal), ...
[E1 -a E1] .. UND (-o .. ODER)

LDP/abs $7.1, Teil 1 von §4

```
while [until] BEDINGUNG
do
   BLOCK
done
```

```
for VAR in LISTE
do
    BLOCK  mit $VAR
done
```

```
case EXPR in
 CASE1)
    BLOCK
  ;;
  ...
esac
```

Kontrollstrukturen / for & seq

seq ... erzeugt eine Sequenz von Ganzzahlen

```
$ for i in $(seq 14 16); do echo $i; done
14
15
16

# C-syntax
$ for (( i=0; i<2; i=i+1)); do echo $i ; done
0
1

# in einem Verzeichnis wo es nur file1 und file2 gibt
$ for f in $(ls *); do echo $f ; done
file1
file2
```

Quoting & Escaping

Quoting heißt, dass ein Inhalt unter Anführungszeichen nicht ausgeführt wird, sondern so wie er ist belassen wird.

- `"..."` ... **schwaches Quoting**, Variablen werden ersetzt
- `'...'` ... **starkes Quoting**, Bash belässt den Inhalt
- `eval [args]` ... liest `args` und führt Befehl(e) aus
  - `$ eval 'echo ab'  -> ab`
- `$ R="echo x y"; eval '$R z'  -> x y z`

Escaping dient dazu, besondere Zeichen einzufügen (\ Zeichen).

- `\n` ... neue Zeile, `\\` ein `\`, `\r` ... Cursor zurück zum Zeilenanfang
  `"\ "` ... Leerzeichen (zB in Dateinamen)
- `'..'\''..'` (nur ') fügt ein ' in einem '..' String ein:
  `$ echo 'text'\''text'  -> text'text`

nützliches Beispiel für Carriage Return:

```
for i in $(seq 4 10); do
   echo -n $i; sleep 1; echo -en "\r    \r";
done
```

Funktionen definieren

In der Bash können *Funktionen* definiert, aufgerufen und *Parameter* übergeben werden. Das macht Code erst richtig wiederverwendbar!

```
function f () { echo $1 ;}    # oder nur  f () { echo $1 ;}
$ f abc
abc

function f2 () { echo $(($1+$2));  echo "$@"; }
$ f2 123 888
1011
123 888

# Variable definiert?
$ f3 () { if [ -z $2 ] ; then echo "nicht definiert"; fi; }
$ f3
nicht definiert
$ f3 1
nicht definiert
$ f3 a b
```

alle, immer mit ""

bash Befehl "shift" verschiebt Argumente nach vorn.

Hintergrundprozesse in bash

Es können Programme im Hintergrund laufen, als Kindprozess der bash (Terminal) und parallel weitergearbeitet werden. Ausgaben (stdout, stderr) werden angezeigt, sofern sie nicht umgeleitet werden.

- **<Befehl> &** ... Startet Befehl im Hintergrund
- **STRG-Z** ... Stoppt Vordergrundprozess (SIGSTOP)
  - **bg** ... setzt Ausführung im Hintergrund fort
  - **fg** ... setzt Ausführung im Vordergrund fort
- **jobs <Zahl>** ... Hintergrundjobs anzeigen
  - -p ... Zeigt PIDs an
- **kill %<Jobzahl>** ... besondere Version von kill
- **wait PID** ... warte bis der Prozess fertig ist:
  Beispiel Synchronisationspunkt in Skript:
  `for p in $(jobs -p); do wait $p; done`

Copyright & Lizenz