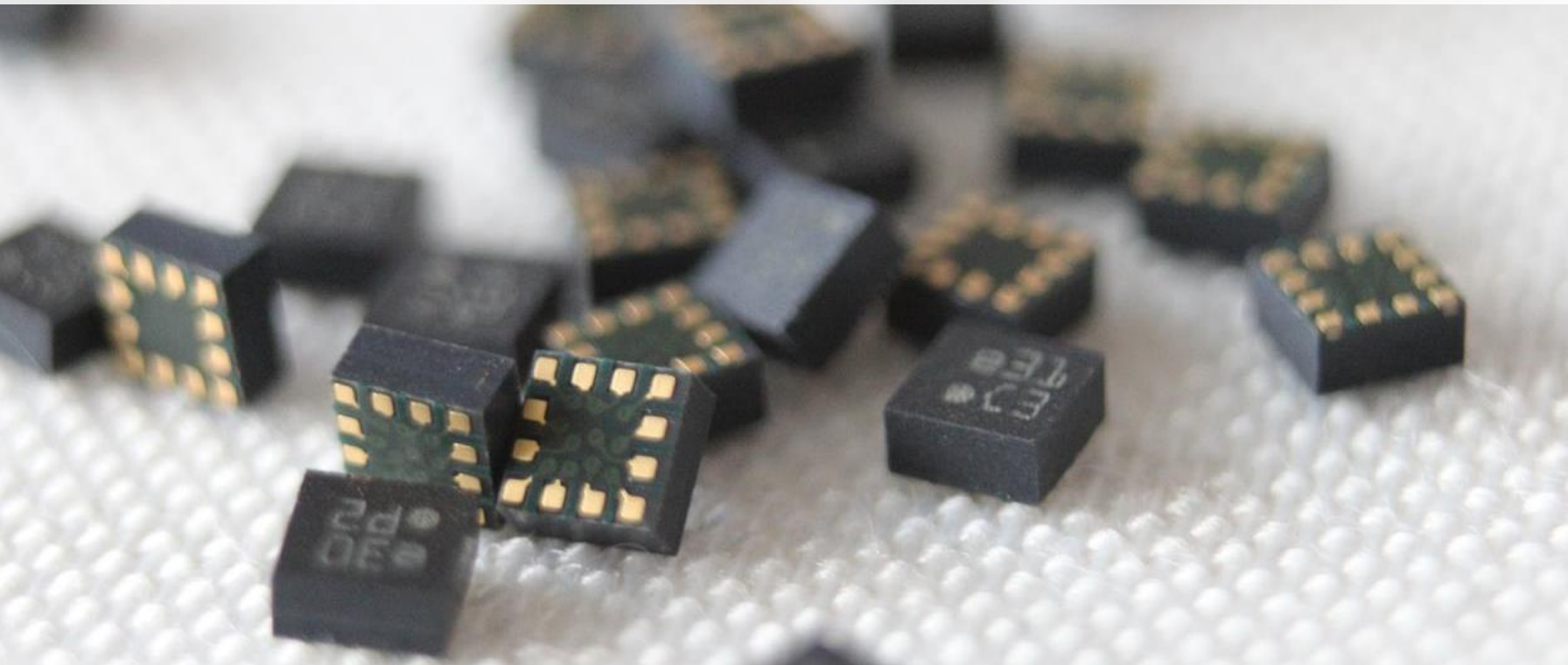


Mit "strace" zum Linux- und BASH-Versteher

Harald König - 15. Augsburger Linux-Infotag 2016



Agenda I

- BST and me

- Let's use strace
 - Erste Schritte
 - Ausgabe von strace einschränken
 - Mehrere Prozesse und Kind-Prozesse tracen
 - Time-Stamps und Performance-Betrachtungen
 - Warum bricht denn XY ab, welche Dateien sind relevant?
 - Welche Login-Skripte wurden ausgeführt
 - Noch mehr Daten sehen

Agenda II

→ Probleme

- strace stört den Prozess-Ablauf
- ptrace() ist nicht kaskadierbar
- SUID-Programme tracen
- Für alle Benutzer lesbarer Output von strace
- Dead-Locks

→ SEE ALSO

→ Conclusion

Agenda III

- strace und die Shell
 - Einfache Grundregeln
 - globbing, wildcards
 - quoting
 - Pipes

- SEE ALSO

- Conclusion



Me

- Erste „OpenSource“ in der Schule (1980, CBM-3032)
- (sup)porting TeX ab 1986 an der Universität Tübingen
- VMS (1985) und UN*X (1987) an der Uni
- Nie wieder INTEL x86 (1989/90)
- Linux 0.98.4 (1992), doch wieder INTEL;-)
- XFree86 (S3, 1993-2001)
- science + computing ag in Tübingen seit 2001
- Bosch Sensortec GmbH in Reutlingen seit 2014
- ...

Bosch – the MEMS supplier

Out of one hand MEMS sensors, actuators and solutions

Bosch Sensortec



CE

- Accelerometers
- Geomagnetic sensors
- Gyroscopes
- Pressure sensors
- Humidity sensors
- Combo sensors
- ASSNs

Akustica



CE

- MEMS microphones

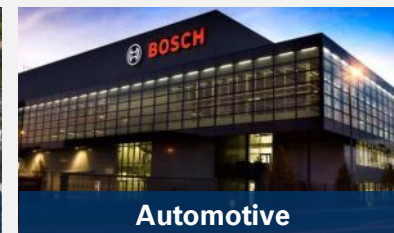
Bosch Connected Devices & Solutions



CE & Industrial

- Smart sensor/ actuator nodes
- Embedded SW & Algorithms
- Customized IoT sensor & actuator solutions

Automotive Electronics



Automotive

- Accelerometers
- Angular rate sensors
- Pressure sensors
- Mass flow sensors

Warum

```
insmod *.ko
```

```
echo "Hallo"
```

```
cat < /etc/passwd | sort
```

```
find -name *.c
```



Man Pages

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

- `man bash`

Let's use strace

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



Erste Schritte

```
$ strace emacs
$ strace -p ` pidof 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
```

....

→ Alles weitere nun live im xterm. . .
(oder im anhängenden Artikel)

Linux-Versteher dank strace

Harald König

Harald.Koenig2@Bosch-Sensortec.com

koenig@linux.de

strace ist ein wahres Wundertool in Linux, man muss es nur einsetzen - und kann damit sehr viel über die Abläufe und Interna von Linux lernen. Mit strace können einzelne oder mehrere Prozesse zur Laufzeit auf System-Call-Ebene „beobachtet“ werden. Damit lassen sich bei vielen Problemen sehr einfach wertvolle Informationen zum Debuggen gewinnen: welche Config-Dateien wurden wirklich gelesen, welches war die letzte Datei oder Shared-Library vor dem Crash usw. Im Unterschied zu interaktiven Debuggern läuft das zu testende Programm mit strace mehr oder weniger in Echtzeit ab, und man erhält schon während des Programm-Laufs jede Menge Ausgaben zu allen erfolgten Kernel-Calls, so dass man den Ablauf des Prozesses „live“ mitverfolgen bzw. den abgespeicherten Output nachträglich bequem auswerten kann.

Auch bei Performance-Problemen kann man mit strace interessante Informationen gewinnen: wie oft wird ein Syscall ausgeführt, wie lange dauern diese, wie lange „rechnet“ das Programm selbst zwischen den Kernel-Calls. Man kann auch den kompletten I/O eines Programms (Disk oder Net) mit strace recht elegant protokollieren und später offline analysieren (oder auch „replay“en, bei Bedarf sogar in „Echtzeit“ Dank präziser Time-Stamps).

Der Vortrag soll anregen, rätselhafte UNIX-Effekte, -Probleme oder Programm-Crashes neu mit strace zu betrachten und damit hoffentlich (schneller) zu Lösungen und neuen Erkenntnissen zu gelangen.

1 Let's use 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.

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:

1.1 Erste Schritte

```
$ 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 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"], [/* 132 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY) = 3
read(3, "harald.mydomain.de\n", 32768) = 19
write(1, "harald.mydomain.de\n", 19) = 19
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() oben).

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"], [/* 132 vars */]) = 0
harald.mydomain.de
$
$ strace -v -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], ["LESSKEY=/etc/lesskey.bin", "MANPATH=/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"... , "CVSROOT=/soft/.CVS", "GS_LIB=/usr/local/lib/ghostscrip"... , "TK_LIBRARY=/new/usr/lib/tk4.0", "MORE=-sl", "WINDOWID=14680077", "QTDIR=/usr/lib/qt3", "JRE_HOME=/usr/lib64/jvm/jre", "USER=harald", "XTERM_SHELL=/bin/bash", "TIMEFORMAT=\\t%R %3lR real,\\t%U use"... , "LD_LIBRARY_PATH=/usr/local/nx/3."... , "LS_COLORS=no=00:fi=00:di=01;34:l"... , "OPENWINHOME=/usr/openwin", "PILOTPORT=usb:", "XNLSPATH=/usr/share/X11 [... 22 lines deleted ...] rap"... , "_=/usr/bin/strace", "OLDPWD=/usr/local/nx/3.5.0/lib/X"...]) = 0
```

Ein besonderer Modus von `strace` bekommt man mit der Option `-c`. Hier wird nicht mehr jeder einzelne Kernel-Call ausgegeben, sondern nur ganz am Ende eine Statistik über die Anzahl der einzelnen Calls sowie die jeweils verbrauchte CPU-Zeit. Diese Ausgabe mag für einen ersten Überblick bei Performance-Problemen ganz nützlich sein, jedoch erfährt man nur wenige Details über den Programmablauf und sieht den Ablauf des Programms nicht mehr „live“ (für die Kombination von Live-Trace und abschließender Statistik gibt es dann die Option `-C`).

1.2 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.mydomain.de\n", 32768) = 19
write(1, "harald.mydomain.de\n", 19harald.mydomain.de
) = 19
read(3, "", 32768)                   = 0

$ strace -e open,read,write cat /etc/HOSTNAME > /dev/null
open("/etc/HOSTNAME", O_RDONLY)      = 3
read(3, "harald.mydomain.de\n", 32768) = 19
write(1, "harald.mydomain.de\n", 19) = 19
read(3, "", 32768)                   = 0

$ strace -e file         cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 132 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)      = 3
```

1.3 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"], [/* 132 vars */]) = 0

$ strace -e execve bash -c /bin/true
execve("/bin/bash", ["bash", "-c", "/bin/true"], [/* 132 vars */]) = 0
execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
$ strace -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/* 132 vars */]) = 0

$ strace -f -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/* 132 vars */]) = 0
execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
execve("/bin/false", ["/bin/false"], [/* 130 vars */]) = 0

$ strace -o OUTFILE -f -e execve bash -c "/bin/true ; /bin/false" ; grep execve OUTFILE
21694 execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/* 132 vars */]) = 0
21695 execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
21696 execve("/bin/false", ["/bin/false"], [/* 130 vars */]) = 0

$ strace -o OUTFILE -ff -e execve bash -c "/bin/true ; /bin/false" ; grep execve OUTFILE*
OUTFILE.22155:execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/*...*/]) = 0
OUTFILE.22156:execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
OUTFILE.22157:execve("/bin/false", ["/bin/false"], [/* 130 vars */]) = 0

```

1.4 Time-Stamps und Performance-Betrachtungen

Bei Performance-Fragen ist es oft hilfreich, dass jede Ausgabe von `strace` zu einem Kernel-Call mit einem Time-Stamp versehen werden kann. Gibt man die Option `-t` ein-, zwei- oder dreimal an, kann man verschiedene Formate des Zeit-Stempels wählen: HH:MM:SS, HH:MM:SS.Mikrosekunden, und die UNIX-Zeit in Sekunden und Mikrosekunden seit 1970 GMT/UTC. Die letzte Form ist für spätere Berechnungen mit den Time-Stamp am einfachsten, die anderen beiden Formen meist leichter lesbar und einfacher mit anderen Time-Stamp z.B. aus `syslog` oder anderen Log-Files abzugleichen, wenn es um die Analyse ganz bestimmter Events geht.

Mit der Option `-r` bekommt man wahlweise auch die Zeit-Differenz zum letzten Kernel-Call in Mikrosekunden (das kann man mit `-ttt` auch später noch selbst ausrechnen und ergänzen). Alle Time-Stamp beziehen sich immer auf den Aufruf des Kernel-Calls (Kernel Entry), man bekommt direkt keinen Time-Stamp vom Zeitpunkt des Rücksprungs vom Kernel zurück in das Programm.

Die Option `-T` fügt am Ende der Ausgabe-Zeile noch in spitzen Klammern die Verweildauer im Kernel an. Nur mit dieser Angabe kann man sich bei Bedarf den absoluten Time-Stamp der Rückkehr aus dem Kernel selbst errechnen, denn bis zum nächsten Kernel-Call könnte viel Zeit im User-Mode verbracht werden, so dass der nächste Kernel-Entry-Time-Stamp nicht immer aussagekräftig ist.

```

$ strace -t -e open,read,write,close cat /etc/HOSTNAME > /dev/null
22:22:25 open("/etc/HOSTNAME", O_RDONLY) = 3
22:22:25 read(3, "harald.mydomain.de\n", 32768) = 19
22:22:25 write(1, "harald.mydomain.de\n", 19) = 19
22:22:25 read(3, "", 32768) = 0
22:22:25 close(3) = 0

```

```
$ strace -tt -e open,read,write,close cat /etc/HOSTNAME > /dev/null
22:22:46.314380 open("/etc/HOSTNAME", O_RDONLY) = 3
22:22:46.314566 read(3, "harald.mydomain.de\n", 32768) = 19
22:22:46.314698 write(1, "harald.mydomain.de\n", 19) = 19
22:22:46.314828 read(3, "", 32768) = 0
22:22:46.314945 close(3) = 0
```

```
$ strace -ttt -e open,read,write,close cat /etc/HOSTNAME > /dev/null
1325971387.860049 open("/etc/HOSTNAME", O_RDONLY) = 3
1325971387.860143 read(3, "harald.mydomain.de\n", 32768) = 19
1325971387.860186 write(1, "harald.mydomain.de\n", 19) = 19
1325971387.860249 read(3, "", 32768) = 0
1325971387.860283 close(3) = 0
```

```
$ strace -r -e open,read,write,close cat /etc/HOSTNAME > /dev/null
0.000067 open("/etc/HOSTNAME", O_RDONLY) = 3
0.000076 read(3, "harald.mydomain.de\n", 32768) = 19
0.000042 write(1, "harald.mydomain.de\n", 19) = 19
0.000039 read(3, "", 32768) = 0
0.000033 close(3) = 0
```

```
$ strace -ttT -e open,read,write,close cat /etc/HOSTNAME > /dev/null
22:24:09.339915 open("/etc/HOSTNAME", O_RDONLY) = 3 <0.000010>
22:24:09.340001 read(3, "harald.mydomain.de\n", 32768) = 19 <0.000012>
22:24:09.340052 write(1, "harald.mydomain.de\n"
```

Meist wird man zunächst den strace-Output in eine Datei schreiben und anschließend „Offline“ verschiedene Auswertungen machen. Hier aber ein kleines „Live“-Beispiel zur Frage „was treibt denn der acroread die ganze Zeit im Hintergrund?“:

```
$ strace -p $( pgrep -f intellinux/bin/acroread ) -e gettimeofday -tt
22:48:19.122849 gettimeofday({1325972899, 122962}, {0, 0}) = 0
22:48:19.123038 gettimeofday({1325972899, 123055}, {0, 0}) = 0
22:48:19.140062 gettimeofday({1325972899, 140154}, {0, 0}) = 0
22:48:19.140347 gettimeofday({1325972899, 140482}, {0, 0}) = 0
22:48:19.142097 gettimeofday({1325972899, 142261}, {0, 0}) = 0
22:48:19.146188 gettimeofday({1325972899, 146335}, {0, 0}) = 0
22:48:19.159878 gettimeofday({1325972899, 160023}, {0, 0}) = 0
CTRL-C
$ strace -p $( pgrep -f intellinux/bin/acroread ) -e gettimeofday -t 2>&1 \
| cut -c-8 | uniq -c
136 22:48:26
138 22:48:27
140 22:48:28
CTRL-C
$ strace -p $( pgrep -f intellinux/bin/acroread ) -e gettimeofday -t 2>&1 \
| cut -c-5 | uniq -c
8309 22:50
8311 22:51
CTRL-C
```

oder als weiteres Beispiel eine kleine Statistik über 10 Sekunden:

```

$ strace -p $( pgrep -f intellinux/bin/acroread ) -c & sleep 10 ; kill %1
[1] 24495
Process 18559 attached - interrupt to quit
[ Process PID=18559 runs in 32 bit mode. ]
Process 18559 detached
System call usage summary for 32 bit mode:
$ % time      seconds  usecs/call   calls   errors syscall
-----
79.13   0.010992         5    2114         poll
 7.88   0.001094         0    2939         clock_gettime
 7.00   0.000972         0    2124        1716 read
 3.07   0.000426         0    1388         gettimeofday
 2.93   0.000407         1     406         writev
 0.00   0.000000         0         1 restart_syscall
 0.00   0.000000         0         2 time
-----
100.00   0.013891                8974    1716 total

```

1.5 Warum bricht denn XY ab, welche Dateien sind relevant?

Wenn ein Programm abbricht, ob nun mit einer „segmentation violation“ oder einer erkennbaren vernünftigen(?) Fehlermeldung, hat man schon halb gewonnen: jetzt muss man „nur“ das Programm mit strace laufen lassen, hoffen, dass damit der Fehler noch reproduzierbar ist und schon findet man kurz vor Ende der strace-Ausgabe vielleicht den ganz offensichtlichen Hinweis auf einen der letzten Schritte, der zum Problem geführt hat.

So könnte das Problem daran liegen, dass eine benötigte Shared-Library nicht gefunden wird. Hier ein Beispiel von emacs der seine libX11.so.6 nicht mehr finden kann:

```

25243 open("/usr/lib64/x86_64/libX11.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
25243 stat("/usr/lib64/x86_64", 0x7fff4f3568c0) = -1 ENOENT (No such file or directory)
25243 open("/usr/lib64/libX11.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
25243 stat("/usr/lib64", {st_mode=S_IFDIR|0755, st_size=212992, ...}) = 0
25243 writev(2, [{"usr/bin/emacs", 14}, {" ", 2}], {"error while loading shared libra"...
25243 exit_group(127) = ?

```

Oft bekommt man bei Problemen postwendend als Antwort „Wir haben seit dem letzten erfolgreichen Lauf gar nichts geändert“. Hier kann man mit strace leicht eine komplette Liste aller geöffneten oder auch gesuchten und nicht gefundenen Dateien erstellen und dann deren Time-Stamps im Filesystem überprüfen, ob denn wirklich nichts neu installiert oder geändert wurde:

```

$ strace -o OUT -e open,execve -f emacs
$ grep \" OUT | cut -d\" -f2 | sort | uniq -c | sort -nr | less
 35 /opt/kde3/share/icons/hicolor/icon-theme.cache
   9 /home/harald/.icons/DMZ/index.theme
   6 /usr/share/emacs/site-lisp/anthy/leim-list.el
   5 /usr/share/emacs/23.3/etc/images/splash.xpm
   4 /usr/share/icons/hicolor/icon-theme.cache
$ ls -ltcd $( grep \" OUT | cut -d\" -f2 | sort -u ) | less
-r--r--r--  1 root  root          0 Jan  7 23:43 /proc/filesystems
-r--r--r--  1 root  root          0 Jan  7 23:43 /proc/meminfo
drwx-----  2 harald users    28672 Jan  7 23:40 /home/harald/.emacs.d/auto-save-list/
drwxrwxrwt 48 root  root    12288 Jan  7 23:33 /tmp//

```



```
-rw-r--r-- 1 root root 3967384 Jan 7 14:29 /usr/local/share/icons/hicolor/icon-theme.cache
-rw-r--r-- 1 root root 94182076 Jan 7 11:34 /usr/share/icons/hicolor/icon-theme.cache

$ strace -o OUT -e open,execve -e signal=!all -f emacs /etc/HOSTNAME
$ cut -d\" -f2 OUT | sort -u > FILELIST
```

```
$ strace -o OUT -e open -f emacs /etc/HOSTNAME
$ grep open OUT | grep -v ENOENT | cut -d\" -f2 | sort -u > FILELIST
```

oder eine Liste aller Zugriffe, die fehlschlagen (vgl. Unterschied zwischen `-e open` und `-e file!`):

```
$ strace -o OUT -e file,execve -f emacs
$ grep ENOENT OUT | cut -d\" -f2 | sort -u
$ grep ' = -1 E' OUT | cut -d\" -f2 | sort -u
```

Wenn man eine komplette FILELIST erstellt, lässt sich leicht prüfen, welche Config-Dateien beispielsweise gelesen wurden, und kann dann nach der Datei suchen, aus der ein magischer String (Makro, Config-Setting, komischer Text in der Ausgabe o.ä.) kommt, um das Problem einkreisen zu können. Man sollte jedoch zuvor noch Einträge mit `/dev/` oder `/proc/` etc. entfernen:

```
$ strace -o OUT -e open,execve -f emacs /etc/HOSTNAME
$ grep -v ENOENT OUT | grep \" | cut -d\" -f2 | sort -u > FILELIST
$ egrep -v '^/dev|^/proc/' FILELIST | xargs grep "Welcome to GNU Emacs"
Binary file /usr/bin/emacs-gtk matches
```

1.6 Welche Login-Skripte wurden ausgeführt

...ist hiermit 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 gezeigten Beispiel bleibt die Datei FILELIST unsortiert, denn auch die Reihenfolge der Login-Skripte ist natürlich interessant:

```
$ 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 (s.u. zu Sicherheitsaspekten eines `strace`-Logs von `sshd` o.ä.):

```
$ 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 ...
```

1.7 Noch mehr Daten sehen

Bei lokalen Dateien reicht eine Datei-Liste und `grep`, um nach der Quelle eines bestimmten Strings zu suchen. Mit Netzwerkverbindungen o.ä. geht das leider nicht so einfach. Für lange Strings bei `read()`, `write()`, usw. werden per default nur die ersten 32 Zeichen ausgegeben, damit kann man im Output von `strace` nicht gut suchen. Hier ist es nützlich, die maximale Länge der Strings zu erhöhen. Oft reichen die ersten 999 Zeichen, manchmal braucht man wirklich die kompletten Records („viel hilft viel“):

```
$ strace -s999 -o OUTFILE -f wget http://www.google.de/
```

(oder gleich `-s 999999`) um nach einem bekannten String zu suchen (z.B. HTTP). Damit findet man den System-Call `recvfrom()` sowie `read()` und kann sich auf den `read()`-Call konzentrieren, wenn man nur die empfangenen Daten untersuchen will (für den gesendeten Daten-Stream entsprechend `sendto()`). Einen guten Überblick über die Netz-Verbindung liefert

```
$ strace -o OUT -e connect,bind,recvfrom,sendto,read wget http://www.google.de/
```

Um die empfangenen Daten zu extrahieren, kann man z.B. die Strings der Rückgabewerte des `recvfrom()`-Calls extrahieren, die ordentlich gequoteten Strings mit dem Shell-Kommando ausgeben und die Quotes konvertieren lassen:

```
$ strace -o OUT -s 999999 -e read wget http://www.google.de/
$ grep 'read(4, "' OUT | cut -d, -f2- | sed -e '1,/google.com/d' \
    -e s'/. [0-9]*' * = [0-9]*$/' -e 's@^@/bin/echo -en @' | sh
```

Nun stören nur noch die Längen-Records des HTTP-Protokolls, aber der interessante Inhalt ist komplett und durchsuchbar.

2 Probleme

Doch selbst ein so traumhaftes Werkzeug wie `strace` ist nicht ganz ohne Schatten. Hier ein paar Hinweise und Warnungen, den Rest erzählt ihnen die Man-Page oder `google` ;-)

2.1 strace stört den Prozess-Ablauf

Die Verwendung von `strace` ist meist transparent für den getrackten Prozess, so sollte eine „Messung“ mit `strace` das Messergebnis meist nicht verfälschen.

Jedoch sollte man immer berücksichtigen, dass die Kommunikation zwischen `strace` und den getrackten Prozess(en) zusätzlichen Rechenaufwand und viele Kontext-Wechsel bedingen. Zumindest bei Performance-Aussagen auf Systemen mit hoher Last sollte man daher besonders vorsichtig sein.

Ab und an hatte ich auch schon Hänger in den Prozessen, meist bei IPC oder mit Realtime-Signalen. „Früher“ (alte Linux-Kernel, vielleicht um 2.6.<klein> und noch älter?!) trat dies häufiger und dann leider reproduzierbar auf – inzwischen eher nicht mehr.

Sollte jedoch ein angehängter Prozess weiter hängen, obwohl `strace` abgebrochen wurde (`strace` lässt sich ab und an nur mit einem beherzten `kill -9` noch bremsen), dann kann es lohnen, den hängenden Prozessen ein `SIGCONT` zu schicken (`kill -CONT ...`), damit diese vielleicht weiterlaufen.

2.2 ptrace() ist nicht kaskadierbar

Der System-Call-Tracer `strace` verwendet, ebenso wie `gdb` oder andere Debugger, den System-Call `ptrace()`, um den zu debuggenden Prozess zu kontrollieren und von diesem Informationen zu erhalten. Da immer nur ein einziger Prozess mittels `ptrace()` einen anderen Prozess kontrollieren kann, ergibt sich automatisch die Einschränkung, dass Prozesse, welche schon mit `gdb/strace` o.ä. debuggt werden, nicht nochmals mit einem weiteren `strace`-Kommando beobachtet werden können:

```
$ strace sleep 999 &
$ strace -p $(pgrep sleep)
attach: ptrace(PTRACE_ATTACH, ...): Operation not permitted
```

2.3 SUID-Programme tracen

SUID-Programme lassen sich aus Sicherheitsgründen von normalen Benutzern in Linux/UNIX nicht mit SUID debuggen oder (s)tracen, denn damit könnte der ausführende und tracende Benutzer ja beliebig in die Programmausführung unter einer anderen UID eingreifen und z.B. Daten zu/von einem Kernel-Call ändern. SUID-Programme werden daher ohne SUID, also unter der eigenen UID ausgeführt (was im Einzelfall auch nützlich sein kann). Wird `strace` unter `root` ausgeführt, ist dies kein Problem mehr: `root` darf ohnehin alles, damit ist auch das Ausführen unter anderen Benutzer-IDs kein erweitertes Sicherheitsproblem.

Will man nun jedoch in einem komplexeren Skript, welches *nicht* unter `root` laufen soll/darf, ein einzelnes SUID-Kommando (z.B. `sudo`) starten, so kann man dies mit `strace` erreichen, wenn man das `strace`-Binary *selbst* auf SUID `root` setzt.

Aber **Vorsicht**: jedes mit diesem SUID-präparierten `strace` ausgeführten „getracete“ Programm läuft damit automatisch unter `root` – eine große Sicherheitslücke, falls andere Benutzer dieses SUID-`strace` ausführen dürfen! Daher sollte man entweder alle Zugriffsrechte für nicht-Owner entfernen (`chmod go= strace` bzw. `chmod 4100 strace`), oder diese „gefährliche“ Variante von `strace` als Kopie in einem für andere Benutzer unzugänglichen Verzeichnis „verwahren“.

BASH-Versteher dank strace

Harald König

Harald.Koenig2@Bosch-Sensortec.com

koenig@linux.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 Oberhausen
$ echo 'Hallo'        $ echo Hallo Oberhausen
$ echo "Hallo"       $ echo 'Hallo Oberhausen'
```

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 *einzel*n im exec()-Aufruf dem Programm übergeben:

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

oder schöner

```
#include <unistd.h>
int main(int arc, char *argv[])
{
    char *const argv[] = { "programm", "Hallo", "Oberhausen 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 (*) verziert. 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 gequotationet 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 „quotation“ 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-Shell, 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 „quotation“: 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 gequotationete 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 Back-quote (`) 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 Oberhausen
Hallo Oberhausen
$ \echo Hallo Oberhausen
Hallo Oberhausen
```

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

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