science + computing
A Bull Group Company

strace – für `bash`-Versteher

Endlich die UNIX-Shell verstehen dank `strace`

Harald König

science + computing ag
IT-Dienstleistungen und Software für anspruchsvolle Rechnernetze
Tübingen | München | Berlin | Düsseldorf

Agenda

science + computing
A Bull Group Company

1. me and s+c
2. Warum?
   - Einige Beobachtungen...
3. Let's use `strace`
   - Erste Schritte mit `strace`
4. Einfache Grundregeln
   - globbing, wildcards
   - quoting
   - pipes
5. Beispiele
   - Einige Beobachtungen...

<me>

science + computing
A Bull Group Company

- OpenSource in der Schule (Pet-2001 / CBM-3032)
- TEX ab 1986 an der Uni
- VMS (1985) und UNX (1987) an der Uni
- Nie wieder INTEL (1989/90 – Intel-Assembler :-(
- Linux 0.98.4 (1992, doch wieder Intel :-(
- XFree86 (S3, 1993-2001)
- science + computing ag in Tübingen seit 2001
- . . .

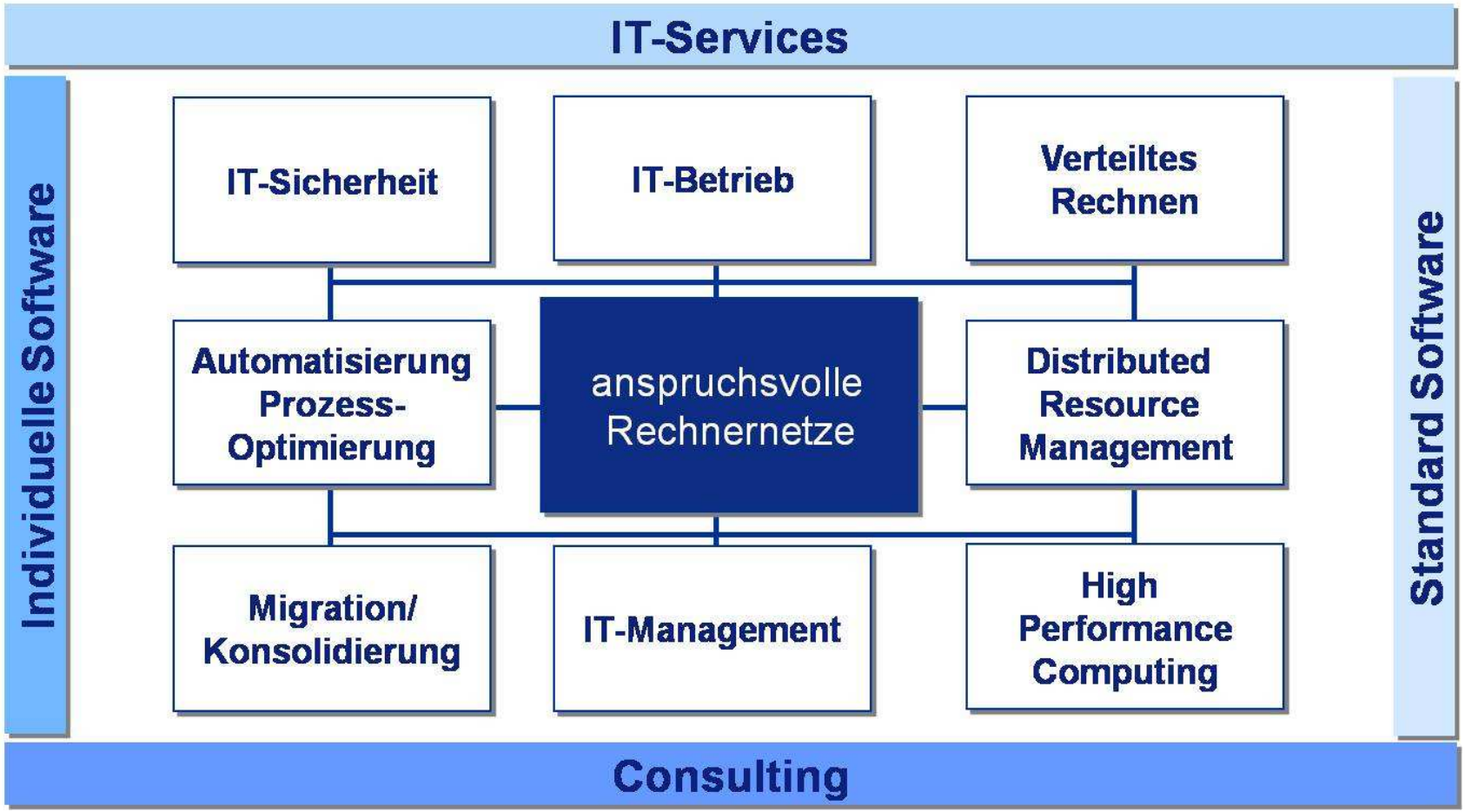
science + computing ag (s+c)

science + computing
A Bull Group Company

Gegründet	1989
Büros	Tübingen München Berlin Düsseldorf
Mitarbeiter	270
Besitzer	Bull S.A. (100 %) seit 10/2008
Jahresumsatz	26 Mio. Euro (07/08)

S+C

science + computing
A Bull Group Company



S+C

science + computing
A Bull Group Company

- Automobilindustrie
- Anlagen- und Maschinenbau
- Luft- und Raumfahrt
- Mikroelektronik
- Chemie / Pharma
- Biotechnologie
- Öffentlicher Dienst

Warum?

science + computing
A Bull Group Company

```
insmod *.ko

echo "Hallo"

cat < /etc/passwd | sort

find -name *.c
```

Let's use `strace`

science + computing
A Bull Group Company

- Datei-Zugriffe
- Programm-Aufrufe
- Datenfluß, Replay
- time stamps und kernel delay
- Statistik

Man Pages

- `man strace`
- `man gdb`
- `man ptrace`
- `man ltrace`
- `man bash`

Erste Schritte mit `strace`

science + computing
A Bull Group Company

```
$ strace emacs
$ strace $( pgrep httpd | sed 's/^/-p/' )
$ strace emacs 2> OUTFILE
$ strace -o OUTFILE emacs
$ strace -e open   cat /etc/HOSTNAME
$ strace -e file   cat /etc/HOSTNAME

$ strace           -p $BASHPID
$ strace -e execve -p $BASHPID
....
```

Alles weitere in den Proceedings bzw. im `xterm`...

Warum?

science + computing
A Bull Group Company

```
insmod *.ko

echo "Hallo"

cat < /etc/passwd | sort

find -name *.c
```

globbing, wildcards

science + computing
A Bull Group Company

quoting

science + computing
A Bull Group Company

pipes

Smartmeter mit `strace`

science + computing
A Bull Group Company

```
tty=/dev/ttyUSB0

stty 50 time 1 min 1 -icanon < $tty

strace -ttt -e write dd if=$tty of=/dev/null bs=1 2>&1 |
tee -a vz1.data |
awk 'NR==1{ t0=$1 }
     /write\(1, ".*"..., 1\)  *= 1/{
         t=$1; print 3600e3 /(t-t1)/2000 ,  t-t0,t-t1,$0;t1=t
         t=int($1 + 0.5)
         system("echo wget -O-  \"http://localhost/volkszaehle
    '
```

strace für bash-Versteher

Harald König
H.Koenig@science-computing.de
koenig@linux.de
http://www.science-computing.de/

Die UNIX-Shell ist ein sehr mächtiges und hilfreiches Werkzeug. Leider verzweifeln nicht nur Linux-Anfänger immer wieder an ihr, wenn man nicht ein paar einfache Grundregeln und Shell-Konzepte kennt. Dieser Beitrag erklärt einige der wesentlichen Grundkonzepte in UNIX und der Shell, und illustriert und zeigt diese „Live“ mit Hilfe von strace. Dabei lernt man so nebenbei, wie nützlich strace beim Verstehen und Debuggen von Shell-Skripten und allen anderen UNIX-Prozessen sein kann.

1 Einleitung

Die UNIX/Linux-Shell (hier immer am Beispiel der bash) ist ein sehr einfaches, aber mächtiges Tool. Doch viele Benutzer haben oft Probleme, dass die Shell nicht ganz so will wie sie: Quoting, Wildcards, Pipes usw. sind immer wieder Ursache für Überraschungen und „Fehlverhalten“. Welches echo ist denn (wann?) das richtige?

```
$ echo Hallo                    $ echo  Hallo Chemnitz
$ echo 'Hallo'                  $ echo  Hallo     Chemnitz
$ echo "Hallo"                  $ echo 'Hallo     Chemnitz'
```

Nach vielen Beobachtungen von solch schwierigen Entscheidungen in der freien Command-Line-Bahn war der letzte Auslöser für diesen Vortrag hier

```
# insmod *.ko
```

die Frage „Kann denn /sbin/insmod auch mit Wildcards umgehen?“.

2 UNIX-Grundlagen: fork() und exec()

Eines der universellen Prinzipien in UNIX ist der Mechanismus zum Starten aller neuen Programme: mit dem System-Call fork() wird ein neuer Prozess erzeugt und dann das neue Programm mit dem System-Call exec() gestartet. Genau dies ist die Hauptaufgabe jeder UNIX-Shell, neben vielen einfacheren internen Shell-Kommandos. Dabei werden die Command-Line-Argumente für das neue Programm *einzeln* im exec()-Aufruf dem Programm übergeben:

```
main() { execl("/bin/echo", "programm", "Hallo", "Chemnitz 1"   , 0); }
main() { execl("/bin/echo", "programm", "Hallo", "Chemnitz", "1", 0); }
```

oder schöner

```
#include <unistd.h>
int main(int arc, char *argv[])
{
  char *const argv[] = { "programm", "Hallo", "Chemnitz 2", NULL };
  return execve("/bin/echo", argv , NULL) ;
}
```

Wie die Parameter für das neue Programm in diesen `exec()` System-Call kommen, ist Aufgabe der Shell. Wie das neue Programm dann diese einzelnen Argumenten (Optionen, Dateinamen, usw.) interpretiert und verarbeitet, liegt ausschließlich beim aufgerufenen Programm.

In Linux heißt der vorhandene Kernel-Call für `fork()` seit anno 1999 mit Kernel-Version 2.3.3 `clone()`, welcher von einer in der `glibc` definierten Funktion `fork()` aufgerufen wird. Alle Varianten des System-Calls `exec()` rufen in der `glibc` den einen Kernel-Call `execve()` auf.

Daher ist es entscheidend zu verstehen, wie die Shell die eingegebene Command-Line verarbeitet und anschließend dem aufzurufenden Programm in `execve(...)` übergibt. Diese Übergabe im Kernel-Call lässt sich nun sehr schön mit `strace` überwachen und anzeigen, womit man exakt sehen kann, was die Shell aus der Eingabe tatsächlich gemacht hat und wie die Aufruf-Parameter des Programms schließlich aussehen.

3 Das Handwerkszeug zur Erkenntnis: `strace`

`strace` kennt viele Optionen und Varianten. Hier können nur die in meinen Augen wichtigsten angesprochen und gezeigt werden – die Man-Page (RTFM: `man strace`) birgt noch viele weitere Informationsquellen und Hilfen.

3.1 Erste Schritte mit `strace`

Je nachdem, was man untersuchen will, kann man (genau wie mit Debuggern wie `gdb`) ein Kommando entweder mit `strace` neu starten, oder aber man kann sich an einen bereits laufenden Prozess anhängen und diesen analysieren:

```
$ strace emacs
```

bzw. für einen schon laufenden `emacs`

```
$ strace -p $( pgrep emacs )
```

und wenn es mehrere Prozesse gleichzeitig zu tracen gibt (z.B. alle Instanzen des Apache `httpd`):

```
$ strace $( pgrep httpd | sed 's/^/-p/' )
```

Das an einen Prozess angehängte `strace -p ...` kann man jederzeit mit `CTRL-C` beenden, das untersuchte Programm läuft dann normal und ungebremst weiter. Wurde

das zu testende Programm durch strace gestartet, dann wird durch CTRL-C nicht nur strace abgebrochen, sondern auch das gestartete Programm beendet.

strace gibt seinen gesamten Output auf stderr aus, man kann jedoch den Output auch in eine Datei schreiben bzw. die Ausgabe umleiten, dann jedoch inklusive dem stderr-Outputs des Prozesses (hier: emacs) selbst:

```
$ strace -o OUTFILE emacs
$ strace emacs 2> OUTFILE
```

Üblicherweise wird jeweils eine Zeile pro System-Call ausgegeben. Diese Zeile enthält den Namen der Kernel-Routine und deren Parameter, sowie den Rückgabewert des System-Calls. Der Output von strace ist sehr C-ähnlich, was nicht sehr wundern dürfte, da das Kernel-API von Linux/UNIX ja als ANSI-C-Schnittstelle definiert ist (meist didaktisch sinnvoll gekürzte Ausgabe):

```
$ strace cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 130 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)         = 3
read(3, "harald.science-computing.de\n", 32768) = 28
write(1, "harald.science-computing.de\n", 28) = 28
read(3, "", 32768)                      = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

Selbst ohne C-Programmierkenntnisse lässt sich diese Ausgabe verstehen und vernünftig interpretieren. Details zu den einzelnen Calls kann man in den entsprechenden Man-Pages nachlesen, denn alle System-Calls sind dokumentiert in Man-Pages (man execve ; man 2 open ; man 2 read write usw.). Nur aus der Definition des jeweiligen Calls ergibt sich, ob die Werte der Argumente vom Prozess an den Kernel übergeben, oder aber vom Kernel an den Prozess zurückgegeben werden (bspw. den String-Wert als zweites Argument von read() und write() im letzten Beispiel).

Für einige System-Calls (z.B. stat() und execve()) erzeugt strace mit der Option -v eine „verbose“ Ausgabe mit mehr Inhalt. Auf der Suche nach mehr Informationen (stat()-Details von Dateien, oder komplettes Environment beim execve()) kann dies oft weiterhelfen.

```
$ strace -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 130 vars */]) = 0
harald.science-computing.de
```

```
$ strace -v -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], ["LESSKEY=/etc/lesskey.bin", "MAN
PATH=/usr/local/man:/usr/loca"..., "XDG_SESSION_ID=195", "TIME=\\t%E real,\\t%
U user,\\t%S sy"..., "HOSTNAME=harald", "GNOME2_PATH=/usr/local:/opt/gnom"...,
 "XKEYSYMDB=/usr/X11R6/lib/X11/XKe"..., "NX_CLIENT=/usr/local/nx/3.5.0/bi"...,
 "TERM=xterm", "HOST=harald", "SHELL=/bin/bash", "PROFILEREAD=true", "HISTSIZE
=5000", "SSH_CLIENT=10.10.8.66 47849 22", "VSCMBOOT=/usr/local/scheme/.sche"..
[ ... many lines deleted ...]
rap"..., "_=/usr/bin/strace", "OLDPWD=/usr/local/nx/3.5.0/lib/X"...]) = 0
```

3.2 Ausgabe von `strace` einschränken

Der Output von `strace` kann schnell *sehr* umfangreich werden, und das synchrone Schreiben der Ausgabe auf `stderr` oder auch eine Log-Datei kann erheblich Performance kosten. Wenn man genau weiß, welche System-Calls interessant sind, kann man die Ausgabe auf einen bzw. einige wenige Kernel-Calls beschränken (oder z.B. mit `-e file` alle Calls mit Filenames!). Das verlangsamt den Programmablauf weniger und vereinfacht das spätere Auswerten des `strace`-Outputs erheblich. Allein die Option `-e` bietet sehr viel mehr Möglichkeiten. Hier nur ein paar einfache Beispiele, welche sehr oft ausreichen, alles Weitere dokumentiert die Man-Page:

```
$ strace -e open             cat /etc/HOSTNAME
open("/etc/HOSTNAME", O_RDONLY)         = 3

$ strace -e open,read,write  cat /etc/HOSTNAME
open("/etc/HOSTNAME", O_RDONLY)         = 3
read(3, "harald.science-computing.de\n", 32768) = 28
write(1, "harald.science-computing.de\n", 28harald.science-computing.de
) = 28
read(3, "", 32768)                      = 0

$ strace -e open,read,write  cat /etc/HOSTNAME > /dev/null
open("/etc/HOSTNAME", O_RDONLY)         = 3
read(3, "harald.science-computing.de\n", 32768) = 28
write(1, "harald.science-computing.de\n", 28) = 28
read(3, "", 32768)                      = 0

$ strace -e file             cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 130 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)         = 3
```

3.3 Mehrere Prozesse und Kind-Prozesse tracen

`strace` kann auch mehrere Prozesse gleichzeitig tracen (mehrere Optionen `-p PID`) bzw. auch alle Kind-Prozesse (Option `-f`) mit verfolgen. In diesen Fällen wird in der Ausgabe am Anfang jeder Zeile die PID des jeweiligen Prozesses ausgegeben. Alternativ kann man die Ausgabe auch in jeweils eigene Dateien je Prozess schreiben lassen (mit Option `-ff`).

Wenn man nicht genau weiß, was man denn eigentlich tracen muss, dann ist die Verwendung von `-f` oder `-ff` angesagt, da man ja nicht ahnen kann, ob evtl. mehrere (Unter-) Prozesse oder Skripte involviert sind. Ohne `-f` oder `-ff` könnten sonst beim Trace wichtige Informationen von weiteren Prozessen entgehen. In meinen einfachen Beispielen mit `emacs` ist dieser selbst auch schon ein Wrapper-Shell-Skript. Hier nun ein paar Varianten als kleine Denksportaufgabe zum Grübeln, wie die `bash` intern so tickt:

```
$ strace -e execve bash -c true
execve("/bin/bash", ["bash", "-c", "true"], [/*...*/]) = 0

$ strace -e execve bash -c /bin/true
execve("/bin/bash", ["bash", "-c", "/bin/true"], [/*...*/]) = 0
execve("/bin/true", ["/bin/true"], [/*...*/]) = 0

$ strace -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/*...*/]) = 0

$ strace -f -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/*...*/]) = 0
execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
execve("/bin/false", ["/bin/false"], [/*...*/) = 0

$ strace -o OUT -f -e execve bash -c "/bin/true ; /bin/false"
$ grep execve OUT
1694 execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], []) = 0
1695 execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
1696 execve("/bin/false", ["/bin/false"], [/*...*/]) = 0

$ strace -o OUT -ff -e execve bash -c "/bin/true ; /bin/false"
$ grep execve OUT*
OUT.2155:execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"],[]) = 0
OUT.2156:execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
OUT.2157:execve("/bin/false", ["/bin/false"], [/*...*/]) = 0
```

3.4 Lasst uns die bash tracen

Um nun einer interaktiven Shell „auf die Finger“ zu schauen, starten wir zwei Terminals. Im ersten Fenster fragen wir die PID der Shell, welche wir nun beobachten wollen, ab mit

```
$ echo $$
12345
```

und im zweiten Fenster starten wir dann mit dieser Information die Überwachung mit `strace`, welche alle aufgerufenen Programme ausgeben soll (ohne oder mit Option `-v`, je nachdem ob das Environment und lange/viele Argumente wichtig sind oder nicht):

```
$ strace -f -e execve -p 12345
$ strace -f -e execve -p 12345 -v
```

Im Folgenden wird in den Beispiel-Ausgaben immer zuerst die Shell-Eingabe und Ausgabe dargestellt, anschließend der zugehörige Output von `strace`.

Und schon geht es los …

4 Globbing (Wildcards)

Zunächst sehen wir uns den Inhalt eines Verzeichnisses mit dem Kommando `ls` an. Einige Dateien werden hier mit einem Stern (*) verziehrt. Grund hierfür ist die magisch aufgetauchte Option „`-F`“, welche von einem `alias` für `ls` stammt:

```
$ ls
echo1*  echo1.c  hello*  hello.c  hello.cpp  hello.h  test.c  test.ko
execve("/bin/ls", ["ls", "-F"], [/*...*/]) = 0

$ type -a ls
ls is aliased to `ls -F'
ls is /bin/ls
```

Um diesen `alias` zu umgehen, kann man entweder `/bin/ls` direkt aufrufen, oder `\ls` eingeben, beides hat die selbe Wirkung. Will man nur die C-Quell-Dateien sehen, so übergibt man `ls` das Argument `*.c`

```
$ /bin/ls
echo1  echo1.c  hello  hello.c  hello.cpp  hello.h  test.c  test.ko
execve("/bin/ls", ["ls"], [/*...*/]) = 0

$ \ls *.c
echo1.c  hello.c  test.c
execve("/bin/ls", ["ls", "echo1.c", "hello.c", "test.c"], [/*...*/]) = 0
```

doch mit `strace` sieht man, dass `ls` überhaupt nicht `*.c` als Parameter bekommt, sondern eine Liste aller einzelnen Dateinamen der C-Quellen. Das Expandieren von Wildcards (Globbing) wird bereits von der Shell übernommen, darum muss sich `ls` und alle anderen Programme nicht mehr selbst kümmern.

Wenn man das Expandieren der Dateinamen in der Shell mit Anführungszeichen (Quoting) verhindert, dann bekommt `ls` wirklich das Argument `*.c` übergeben und weiss damit nichts anzufangen:

```
$ \ls "*.c"
ls: cannot access *.c: No such file or directory
execve("/bin/ls", ["ls", "*.c"], [/*...*/]) = 0
```

Damit beantwortet sich auch die eingangs gestellte Frage nach `/sbin/insmod`:

```
# insmod *.ko
execve("/sbin/insmod", ["/sbin/insmod", "test.ko"], [/*...*/]) = 0

# insmod "*.ko"
insmod: can't read '*.ko': No such file or directory
execve("/sbin/insmod", ["/sbin/insmod", "*.ko"], [/*...*/]) = 0
```

`/sbin/insmod` kann mit Wildcards ebenso wenig umgehen wie `/bin/ls`, der Befehl `insmod` erwartet als Argument auf der Kommando-Zeile einen korrekten Dateinamen eines Kernel-Moduls. Bei `insmod` sollte man jedoch beachten, dass dieses Kommando nur ein einziges Kernel-Modul auf der Kommando-Zeile erwartet. Daher sollte man `insmod *.ko` so nur Ausführen, wenn man sicher ist, dass sich nur genau ein Kernel-Modul `*.ko` im Verzeichnis befindet!

Das klassische Gegenbeispiel dieses ansonsten UNIX-üblichen Verhaltens ist der Befehl `find` mit seiner Option `-name`. Hier kann man nach `-name` ein „Pattern“ mit Wildcards angeben, nach welchem gesucht wird. Es führt zu Fehlern oder überraschendem Verhalten, wenn die Wildcards nicht gequotet und deshalb von der Shell expandiert werden. Im ersten Beispiel wird ausschließlich nach `test.ko` gesucht und nicht

nach allen Kernel-Modulen (Überraschung), im zweiten Beispiel gibt es eine kryptische Fehlermeldung von `find`, da man dessen Syntax nicht eingehalten hat:

```
$ find .. -name *.ko
../dir/test.ko
execve("/usr/bin/find", ["find", "..", "-name", "test.ko"], [/*...*/]) = 0

$ find .. -name *.c
find: paths must precede expression: hello.c
Usage: find [-H] [-L] [-P] [-Olevel] [-D help|tree|search|stat|rates|opt|exec]
            [path...] [expression]
execve("/usr/bin/find",["find", "..", "-name", "echo1.c", "hello.c", "test.c"],[]) = 0
```

Hier muss man mit Anführungszeichen oder Backslash das Expandieren in der Shell verhindern, damit `find` für `-name` das richtige Pattern übergeben bekommt:

```
$ find .. -name \*.ko
execve("/usr/bin/find", ["find", "..", "-name", "*.ko"], [/*...*/]) = 0
../dir/test.ko
../dir2/test2.ko

$ find .. -name "*.c"
../dir/echo1.c
../dir/hello.c
../dir/test.c
../dir2/test2.c
execve("/usr/bin/find", ["find", "..", "-name", "h*.c"], [/*...*/]) = 0
```

5 Quoting

Schon die Beispiele zu Wildcards zeigen, dass man in der Shell ab und an „quoten“ muss. Man fragt sich nur, wann und wie man es richtig macht. In diesem Zusammenhang muss man die Zeichen mit Sonderfunktionen in der Shell kennen, die man evtl. im Einzelfall umgehen will.

Hierzu gehören, wie schon gesehen, die Wildcards für Pattern (* und ?), das Dollarzeichen ($) für Variablennamen u.ä., Größer-, Kleiner- und Pipe-Zeichen (< > |) für Ein- und Ausgabe-Umleitung sowie Pipes, runde Klammern für Sub-Shells, das &-Zeichen für Hintergrund-Prozesse, das Ausrufezeichen (!) für die Shell-History, die sogenannten White-Space-Characters (Space, Tab, Zeilenende), der Backquote (‘) für die Ausgaben-Einfügung, sowie natürlich auch die Quoting-Zeichen (\ ’ ") selbst, und vermutlich noch ein paar weitere hier vergessene Zeichen.

Die Shell kennt drei Methoden, Zeichen mit Sonderfunktionen in der Shell, wie z.B. die Wildcards, zu „quoten“: den Backslash (\), die einfachen (’) und doppelten (") Anführungszeichen (engl. *quotes*, daher der Begriff Quoting). Die (Sonder-) Funktion der drei Quoting-Zeichen ist auch schnell erklärt:

Der Backslash (\) wirkt auf das nächste Zeichen und hebt dessen Sonderfunktion auf.

In Zeichenketten innerhalb von einfachen Anführungszeichen (’) gibt es keinerlei Sonderfunktionen mehr, mit der einzigen Ausnahme des (’) selbst, welches die gequotete Zeichenkette beendet. Auch der Backslash hat keine Sonderfunktion mehr.

Daher kann auch das Single-Quote in solchen Zeichenketten *nicht* gequotet werden. Explizite Single-Quotes kann man nur außerhalb gequoteter Strings bzw. innerhalb doppelter Anführungszeichen verwenden.

In Zeichenketten innerhalb von doppelten Anführungszeichen (") sind noch einige wenige Sonderzeichen aktiv: das Dollarzeichen ($) für Variablennamen, das Backquote (‘) für die Ausgaben-Einfügung und das Ausrufezeichen (!) bei der interaktiven Eingabe, sowie der Backslash (\) um diese Zeichen und auch sich selbst quoten zu können. Leerzeichen, Tabular-Zeichen und Zeilenenden bleiben in beiden Arten der Anführungszeichen erhalten.

echo ist in der bash ein *interner* Befehl und wird dadurch nicht von `strace` erfasst, auch nicht mit einem Backslash davor, wie bei dem Alias von `ls`:

```
$ echo Hallo    Chemnitz
Hallo Chemnitz
$ \echo Hallo              Chemnitz
Hallo Chemnitz
```

Um Leerzeichen zu erhalten, muss gequotet werden, ob mit einfachen oder doppelten Anführungszeichen oder mit Backslashes ist hier unerheblich:

```
$ /bin/echo Hallo    Chemnitz
Hallo Chemnitz
execve("/bin/echo", ["/bin/echo", "Hallo", "Chemnitz"], [/*...*/]) = 0
$ /bin/echo "Hallo    Chemnitz"
Hallo    Chemnitz
execve("/bin/echo", ["/bin/echo", "Hallo    Chemnitz"], [/*...*/]) = 0
$ /bin/echo 'Hallo    Chemnitz'
Hallo    Chemnitz
execve("/bin/echo", ["/bin/echo", "Hallo    Chemnitz"], [/*...*/]) = 0
$ /bin/echo Hallo\ \ \ \ Chemnitz
Hallo    Chemnitz
execve("/bin/echo", ["/bin/echo", "Hallo    Chemnitz"], [/*...*/]) = 0
```

Doch auch schon richtig gequotete Leerzeichen können schnell wieder verschwinden, wenn man einmal das Quoten vergisst

```
$ a="A  B     C"
$ /bin/echo $a
A B C
execve("/bin/echo", ["/bin/echo", "A", "B", "C"], [/*...*/]) = 0
$ /bin/echo "$a"
A  B     C
execve("/bin/echo", ["/bin/echo", "A  B     C"], [/*...*/]) = 0
$ /bin/echo '$a'
$a
execve("/bin/echo", ["/bin/echo", "$a"], [/*...*/]) = 0
```

und auch Wildcards müssen stets gebändigt und im Quote-Zaum gehalten werden,

solange man deren Expansion nicht wünscht:

```
$ b="mit find kann man nach *.c suchen..."
$ /bin/echo "$b"
mit find kann man nach *.c suchen...
execve("/bin/echo", ["/bin/echo", "mit find kann man nach *.c suchen..."], []) = 0

$ /bin/echo $b
mit find kann man nach echo1.c hello.c test.c suchen...
execve("/bin/echo", ["/bin/echo", "mit", "find", "kann", "man", "nach",
                     "echo1.c", "hello.c", "test.c", "suchen..."], []) = 0
```

Benötigt man auf einer Kommandozeile sowohl die Wirkung von einfachen als auch doppelten Anführungszeichen, so muss man diese entsprechend abwechselnd öffnen und schliessen, oder man verwendet Backslashes wo möglich:

```
$ echo 'Die Variable "a" wird mit $a ausgelesen und hat den Wert '"'$a'"
Die Variable "a" wird mit $a ausgelesen und hat den Wert 'A  B     C'
execve("/bin/echo", ["/bin/echo", "Die Variable \"a\" wird mit $a ausgelesen
                     und hat den Wert 'A  B     C'"], [/*...*/]) = 0

$ echo "Die Variable \"a\" wird mit \$a ausgelesen und hat den Wert '$a'"
Die Variable "a" wird mit $a ausgelesen und hat den Wert 'A  B     C'
execve("/bin/echo", ["/bin/echo", "Die Variable \"a\" wird mit $a ausgelesen
                     und hat den Wert 'A  B     C'"], [/*...*/]) = 0

$ /bin/echo Die Variable \"a\" wird mit \$a ausgelesen und hat den Wert \'"$a"\'
Die Variable "a" wird mit $a ausgelesen und hat den Wert 'A  B     C'
execve("/bin/echo", ["/bin/echo", "Die", "Variable", "\"a\"", "wird", "mit",
       "$a", "ausgelesen", "und", "hat", "den", "Wert", "'A  B     C'"], []) = 0
```

und noch einmal kurz die schon besprochenen Wildcards

```
$ /bin/echo *.c
echo1.c hello.c test.c
execve("/bin/echo", ["/bin/echo", "echo1.c", "hello.c", "test.c"], [/*...*/]) = 0

$ /bin/echo \*.c
*.c
execve("/bin/echo", ["/bin/echo", "*.c"], [/*...*/]) = 0

$ /bin/echo '*.c'
*.c
execve("/bin/echo", ["/bin/echo", "*.c"], [/*...*/]) = 0
```

6 Welche Login-Skripte wurden ausgeführt

... ist mit `strace` auch ganz allgemein zu prüfen und zu verifizieren. Entweder kann man der entsprechenden Shell vertrauen, dass diese sich z.B. mit deren Option `--login` wirklich identisch zu einem „echten“ Login verhält. Mit dieser Annahme reicht es, eine File-Liste zu erstellen und diese zu analysieren. Im Folgenden bleibt die `FILELIST` unsortiert, denn auch die Reihenfolge der Login-Skripte ist natürlich interessant Ansonsten sind diese Listen mit Dateinamen nach `sort -u` meist leichter lesbar.

```
$ strace -o OUT -e open -f bash --login -i
exit
$ grep -v ENOENT OUT | grep \" | cut -d\" -f2 > FILELIST1
$ strace -o OUT -e file -f bash --login -i
exit
$ grep \" OUT | cut -d\" -f2 > FILELIST2

$ egrep "^/etc|$HOME" FILELIST1
/etc/profile
/etc/bash.bashrc
/etc/venus/env.sh
/etc/bash.local
/home/koenig/.bashrc
/etc/bash.bashrc
/etc/venus/env.sh
/etc/bash.local
/home/koenig/.bash_profile
/home/koenig/.profile.local
/home/koenig/.bash_history
/etc/inputrc
/home/koenig/.bash_history
```

Wenn man der Shell nicht so ganz traut, dann sollte man die „Quelle“ eines Logins selbst tracen! Als Nicht-`root` kann man dies z.B. noch mit `xterm -ls` versuchen. Oder man traced als `root` direkt den SSH-Daemon an Port 22 oder ein `getty`-Prozess mit allen Folge-Prozessen:

```
$ lsof -i :22
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
sshd    2732 root    3u  IPv4  15551      0t0  TCP *:ssh (LISTEN)
$ strace -p 2732 -ffv -e file -o sshd.strace ....
```

7 Conclusion

Was man mit `strace` noch alles über die Skript-Ausführung erfahren und das gesamte UNIX-/Linux-System lernen kann, ist Stoff für weitere Vorträge. Der Phantasie bietet sich hier viel Raum für weitere Einsatzmöglichkeiten.

Nun einfach viel Spaß mit der Shell, und Erfolg und neue UNIX-Erkenntnisse beim `stracen`!

Literatur

[1] RTFM: `man strace bash`

[2] http://linux.die.net/man/1/bash — Man-Page von `bash`

[3] http://linux.die.net/man/1/strace — Man-Page von `strace`