**30 Shell Programming DO en DON'Ts**

Wouter Liefting
liefting@nl.ibm.com

**1. Gebruik dubbele quotes bij ELKE variabele die je refereert:**

Goed: echo "$een"

Fout: echo $een

Waarom? Als een variable spaties bevat of leeg is krijg je op de meest onmogelijke plaatsen (bv. if [ $1 = "bla" ] vs. if [ "$1" = "bla"]) een syntax error.

**2. Gebruik $@ en geen $***

Waarom? $* gooit al je positionele parameters op 1 grote hoop, en da's meestal niet wat je wilt - vooral niet als sommige positionele parameters (bv. "My Documents") spaties bevatten of expres leeg zijn.

**3. Zet een default TMOUT setting in de shell profile van root**.

Waarom? Dan logt root automatisch uit na bv. 300 seconden (5 minuten) inactiviteit.

**4. Zorg dat een shell script wat door anderen gebruikt wordt, diverse shell-variabelen zet op een normale waarde**. Het gaat dan oa. om $PATH, $LANG, $LC_*.

Waarom? Diverse programma's, waaronder sort, date en dergelijke maken hier gebruik van en kunnen bij afwijkende waarde onverwachte output opleveren.

**5. Start elk shell script met #!/bin/sh of #!/bin/ksh of #!/bin/bash**, afhankelijk van of je POSIX compliant programmeert of ook bash/ksh specifieke functies gebruikt. Als je dit namelijk helemaal niet doet dan wordt het script uitgevoerd door de shell die de gebruiker "toevallig" gebruikt, en dat zou best wel eens bv. de csh kunnen zijn. Uitzondering: shell scripts die bedoeld zijn om ge"source"d te worden in je omgeving (bv. .bashrc).

**6. Trap** de signalen 1 2 3 en 15, op zijn minst, als je een script hebt wat langer dan een paar seconden draait. Zeker als je temporary files aanmaakt: zorg dan dat je ze verwijdert als onderdeel van de signal handler. Maar zorg er wel voor dat je signal handler zo robuust is dat hij niet kan "hangen", anders gooit de gebruiker er alsnog een kill -9 tegenaan.

**7. Maak temporary files aan met $$ in de bestandsnaam**. Of beter nog, gebruik **mktemp**. Want wat gebeurt er anders als jouw script door twee gebruikers

tegelijkertijd wordt opgestart?

**8. Shell script of programma killen?** Altijd eerst met kill (= signaal 15), even wachten en dan pas met signaal 9. Geef een script de kans zichzelf netjes te beeindigen! (En kende je trouwens ook het Linux killall commando al?)

**9. Schrijf je shell script in de eerste plaats voor leesbaarheid, en pas in de tweede plaats voor performance**. Als je door slim programmeren een miliseconde execution time kan uitsparen maar het kost je collega tientallen minuten om te doorgronden wat je bedacht hebt, heb je nog geen winst gemaakt.
**Maar**: Als je middenin diep geneste loops enkele miliseconden tijdwinst kan boeken kan dat wel hele grote gevolgen hebben voor de totale executie tijd. In dat geval kan het de moeite lonen om slim ipv. netjes te programmeren. Maar beter is het om ingewikkelde geneste loops helemaal niet in shell te programmeren maar bv. in zijn geheel in **awk**, **sed** of **perl**.
Oh, en als er ingewikkelde dingen gebeuren in een script, leg dat dan in je commentaar even uit in gewone-mensentaal!

**10. Shell variable namen die je zelf bedenkt doe je in kleine letters**. Dit om verwarring met de "standaard" shell variabelen (ook in de toekomst) te voorkomen. Die laatste zijn nl. allemaal in hoofdletters. En een shell variabele heeft uiteraard een naam die slaat op wat je erin opslaat, toch?

**11. Gebruik NIET het select statement om gebruikersmenus te maken, maar investeer in plaats daarvan tijd in het leren van dialog**.
Waarom? Dialog werkt full-screen, heeft veel meer mogelijkheden om input te controleren en ziet er ook veel fraaier uit. 't is wel lastiger te leren, helaas.

**12. Probeer het gebruik van elif binnenin een if-then-else-fi constructie te vermijden**. Dit maakt je programma's, en in het bijzonder de condities waaronder een bepaalde actie uitgevoerd wordt, vaak erg ondoorzichtig. Zeker als de expressies die bij de if en bij de elif geevalueerd worden ook nog eens totaal geen relatie met elkaar hebben. Het nesten van twee if-then-fi constructies is dan veel logischer en duidelijker. En als de expressies wel met elkaar te maken hebben, dan is een case-do-done statement wellicht veel beter op zijn plaats.

**13. Gebruik geen nohup om een proces in de achtergrond te forceren. Gebruik in plaats daarvan "echo command | at now".**
Waarom? nohup vangt namelijk de HUP signalen naar child processen niet per se op en dit leidt tot hele rare effecten.

**14. Gebruik het tooltje "seq" in combinatie met for loops (alleen onder Linux; AIX kent geen seq). for i in `seq 1 10`** is veel gemakkelijker dan **for i in 1 2 3 4 5 6 7 8 9 10**

**15. Gebruik "exit", "break" en "continue" in loops om extreme situaties (ihb. foutsituaties) op te vangen, in plaats van hele slimme expressies die hier rekening mee houden**. Sommige mensen vinden het gebruik van een "goto"-achtig statement (wat een exit, break of continue in feite is) een teken van slecht programmeren en slecht programmaontwerp. Maar als je het spaarzaam gebruikt en alleen als er onverwachte dingen gebeuren (foutsituaties) dan maakt het over het algemeen je programma heel veel leesbaarder.

**16. Test de return code van externe commandos die fout kunnen gaan en handel een niet-nul return code netjes af.** Die foutafhandeling wordt natuurlijk geprint naar stderr ( >&2 ) en niet naar stdout. En levert jouw eigen script een "nette" return code op als het goed of fout is gegaan, voor als anderen jouw script gaan aanroepen vanuit hun scripts?

**17. Ga slim om met debug-statements bij grote scripts**. Heel simpel: Zet een debug=1 aan het begin van je script, en pas vervolgens regelmatig [ "$debug" -eq 1 ] && echo "Debug output: Variabele bla is nu $bla" >&2 toe. Als je script goed werkt hoef je alleen maar debug=0 te zetten en al je debug output is verdwenen. Nog mooier: maak een functie "debug" die je elke keer aanroept als je een stuk debug output wil zien. Of gebruik de debug faciliteiten van bash (o.a. tracked variables).

**18. Gebruik built-in arithmetich (let, expr, $(()) ) alleen als je <u>zeker weet</u> dat je getallen (ook tussenresultaten) in een 32-bit signed integer (tussen +/- 2 miljard) passen, ook in de toekomst**. Anders gebruik je **bc**. En als je gaat delen in de shell, hou er dan rekening mee dat de tussenresultaten afgerond worden. Voer de deling daarom pas op het laatste moment uit of je krijgt onnauwkeurige resultaten.

**19. Gebruik het "logger" commando om log output te genereren, in plaats van je eigen logging te doen.** Als je netjes logt via logger komt alles bij de syslog daemon terecht en van daaruit in de files die je ingesteld hebt in /etc/syslog.conf. Ook log analyse, log rotatie en dergelijke is dan allemaal netjes geregeld.

**20a. Leer perl**. Vooral als je grote (100+ regels) shell scripts schrijft met nested loops, waarin je ingewikkelde string en file manipulaties doet, met multi-dimensionale arrays en dergelijke is het in perl ineens veel sneller en gemakkelijker,

omdat in perl alle functionaliteit in de taal zelf ingebakken zit op een consistente manier, terwijl je in de shell allerlei externe programma's moet aanroepen, die ieder hun eigen syntax gebruiken (sed, awk, tr, head, tail, sort, grep, ...)

**20b. Als je geen perl wilt leren, leer dan tenminste wel hoe je "perl -pi -e" gebruikt voor "inline editing"**. Je kan dan nl. met een sed-achtige expressie een wijziging maken *in* een file, zonder dat je deze hoeft te kopieeren naar een tijdelijke file en dan weer terug moet zetten. Voorbeeldje:
perl -pi -e 's/initdefault:3:/initdefault:5:/' /etc/inittab

**21. Scripts die je voor jezelf wilt houden gaan in $HOME/bin, scripts die voor iedereen bedoeld zijn gaan in /usr/local/bin**. Elke andere plaats levert het risico op dat ze ofwel overschreven worden bij een OS upgrade (/bin, /usr/bin) of niet in je $PATH staan.

**22a. Als gewone gebruikers een script moeten uitvoeren met root-rechten, gebruik dan sudo**. En wijzig je /etc/sudoers file met visudo ivm. locking. Je kan namelijk geen SUID of SGID bit zetten op een shell script, uit oogpunt van beveiliging.

**22b. Shell scripts die uitgevoerd worden onder sudo moet je eerst even bekijken door de ogen van een hacker.** Je geeft namelijk een gewone gebruiker voor de duur van de uitvoering van dat script root rechten, en die zou hij kunnen misbruiken als hij iets voor elkaar krijgt buiten jouw shell script om. Bijvoorbeeld:

- Als jij vanuit jouw shell script **vi** aanroept, dan kan de gebruiker in vi **:!rm -fr /*** doen. Interactieve scripts zijn eigenlijk sowieso uit den boze onder **sudo**.
- Verifieer dat de parameters die de gebruiker opgeeft inderdaad wel legaal zijn, en dat als de gebruiker bv. een filenaam opgeeft, de gebruiker inderdaad ook toegang (nodig) heeft tot deze file. (NB. Als je bv. test -r doet onder sudo dan controleer je daarmee of de file readable is voor root, niet voor de gebruiker.)
- Als jij in je shell script niet het $PATH statement controleert en "ls" aanroept als "ls" in plaats van "/bin/ls", dan kan de gebruiker zijn eigen ls-variant schrijven in $HOME/bin en $HOME/bin vooraan in $PATH zetten, met desastreuze gevolgen.
- Wat gebeurt er als de gebruiker als input " 'filenaam | rm -fr /*' " opgeeft? Heb jij alles netjes gequote-d of voer je dan als root toch stiekem het rm -fr /* commando uit?

Het is in de praktijk het makkelijkste je scripts onder sudo te beperken tot niet-interactieve scripts die bovendien geen parameters of stdin accepteren.

**23. Programmeer voor nul, een of oneindig parameters**. Probeer willekeurige

limieten te vermijden: er is altijd een gebruiker die net 1 parameter meer opgeeft dan jij had voorzien. Een goede programmeur programmeert voor nul, een of oneindig parameters, en maakt daarom netjes gebruik van while loops, shift, getopts en dergelijke.

**24. Foute opties en argumenten? Druk een beschaafd "usage" statement af.**

**25. Gebruik xargs voor ingewikkelde command interpolation constructies.**
**Dus niet:** ingewikkeld_commando_a `ingewikkeld_commando_b`
**Maar:** ingewikkeld_commando_b | xargs ingewikkeld_commando_a
Want: Veel simpler op te zetten en te debuggen. En als je netjes met de opties omgaat (bv. find . -print0 | xargs -0) heb je nog veel meer controle ook. Zeker als de uitvoer van het ene commando bv. filenamen met spaties blijkt te bevatten.

**26a. Schrijf je een serie shell scripts die allemaal met elkaar te maken hebben?** Maak dan 1 shell script "common_procedures" met daarin diverse functies die in meerdere scripts voor komen, en "source" (met het . commando) dit shell script in al je andere shell scripts. Scheelt een hoop kopieerwerk en een eventuele fout hoef je maar op 1 plek te herstellen.

**26b. Schrijf je een aantal shell scripts die precies het tegenovergestelde doen van elkaar, of iets wat heel sterk op elkaar lijkt?** Overweeg dan 1 script te maken wat afhankelijk van de waarde van $0 verschillende acties uitvoert, en maak hard links naar dit script, zodat het onder de verschillende namen bekend is.

**27. Verberg geen constanten (getallen, filenamen, ...) in je script**. In plaats daarvan: definieer aan het begin van je script een serie variabelen die de constanten bevatten, en gebruik in je script die variabelen. Dit maakt achteraf wijzigen van de constanten veel gemakkelijker. Nog mooier is de constanten in een aparte configuratie-file te zetten en deze in te lezen met het "source" of "." commando.

**28. Als een bepaald onderdeel van een shell script langer dan een paar seconden kan duren, waarschuw dan van tevoren de gebruiker zodat hij/zij niet ongeduldig wordt.** Of improviseer een soort van progress bar/counter oid.

**29a. Gebruik aliases voor interactieve shells en functies voor in shell scripts.**
**29b. Gebruik de notatie "function bla" in plaats van "bla()" om een functie te definieren.**
Waarom? Functies, mits gedefinieerd als "function bla", houden vrijwel automatisch al hun eigen variabelen "binnenboord" - zeker als je ze als lokaal declareert met

typeset, declare of local. Da's precies wat je wilt in een shell script. Maar voor interactief gebruik is een alias makkelijker omdat je dan toch vrijwel niet met variabelen werkt, en een alias in 1 regel te definieren valt.

**30. Gebruik set -- $var als $var spaties bevat en je de individuele waarden wilt hebben, in plaats van gedrochten als "var1=`echo $var | awk '{print $1}'`".**
Bijvoorbeeld:

```
var="bla blabla blablabla"
set -- $var
echo $0          # bla
echo $1          # blabla
echo $2          # blablabla
```

Let wel op dat de originele $1, $* en $@ hiermee overschreven worden!

**En tenslotte**: Shell scripts zijn vaak aan verandering onderhevig. Als je meteen aan het begin van je shell scripting carriere investeert in het leren en opzetten van een versie/revisie control system (bv. **rcs**, **cvs** of **subversion**) dan kan dat in de toekomst heel veel vruchten afwerpen.

**Lijst van handige O'Reilly boeken**

Shell Scripting ("The Turtle Book") - ISBN 0-596-00595-4
Sed & Awk - ISBN 1-56592-225-5
Bash - ISBN 0-596-00965-8
Ksh - ISBN 0-595-00195-9
Learning Perl - ISBN 0-596-10105-8
Programming Perl ("The Camel Book") - ISBN 0-596-00027-8
UNIX Power Tools ("The Screwdriver Book") - ISBN 0-596-00330-7