Studienprogramm Sachbearbeiter:in Digitale Forensik
Praktikum Digitale Forensik Grundlagen
Dozenten: Tim Wetterau | Leander Hossfeld
wetterau@hs-mittweida.de | hossfeld@hs-mittweida.de
Stand: 25.03.2024

HOCHSCHULE MITTWEIDA
University of Applied Sciences

Digitale Forensik

Teil 4 – Shell-Skripting

In diesem Praktikum sollen die Grundlagen der Shell-Programmierung an drei Aufgaben verdeutlicht werden. Stellen Sie sicher, dass Sie die grundlegenden Kommandos für die Kommando-basierte Anwendung von Linux beherrschen. Die Voraussetzung für ein erfolgreiches Praktikum ist die Ausführung der Praktikumsaufgaben in der Bourne Again Shell (BASH). Aus der Vorlesung sollte bekannt sein, dass es in dieser Shell zu Problemen mit deutschen Umlauten kommen kann. Daher sind diese zu vermeiden. Ein Shell-Skript beschreibt den Ablauf bestimmter Kommandos. Daher eignet sich ein Skript sehr gut zu forensischen Analysen, bei denen bestimmte Arbeitsschritte in gleicher Art mehrmals ausgeführt werden müssen. Beispielsweise, dass Auslesen von registrierten Nutzern, Programmversionen oder der Sicherung spezieller Dateien. Durch das Skript werden diese nur einmal in Textform in einer Datei gespeichert und können anschließend durch das Aufrufen der Datei ausgeführt werden.

Vorbereitung

Zur Durchführung des Praktikums benötigen Sie die Kenntnisse aus den Praktika 1 bis 3 sowie eine funktionierende Linux Mint-VM. Stellen Sie zunächst sicher, dass diese startet und Sie ein Terminal geöffnet haben.

Grundlagen Programmstrukturen

Im letzten Praktikum soll es um Programmstrukturen gehen. Sie sind essenziell zur bedingungsgesteuerten Automatisierung von repetitiven Arbeitsschritten. Die Programmstrukturen im Linux Bash-Skripting gleichen rein strukturell denen anderer Programmiersprachen, wie Java oder C. Die grundlegenden Strukturen sollen in den nächsten Kapiteln etwas näher beleuchtet werden.

Variablen

Variablen sind wichtig zur Zwischenspeicherung von Werten, die verarbeitet werden sollen und mehrmals verwendet werden. Dazu nutzen wir die Syntax:

```
$ <Variablenname>=<Zahl>
$ <Variablenname>='<String>'
```

Damit haben wir den Variablennamen mit einem Wert belegt. Wollen wir diesen Wert wieder abrufen, nutzen wir das **$**-Zeichen und den Namen der Variable, wie in folgender Syntax:

```
$ echo $<Variablenname>
```

Ebenso können Variablen das Ergebnis von anderen Befehlen zugewiesen bekommen. Dazu nutzen zu dem **$** noch runde Klammern, wie in folgendem Beispiel:

```
$ datum=$(date)   # zuweisen des Datums zur Variable datum
$ echo $datum     # ausgaben des Wertes von datum auf derm Terminal
```

Für die Berechnung von Zahlen verwenden wir den Befehl **expr** mit der folgenden Syntax:

```
$ expr <Berechnung>
$ expr 5 + 5                    # Beispielrechnung
$ 10
```

*Berechnen Sie wie viele Tage ein Monat haben müsste unter der Annahme, dass ein Jahr aus **365** Tagen und **12** Monaten besteht. Speichern Sie das Ergebnis in der Variable* **`month`***. Schreiben Sie einen **einzeiligen Befehl**, der diese Aufgabe erfüllt. **Zusatz:** Schreiben Sie die Variable noch in eine Datei month.txt. Beide Befehle sollen zusammen auf eine Zeile geschrieben werden. Nutzen Sie Operatoren.*

Verzweigungen

Verzweigungen dienen dazu einen bedingungsgesteuerten Programmablauf zu realisieren. Z.B. Zur Reaktion auf Benutzereingaben oder die Übergabe von Flags an das Skript selbst. Wir prüfen dazu grundlegend immer einen Sachverhalt gegenüber einer Bedingung, die der Sachverhalt erfüllen muss. Also wenn **`x`**, dann **`y`**, sonst **`z`**. Die einfachste Möglichkeit dazu bietet die IF-Verzweigung. Sie unterliegt in der bash der folgenden Syntax:

```
# if [ <Bedinungsprüfung1> ]
# then
#   <Ausführungsblock1>
# elif [ <Bedinungsprüfung2> ]
# then
#   <Ausführungsblock2>
# else
#   <ELSE-Ausführungsblock>
# fi
```

Die Syntax ist so zu verstehen, dass wir in der Klammer nach **`IF`** eine Prüfung auf die gegebene Bedingung durchführen. Wird diese erfüllt, gehen wir in den Ausführungsblock1. Das folgende **`ELIF`** bedeutet „**`else if`**“, also übersetzt „ansonsten, wenn“. Es wird nur dann geprüft, wenn die **`Bedinungsprüfung1`** nicht erfüllt wurde. Wenn auch die **`Bedingungsprüfung2`** nicht erfüllt wurde, dann wird unweigerlich der **`ELSE`**-Block ausgeführt. In **`ELSE`** stehen Bedingungen, die ausgeführt werden, wenn keine der vorherigen Bedingungsprüfungen erfüllt wird. Abgeschlossen wird der **`IF`**-Block syntaktisch mit dem Wörtchen **`FI`** also einem **`IF`** rückwärts. Die Anzahl der IF-Blöcke lässt sich beliebig erweitern und **`ELSE`**-Blöcke sind nicht zwingend erforderlich, was aber je nach Anwendungsfall vom Programmierer entschieden werden muss.

Für die Bedingungsprüfungen gibt es Bedingungsausdücke, die sich auch nach Anwendungsfall unterscheiden, z.B. ob wir auf Zahlen, Wörter oder Dateisystempfade prüfen. Einige wichtige sind in **Tabelle 4** aufgezeigt.

Tabelle 1: Bedingungsausdrücke für das Bash-Skripting

Ausdruck	Beschreibung	Eselsbrücke
Pfade		
`-d`	Angegebener Pfad ist ein Verzeichnis	directory
`-f`	Angegebener Pfad ist eine Datei	file
`-r`	Angegebener Pfad ist lesbar	readable
`-w`	Angegebener Pfad ist schreibbar	writable
Strings		
`-z`	Länge des Strings ist Null	zero
`-n`	Länge des Strings ist > 0	not zero
`==`	String1 ist gleich String2	
`!=`	String1 ist ungleich String2	
Mathematische Ausdrücke		
`-eq`	Zahl1 ist gleich Zahl2	equals
`-ne`	Zahl1 ist ungleich Zahl2	not equals
`-gt`	Zahl1 ist größer als Zahl2	greater than
`-ge`	Zahl1 ist größer gleich Zahl2	greater equals
`-lt`	Zahl1 ist kleiner als Zahl2	less than
`-le`	Zahl1 ist kleiner gleich Zahl2	less equals

Beispiel: Wir wollen ein Skript schreiben, welches prüft, ob Freitag ist und eine entsprechende Zeile ausgibt 😊

```
#!/bin/bash                                                    # Erklärung, dass Skript durch bash interpretiert, wird

tag=$(date +"%A")                                              # Zuweisen von tag, +"%A" gibt Wochentag zurück
if [ $tag == „Freitag“ ]                                       # Prüfen, ob tag gleich „Freitag“ ist
then                                                           # wenn ja, dann
      echo „Juhu es ist Freitag, Morgen ist Wochenende“        # Ausgeben dieses Textes
elif [ $tag == „Samstag“ || $tag == „Sonntag“ ]                # sonst, wenn tag gleich Samstag ODER Sonntag
then                                                           # dann
      echo „Es ist schon Wochenende“                           # diesen Text ausgeben
else                                                           # wenn nichts davon erfüllt, dann
      echo „Es ist noch kein Wochenende in sicht. Arbeite weiter …“   # diesen Text ausgeben
fi                                                             # Ende If-Verzweigung
```

Lesen Sie sich die Kommentare, also alles, was hinter den „**#**“ steht, durch und versuchen Sie zu verstehen, wie das Skript arbeitet. Übernehmen Sie das Skript in eine neue Datei mit **nano**. Sie können dazu die gemeinsame Zwischenablage von Gast- und Hostsystem verwenden. Einfach kopieren und in **nano** mit der Tastenkombination **Strg + Shift + V** einfügen. Speichern Sie die Datei und verlassen Sie **nano** wieder. Machen Sie das Skript ausführbar (Zugriffsrechte ändern) und führen Sie es anschließend mit dem folgenden Befehl aus und schauen Sie, was auf dem Terminal ausgegeben wird.

```
$ ./<Skriptname>
```

Alternativ können Verzweigungen auch mit einer **case**-Verzweigung realisiert werden. Das geht aber an der Stelle zu tief in die Programmierung hinein und wird daher nicht weiter beleuchtet. Belesen Sie sich bitte selbst über dessen Anwendung bei Interesse. Dazu sei Ihnen folgende Quelle gegeben:

- Bash – Arbeiten und programmieren mit der Shell – opensuse.org

Schleifen

Ein weiteres Element der bedingungsgesteuerten Programmierung ist die Schleife. Also das Wiederholen einer Aktion, bis eine Bedingung erfüllt oder nicht mehr erfüllt, je nach Anwendung. Dabei unterscheiden wir zwischen folgenden Schleifenarten in der Bash:

- **while**
- **until**
- **for**

Wir bitten Sie sich aufgrund des beschränkten Rahmens mit den **while** und **until**-Schleifen selbst in der gegebenen Quelle zu beschäftigen. Die **for**-Schleife soll aber näher beschrieben werden, da Sie diese brauchen, um die folgenden Aufgaben zu bearbeiten. Sie ist dazu da eine Folge von Befehlen mit einer vorher bekannten Anzahl zu wiederholen. Das kann sinnvoll sein, aber je nach Anwendungsfall auch nicht. Sehen wir uns die Syntax der **for**-Schleife an:

```
# For <zähler> in {<min>..<max>}       # <min> und <max> sind natürliche Zahlen.
# Do
#   <Ausführungsblock>
# Done
```

Achten Sie beim Setzen des Rahmens in den geschwungenen Klammern auf die ZWEI Punkte dazwischen ohne Leerzeichen. Im Ausführungsblock können wieder beliebige Befehle ausgeführt werden, welche nacheinander ausgeführt werden und wiederholt werden sollen.

Beispiel: Wir wollen ein Skript, welches im Terminal von 0 bis 10 zählt.

```
#!/bin/bash           # immer angeben

for i in {0..10}      # für i im Bereich von 0 bis 10
do                    # mach
      echo $i         # i auf der Kommandozeile ausgeben
done                  # Ende wenn i = 10 ist, ansonsten wieder von vorn und i = i + 1
```

Die Grenzen sind uns vorher bekannt und können daher problemlos in den geschwungenen Klammern gesetzt werden. Es ist zu beachten, dass wir i als unsere Zählvariable nutzen. Bei jedem Durchlauf der Schleife wird i AUTOMATISCH inkrementiert, also um 1 erhöht. Und per echo geben wir den aktuellen Zustand von i aus.

Übernehmen Sie auch dieses Skript in Ihre VM und testen Sie. Es steht Ihnen frei, die beiden Skripts auch ein wenig zu manipulieren und zu schauen, was Ihre Änderungen bewirken. Probieren ist besser als Studieren, fürchten Sie sich nicht vor Fehlern.

Erstellung eines Shell-Skripts

Öffnen Sie ein Terminal und schauen Sie sich die Ausgabe der folgenden Befehle an:

```
$ uname -a
$ cat /etc/passwd
$ cat /etc/group
$ cat /proc/cpuinfo
$ df -h
```

Diese Befehle sollten Ihnen bekannt sein und beispielhaft zeigen, welche Standardinformationen aus einem Linux-System gewonnen werden können. Da wir diese Informationen immer von einem System sichern wollen, bauen wir uns daraus ein Skript zur Arbeitserleichterung in der Zukunft. Wir benutzten zum Bearbeiten den nano-Editor. Legen Sie das Skript an mit:

```
$ nano basic_info.sh
```

Die Informationen speichern wir in entsprechenden Dateien in einem neuen Unterordner ab. Dazu beginnen wir das Skript mit:

```
$ mkdir main_info
$ cd main_info
```

Nun speichern wir die Informationen aus den oben getesteten Befehlen in dem soeben erstellten Ordner ab. Dazu ergänzen wir die folgenden Befehle:

```
$ uname -a > os_info
$ cp /etc/passwd passwd
$ cp /etc/group group
$ cat /proc/cpuinfo > cpu_info
$ df -h > storage_info
```

Speichern Sie die Datei mit **Strg+o** und beenden Sie den Editor mit **Strg+x**. Starten Sie anschließend das Skript mit:

```
$ bash ./basic_info.sh
```

Gerne können Sie nun in den Ordner „**main_info**“ wechseln und sich die erzeugten Dateien anschauen. Um das Betriebssystem zu informieren, dass es sich bei dieser Datei um ein ausführbares Programm handelt, ändern wir die Rechte. Dafür geben Sie folgenden Befehl ein:

```
$ chmod u+x basic_info.sh
```

Anschließend können wir in der ersten Zeile im Skript angeben, mit welchen Interpreter das Skript auszuführen ist. Öffnen Sie dafür das Skript erneut mit:

```
$ nano basic_info.sh
```

und ergänzen Sie in der ersten Zeile:

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

Nun können Sie das Skript starten mit:

```
$ ./basic_info.sh
```

Sie merken, dass das Skript meckert, da der Ordner „**main_info**“ bereits existiert. Damit das Erstellen des Ordners nur ausgeführt wird, wenn dieser **nicht existiert**, prüfen wir dessen Existenz mit einer Abfrage. Schreiben Sie folgende Abfrage unter die erste Zeile:

```
if [ ! -d main_info ]
then
        mkdir main_info
        cd main_info
        uname -a > os_info
        cp /etc/passwd passwd
        cp /etc/group group
        cat /proc/cpuinfo > cpu_info
        df -h > storage_info
        exit 0
fi
```

Derzeit werden Daten in dem Ordner **main_info** bei jeder Ausführung überschrieben. Das ist aus forensischer Betrachtung eine Katastrophe. Es könnten Spuren zerstört werden, falls es auf dem untersuchten System ein Ordner **main_info** gäbe. Sehr unwahrscheinlich, aber nicht auszuschließen. Also soll unser Skript eine Warnung ausgeben, wenn der Ordner bereits existiert und anschließend mit einem Fehlerindikator sich beenden. Lesen Sie mit:

```
$ echo $?
```

den Status der letzten Programmbeendigung aus. Unter Linux beenden Programme standardmäßig mit 0. Jede andere Zahl ist ein Fehlercode. Damit sich unser Skript mit einem Fehler beendet, passen wir den Else-Zweig der If-Verzeigung entsprechend an:

```
else
        echo "Der Ordner existiert bereits." >&2
        echo "Beende ohne Änderung" >&2
        exit 1
fi
```

Speichern und führen Sie das Skript erneut aus, um das fehlerhafte Beenden zu überprüfen. Löschen Sie anschließend den Ordner **main_info** mitsamt dem Inhalt:

```
$ rm -r main_info
```

Führen Sie das Skript erneut aus. Der Ordner wird erstellt und das Skript beendet sich korrekt. Mit der folgenden Funktion können wir das Skript durch weitere Skripte ausführen lassen, falls beispielsweise eine Festplatte an unser Forensik-System angeschlossen wird.

Die Ausgabe unter Linux unterteilt sich in verschiedene Teilausgaben. Die Standardausgabe ist gedacht für eine normale Ausgabe an den Nutzer. Die Fehlerausgabe ist, wie der Name beschreibt, für Error gedacht. Im Moment nutzt unser Skript die Standardausgabe für die Error. Ändern wir dies nun:

```
echo „Der Ordner main_info existiert bereits!“ >&2
echo „Beende ohne Änderungen!“ >&2
```

Das Kürzel **&1** ist die Standardausgabe, **&2** die Fehlerausgabe. Wenn wir nun das Skript mit:

```
$ ./basic_info.sh > /dev/null
```

aufrufen, werden Standardausgaben ignoriert, jedoch Fehlerausgaben weiterhin ausgegeben. Umgekehrt hat:

```
$ ./basic_info.sh 2> /dev/null
```

den gegenteiligen Effekt. Fehlerausgaben werden ignoriert und Standardausgaben ausgegeben.

Shell-Skript verstehen

Sie haben bei der Untersuchung eines beschlagnahmten Datenträgers das folgende Shell-Skript gefunden. Die Ausführung fremden Programmcodes kann zur Infektion Ihres Systems führen. Daher beschließen Sie, das Skript zu untersuchen, indem Sie dessen Funktionsweise anhand des Quellcodes analysieren. Wozu dient das folgende Skript? Gibt es ein Hinweis auf betriebenes illegales Glücksspiel?

```
#!/bin/bash
zahl1=$((RANDOM % 20 +1))
count=0
while [ $count -lt 3 ];
do
        let count=$count+1;
        echo -e "\e[1;31m$count. Versuch""\e[0m"
        read -p "Welche Zahl wird gesucht? : " zahl2
        if [ "$zahl1" -eq "$zahl2" ]
        then
                echo -e "\e[1;31mDein Tipp ist richtig!""\e[0m"
                break
        elif [ "$zahl1" -lt "$zahl2" ]
        then
                echo -e "\e[1;31mDein Tipp ($zahl2) ist groesser als die gesuchte Zahl"
                echo -e "\e[1;31mVersuch es nochmal : ""\e[0m"
        elif [ "$zahl1" -gt "$zahl2" ]
        then
                echo -e "\e[1;31mDein Tipp ($zahl2) ist kleiner als die gesuchte Zahl"
        echo -e "\e[1;31mVersuch es nochmal : ""\e[0m"
        fi
done
if [ "$zahl1" -ne "$zahl2" ]
then
        echo -e "\e[1;31mDein Tipp ist falsch! Die gesuchte Zahl war ($zahl1)" !"\e[0m"
fi
exit 0
```

Zusatz: Shell-Skript lottozahl

In der letzten Aufgabe sollen Sie selbstständig ein ausführbares Skript ***lottozahl*** entwerfen, welches 6 zufällige Zahlen (zwischen 1 und 49) festlegt und anschließend auf der Kommandozeile anzeigt. Beginnen Sie zunächst mit dieser Aufgabe.

Tipp: *Beginnen Sie zunächst zu überlegen, wie eine Zufallszahl im Bereich von 1 bis 49 gebildet werden kann. Anschließend gilt es zu überlegen, wie wir statt einer, sechs Zahlen ausgeben können. Denken Sie auch an die nötigen Rechte und Angaben, die das Skript besitzen muss, um ausgeführt werden zu können.*

Sorgen Sie mit einer Ergänzung nun dafür, dass die Zahlen sortiert ausgegeben werden!

Fügen Sie als letzte Ergänzung folgende Eigenschaft hinzu: Wenn das Skript mit dem Parameter **-s** aufgerufen wird, soll eine zusätzliche Superzahl im Zahlenbereich von 9 bis 39 ausgegeben werden.