Let's make better* scripts

* Improved readability, increased fault-tolerance, and more security

Michael Boelen
michael.boelen@cisofy.com
NLUUG, November 2019
Before we begin...
Topics (blue pill)

- Why Shell Scripting?
- Challenges
- Reliability
- Style
- Tools
- Tips and Tricks
Topics (red pill)

- When shell (and why not)
- Common mistakes
- More reliable scripts
- and readable...
- Tools for the lazy
- Tips and tricks (no time for that, homework)
Michael Boelen

- **Open Source since 2003**
  - Lynis, Rootkit Hunter

- **Business**
  - Founder of CISOfy

- **Other**
  - Blogger at linux-audit.com
  - Content creator at linuxsecurity.expert
Let’s do this together

Assumptions
You do Dev || Ops
Linux, BSD, macOS,
Created a script before

Questions
During, at the end, and after the talk

Input welcome
Alternatives, feedback

Share
@mboelen @nluug nluug
Lynis

- Security: system auditing tool
- 2007
- GPLv3
- 25000+ lines of code
- POSIX
- #!/bin/sh
My goals for today

1. Share my knowledge
2. Learn from yours
3. Improve your project (or mine)
Why Shell Scripting?
Why?

- Powerful
- Quick
- Low on dependencies
What?

Shell scripts = glue
Potential

Small scripts can grow...

… and become an open source project!
Why not?
Challenges and Common Mistakes
Challenge 1: `#!/bin/?`

<table>
<thead>
<tr>
<th>Shell</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh</td>
<td>Portable</td>
<td>Not all features available</td>
</tr>
<tr>
<td>bash</td>
<td>Features</td>
<td>Not default on non-Linux</td>
</tr>
<tr>
<td>ash/dash</td>
<td>Portable and fast</td>
<td>Some features missing</td>
</tr>
<tr>
<td>ksh</td>
<td>Features and fast</td>
<td>Not default on Linux</td>
</tr>
<tr>
<td>zsh</td>
<td>Features</td>
<td>Not default</td>
</tr>
</tbody>
</table>
Challenge 1: #!/bin/?

Portable sh
Your company only bash
For yourself pick something

Tip: use #!/usr/bin/env bash
Challenge 2: Readability

1. `#!/bin/sh`
2. `var_with_value="red"`
3. `: ${var_with_value:="blue"}`
4. `echo "${var_with_value}"`

Red or Blue?
Challenge 2: Readability

: ${var_with_value:="blue"}

Assign a value when being empty or unset
Challenge 3: The Unexpected

#!/bin/sh
filename="test me.txt"
if [ $filename = "test me.txt" ]; then
    echo "Filename is correct"
fi

3: [: test: unexpected operator
You VS Script
Find the flaw (1)

```bash
#!/bin/sh
chroot=$1
rm -rf $chroot/usr/lib/ssl
```
Find the flaw (1)

1  #!/bin/sh
2  chroot=$1
3  rm -rf $chroot/usr/lib/ssl
You VS Script
1 - 0
Find the flaw (2)

cat /etc/passwd | grep michael

**Goal**: retrieve details for user ‘michael’
Find the flaw (2)

cat /etc/passwd | grep michael

Better:
grep michael /etc/passwd
grep "^michael:" /etc/passwd
awk -F: '{if($1=="michael") print}' /etc/passwd
getent passwd michael
You VS Script
2 - 0
Find the flaw (2)

1 if [-d $i]
2 then
3 echo "$i is a directory! Yay!"
4 else
5 echo "$i is not a directory!"
6 fi
Find the flaw (2)

```bash
if [ -d $i ]
  then
    echo "\$i is a directory!"
  else
    echo "\$i is not a directory!"
fi
```
Style
Why style matters

● Craftsmanship
● Code reviews
● Bugs
Example

Option 1

if [ "${var}" = "text" ]; then
  echo "found text"
fi

Option 2

[ "${var}" = "text" ] && echo "found text"
Example: be concise?

Option 1

command
if [ $? -ne 0 ]; then
    echo "command failed"; exit 1
fi

Option 2

command || { echo "command failed"; exit 1; }

Option 3

if ! command; then echo "command failed"; exit 1; fi
var or VAR?

var
Few variables
Few times used

VAR
Many variables
Used a lot in script
Commands

Use full options

--quiet instead of -q
--verbose instead -v
etc
# Style guide

## Shell Style Guide

Revision 1.26

Paul Armstrong

Too many more to mention

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ➤. You may toggle all summaries with the big arrow button.

### Table of Contents

<table>
<thead>
<tr>
<th>Shell Files and Interpreter Invocation</th>
<th>File Extensions</th>
<th>SUID/SGID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>STDOUT vs STDERR</td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td>File Header</td>
<td>Function Comments</td>
</tr>
<tr>
<td>Formatting</td>
<td>Indentation</td>
<td>Line Length and Long Strings</td>
</tr>
<tr>
<td>Features and Flows</td>
<td>Variable expansion</td>
<td>Quoting</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>Command Substitution</td>
<td>Test if and if</td>
</tr>
<tr>
<td></td>
<td>Pipes to While</td>
<td></td>
</tr>
<tr>
<td>Calling Commands</td>
<td>Checking Return Values</td>
<td>Builtin Commands vs. External Commands</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Source Filenames</td>
<td>Read-only Variables</td>
</tr>
</tbody>
</table>
Focus on reliability
Reliability

- Quality
- Do(n’t) make assumptions
- Expect the unexpected
- Consider worst case scenario
- Practice *defensive programming*
Defensive programming

Wikipedia:

“is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.”

“practices are often used where high availability, safety or security is needed.”
Defenses

Intended operating system?

```bash
#!/bin/sh
if [ ! "$(uname)" = "Linux" ]; then
echo "This is not a Linux system and unsupported"
exit 1
fi
```
Defenses

```bash
#!/bin/sh

if ! $(awk -F= '{if($1 == "NAME" && $2 ~ /^"CentOS|Ubuntu"$/){rc = 1};
{exit !rc}}' /etc/os-release 2> /dev/null)
then
  echo "Not CentOS or Ubuntu"
  exit 1
fi
```
Defenses

set -o nounset
(set -u)

Stop at empty variable
Useful for all scripts
Defenses

set -o errexit
(set -e)

Exit upon $? -gt 0
Useful for scripts with dependant tasks
Use command || true to allow exception
Defenses

set -o pipefail

Useful for scripts with pipes: mysqldump | gzip
(Not POSIX…)

46
Defenses

set -o noglob
(set -f)

Disable globbing (e.g. *)
Useful for scripts which deals with unknown files
Defenses

set -o noclobber
(set -C)

Don’t truncate files, unless >| is used
Defenses

1 #!/bin/sh
2 set -o noclobber
3 MYLOG="myscript.log"
4 echo "$(date --rfc-3339=seconds) Start of script" >| ${MYLOG}
5 echo "$(date --rfc-3339=seconds) Something" > ${MYLOG}

11: ./script: cannot create myscript.log: File exists
Defenses

Caveat of set options

Enable with - (minus)
Disable with + (plus)

Learn more: The Set Builtin
Defenses

Reset localization

export LC_ALL=C
Defenses

Execution path

export PATH="/bin:/sbin:/usr/bin:/usr/sbin"
Defenses

Use quotes and curly brackets, they are free

[ $foo = "bar" ]
[ "$foo" = "bar" ]
[ "${foo}" = "bar" ]
Defenses

Read-only variables

readonly MYVAR="$(hostname -s)"

(Not POSIX…)
Defenses

Use traps
trap cleanup INT TERM
trap status USR1
Defenses

Untrap

trap - EXIT
Defenses

Temporary files

mktemp /tmp/data.XXXXXXXXXXX
Tools
Linting

IS SHOWING A LOT OF WARNINGS! WHAT CAN I DO?

© GARABATOKID
bash -n

$ echo 'myvar="TEST"' | bash -n
bash: line 1: unexpected EOF while looking for matching ``'
bash: line 2: syntax error: unexpected end of file

17: ./sync-vm-backups-to-usb: Syntax error: "(" unexpected (expecting "then")

Alternative: bash -n script
sh

- Name?
- Formatting

https://github.com/mvdan/sh

usage: shfmt [flags] [path ...]

If no arguments are given, standard input will be used. If a given path is a directory, it will be recursively searched for shell files - both by filename extension and by shebang.

- `version` show version and exit
- `-l` list files whose formatting differs from shfmt's
- `-w` write result to file instead of stdout
- `-d` error with a diff when the formatting differs
- `-s` simplify the code

Parser options:

- `-ln str` language variant to parse (bash/posix/mksh, default "bash")
- `-p` shorthand for `-ln=posix`

Printer options:

- `-i uint` indent: 0 for tabs (default), >0 for number of spaces
- `-bn` binary ops like && and | may start a line
- `-cl` switch cases will be indented
- `-sr` redirect operators will be followed by a space
- `-kp` keep column alignment paddings
- `-mn` minify program to reduce its size (implies -s)

Utilities:

- `-f` recursively find all shell files and print the paths
- `-tojson` print syntax tree to stdout as a typed JSON
sh: POSIX check

$ echo '(((total=5*7)))' | ./shfmt -p
  ( (total=5*7))

$ echo 'my_array=(foo bar)' | ./shfmt -p
  <standard input>:1:10: arrays are a bash/mksh feature
Tool: checkbashisms

$ checkbashisms
Usage: checkbashisms [-n] [-f] [-x] script ...
   or: checkbashisms --help
   or: checkbashisms --version
This script performs basic checks for the presence of bashisms
in /bin/sh scripts and the lack of bashisms in /bin/bash ones.
Tool: checkbashisms

possible bashism in /development/lynis/include/functions line 2417 (type):
  if type -t typeset; then
possible bashism in /development/lynis/include/functions line 2418 (typeset):
  typeset -r $1
Tool: ShellCheck

Usage: shellcheck [OPTIONS...] FILES...

--check-sourced Include warnings from sourced files
--color[=WHEN] Use color (auto, always, never)
--include=CODE1,CODE2.. Consider only given types of warnings
--exclude=CODE1,CODE2.. Exclude types of warnings
--format=FORMAT Output format (checkstyle, diff, gcc, json, json1, quiet, tty)
--enable=check1,check2.. List of optional checks to enable (or 'all')
--source-path=SOURCEPATHS Specify path when looking for sourced files ("SCRIPTDIR" for script's dir)
--shell=SHELLNAME Specify dialect (sh, bash, dash, ksh)
--severity=SEVERITY Minimum severity of errors to consider (error, warning, info, style)
--external-sources Allow 'source' outside of FILES
Tool: aspell

Grammar check?
Tool: Automated testing

Verify expectations

Projects:

- Bash Automated Testing System
- shUnit2
- shpec
Conclusions

- Scripts = glue
- Portability or features
- Use other language when needed
- Protect variables
- Check your scripts
What questions do you have?

Get connected

- Twitter (@mboelen)
- LinkedIn (Michael Boelen)
Tips and Tricks
POSIX

Useful links

The Open Group Base Specifications Issue 7, 2018 edition

Shell & Utilities

→ Shell Command Language and Utilities

Utilities

- admin
- alias
- ar
- asa
- at
- awk
When to use bash

<table>
<thead>
<tr>
<th>declare/typeset</th>
<th>Define a variable type (integer, array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays</td>
<td>Data entries</td>
</tr>
<tr>
<td>type</td>
<td>Describe command</td>
</tr>
<tr>
<td>extended globbing</td>
<td>Expand file names</td>
</tr>
<tr>
<td>for loops with integers</td>
<td>for ((i=0; i&lt;10; i++)); do echo $i; done</td>
</tr>
<tr>
<td>extended operator</td>
<td>if [[ &quot;$1&quot; =~ ^m*$ ]]; then</td>
</tr>
<tr>
<td>and more...</td>
<td></td>
</tr>
</tbody>
</table>
POSIX
Binary and built-in
Basic comparisons

Not POSIX
Keyword
Advanced features
Builtins VS binaries

Differences
- Builtin has lower overhead
- Binary may have more features

Commands
- enable -a | awk '{print $2}'
- compgen -b
- builtin
- man builtins
- command -v cd
- type -a [}
## Variables

<table>
<thead>
<tr>
<th></th>
<th>POSIX</th>
<th>bash</th>
<th>ksh</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td>global</td>
<td>global, unless ‘local’ is used</td>
<td>global or local (based on function or funcname())</td>
</tr>
<tr>
<td><strong>Local overrides global?</strong></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Variables

Variable possibly unset? Use:

```bash
if [ "${name:-}" = "Michael" ]; then
  ...
fi
```
Screen output

Use printf instead of echo

Output of echo strongly depends on flags and how it handles escape sequences.
Dealing with fatal errors

#!/bin/sh
Fatal() {
    msg="${1:-"Unknown error"}"
    logger "${msg}"
    echo "Fatal error: ${msg}"
    # optional: call cleanup?
    exit 1
}
command || Fatal "Something happened"
Versioning

Semantic versioning!

Major.Minor.Patch

Learn more: semver.org
Common issues with software

- No clear license
- Unclear goal
- Authorship
- Versioning
- Changelog missing
Changelog

Keep a changelog

- History
- Trust
- Troubleshooting

Learn more: keepachangelog.com
Options

--full-throttle-engine, -f
--help, -h, or help
--version, -V

https://github.com/docopt/docopts

Learn more: docopt.org
Troubleshooting

Use ‘set’ options for debugging:
- `v` (verbose) - input is written stderr
- `x` (xtrace) - show what is executed
FOSS tool? Focus areas

Basics
- Project description
- Tool category
- Typical user
- License
- Author
- Language
- Keywords
- Latest release

Quality
- Changelog
- Popularity
- Documentation
- Code
- Releases

Usage
- Installation
- Ease of use

Ease of use
Tool review

LSE top 10  Lynis (2)

Tool and Usage

<table>
<thead>
<tr>
<th>Project details</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>2007</td>
</tr>
<tr>
<td>License</td>
<td>GPLv3</td>
</tr>
<tr>
<td>Programming language</td>
<td>shell script</td>
</tr>
<tr>
<td>Author</td>
<td>Michael Boelen</td>
</tr>
<tr>
<td>Latest release</td>
<td>2.6.8 [2018-08-25]</td>
</tr>
</tbody>
</table>

Project health

This score is calculated by different factors, like project age, last release date, etc.
Let's torn down something!

#!/bin/sh
set -u
hostname=$(hostname)
lockfile=/var/lock/create-backups
timestamp=$(date "+%s")
today=$(date "+%F")
gpgkey=$(gpg --keyid-format LONG --list-keys backup@rootkit.nl 2> /dev/null | awk '/^pub/ { print $2 }' | awk -F/ '{ print $2 }' | head -1)

if [ -z "${hostname}" ]; then echo "Error: no hostname found"; exit 1; fi

if [ ! -z "${lockfile}" ]; then
  if [ -f ${lockfile} ]; then
    echo "Error: Backup still running. Removing lock file to prevent backup script running next day"
    rm ${lockfile}
    exit 1
  fi
fi

if [ ! -f /etc/duplicity/filelist-patterns ]; then echo "Could not find filelist-patterns"; exit 1; fi

# Run backup
/usr/bin/duplicity --encrypt-key ${gpgkey} --full-if-older-than 1W --ssh-options="-oProtocol=2 -oIdentityFile=/root/.cron/rsync-backup-key" --include-filelist /etc/duplicity/filelist-patterns --verbosity error --no-print-statistics / rsync://10.0.0.50:873::${hostname}

if [ ! -z "${lockfile}" ]; then
  if [ -f ${lockfile} ]; then rm ${lockfile}; fi
fi

# Add a daily timestamp to the file for restore checking
echo "${hostname}-${timestamp}-${today}" > /etc/backup.data
Useful reads

Bash documentation: https://www.gnu.org/software/bash/manual/html_node/
The Bash Hackers Wiki: https://wiki-dev.bash-hackers.org/
Bash pitfalls: http://mywiki.wooledge.org/BashPitfalls
Cheat sheet: https://devhints.io/bash
Rich’s sh (POSIX shell) tricks: www.etalabs.net/sh_tricks.html

And check out Lynis source code: https://github.com/CISOfy/lynis
Credits

Images
Where possible the origin of the used images are included in the slides. Some came without an origin from social media and therefore have no source. If you are the owner, let us know and we add the source.
Nothing makes you more humble than having to maintain a single codebase for years. Only then do you really get see the full extent of your poor choices and ill-informed thinking. I feel lucky to have this learning opportunity.