– Lösung zur Praktikumsaufgabe 5 –

Thema: *Resultate und Funktionen in der Shell*

1.

Listing 1: Lösung von Aufgabe 1.a)

a)
```
#!/bin/bash
if [ $# -ne 1 ]; then
    echo "Usage: $0 <Kommando>"
    exit 1
fi
$1
echo $1 returns $?
exit 0
```

Da das Skript nur einen einzigen Parameter übernimmt, muss das auszuführende Kommando ggf. in Anführungszeichen gesetzt werden (, um damit als *ein* Parameter übergeben zu werden). Kommandos, die fehlschlagen, sind beispielsweise:

```
~> ./aufgabe-05-01a.sh "less /etc/shadow"
/etc/shadow: Keine Berechtigung
less /etc/shadow returns 1
~> ./aufgabe-05-01a.sh "ls arglbargl"
ls: arglbargl: Datei oder Verzeichnis nicht gefunden
ls arglbargl returns 2
```

b) Der einfachste Weg zur Übergabe aller Kommandozeilenparameter auf einmal ist der Shell-Ausdruck `$*`. Zitat `man bash`

„$* expands to the positional parameters, starting from one.“

Der Ausdruck repräsentiert die *gesamte* Kommandozeile. Damit vereinfacht sich unser Skript zu:

Listing 2: Lösung von Aufgabe 1.b)

```
#!/bin/bash
$*
echo "$*" returns $?
exit 0
```

Es gibt jedoch noch einen anderen Weg, den so genannten *Shift*-Operator. Wenn dieser aufgerufen wird, dann findet eine Neunumerierung der Kommandozeilenparameter statt: `$0` geht verloren, `$1` wird zu `$0`, `$2` wird zu `$1` ... `$n` wird zu `$n-1`.

Mittels dieses Operators können wir iterativ die Kommandozeile des eigenen Aufrufs generieren:

Listing 3: Lösung der Aufgabe 1.b) mittels *Shift*-Operator

```
#!/bin/bash
```

```
params=$#
cmdline=""

# write params into cmdline, one after another
for (( c=0 ; $c < $params ; c++ ))
do
  cmdline="$cmdline $1"
  shift
done

# call cmdline and print its return value
$cmdline
echo $cmdline returns $?

# epilogue
exit 0
```

Wir müssen nur die originale Anzahl Kommandozeilenparameter in der Variablen `params` aufheben, weil sich `$#` bei jedem Shift-Aufruf um eins verringert.

2. Das Shellskript hat folgende Gestalt:

Listing 4: Lösung der Aufgabe 2

```
#!/bin/bash
rndvalue=$RANDOM
echo generated $rndvalue
exit $rndvalue
```

Drei Aufrufe könnte beispielsweise folgendermaßen aussehen:

```
~> ./aufgabe-05-01b.sh ./aufgabe-05-02.sh
generated 1122
./aufgabe-05-02.sh returns 98
~> ./aufgabe-05-01b.sh ./aufgabe-05-02.sh
generated 19777
./aufgabe-05-02.sh returns 65
~> ./aufgabe-05-01b.sh ./aufgabe-05-02.sh
generated 4756
./aufgabe-05-02.sh returns 148
```

Nicht weiter überraschend gilt

$$\begin{aligned} 1122 \mod 256 &= 98, \\ 19777 \mod 256 &= 65, \text{ und} \\ 4756 \mod 256 &= 148. \end{aligned}$$

Übersteigt der Rückgabewert die Größe eines Bytes, so werden die höherwertigen Bits offensichtlich einfach abgeschnitten.

3.

Listing 5: Lösung der Aufgabe 3

```
#!/bin/bash
user=robge
host=rob.rz.htw-dresden.de
path=\~/tmp

# securely copies argument to $user@$host:$path
# returns
#   0, if copy successful
#   1, if not
copyfile()
{
    scp $1 $user@$host:$path 2>/dev/null
    if [ $? -eq 0 ]; then
  return 0
    else
  return 1
    fi
}

# some sanity checks
if [ $# -eq 0 ]; then
    echo Usage: $0 \<directory\>
    exit 1
fi
if [ ! -d $1 ]; then
    echo $1 is no directory
    exit 1
fi

# main
for file in $1/*
do
  copyfile $file
  if [ $? -eq 0 ]; then
      echo copying $file successful
  else
      echo copying $file failed
  fi
done

#epilogue
exit 0
```

Erklärungsbedürftig ist das `for`-Konstrukt zu Beginn der eigentlichen Hauptroutine („main“): Es expandiert den Ausdruck nach `in` und weist der Variablen `file` in jeder Iteration genau ein Match (einen Dateinamen) daraus zu, und zwar solange, bis alle Dateinamen einmal zugewiesen wurden. `$1` ist der als Kommandozeilenparameter übergebene Name eines Verzeichnisses, und `*` selektiert alle Dateien (mit Ausnahme derjenigen, deren Name mit einem Punkt beginnt).

4. Einige Gedanken:

- Es handelt sich um eine sehr elegant und kurz formulierte fork()-Bombe, kodiert in Bash.
- Die öffnenden und schließenden Klammern `()` zeigen an, dass es sich um eine Funktionsdefinition, der Funktionsname ist offenbar `:`.
- Durch Semikolon getrennt wird die zunächst definierte Funktion aufgerufen.
- Man könnte also genausogut schreiben: `func(){ func | func & }; func`.
- Die Funktion ruft zwei Instanzen von sich wieder auf, einmal im Vordergrund und einmal im Hintergrund; beide Instanzen sind über eine Pipe miteinander verbunden.
- Jede aufgerufene Instanz wird wiederum in einer *bash* ausgeführt; einmal im Vordergrund und einmal im Hintergrund.
- Wichtig ist, dass pro Aufruf zwei Shells gestartet werden. Mittels `func(){ func & }; func` ist dies nicht der Fall; es wird für pro Aufruf genau eine Shell gestartet; die Rate der neugestarteten entspricht der Rate der beendeten Shells (, was sich mit `top` ganz gut beobachten lässt).
- Noch schneller bleibt der Rechner stehen, wenn bei jedem Aufruf drei neue Funktionen gestartet werden, etwa `func(){ func | func | func & }; func`
- Es handelt sich gewissermaßen um ein Kunstwerk. Der Autor, ein gewisser „Jaromil“ äußert sich hier: *http://www.digitalcraft.org/index.php?artikel_id=278*
- Mehr Informationen finden Sie hier: *http://www.runme.org/project/+forkbombsh/*

Interessant ist, wie man sich gegen solcherart Angriffe schützen kann; dies geschieht mit dem Bash-Befehl `ulimit`[1] (hier mit dem Switch `-u`), mit dessen Hilfe der Administrator verschiedene Ressourcen pro Nutzer beschränken kann, z. B. die maximale Größe des ausfassbaren Speichers, die maximale Anzahl eröffneter Dateien.

[1] Nicht zu verwechseln mit dem externen Kommando `ulimit`, das die maximale Größe geschriebener Dateien limitiert.