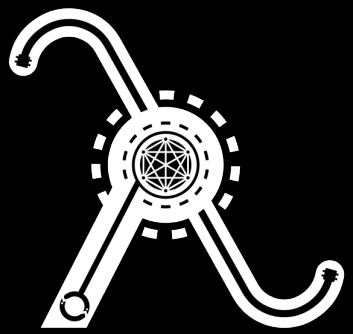
Beautiful Bash: A community driven effort

Lets make reading & writing Bash scripts fun again!

Aaron Zauner
azet@azet.org



lambda.co.at:
Highly-Available, Scalable & Secure Distributed Systems

DevOps/Security Meetup Vienna - 17/12/2014

Introduction

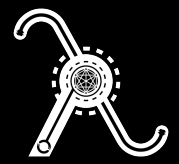
Working towards a community style guide

Doing it wrong

Modern Bash scripting (Welcome to 2014!)

Conclusion

Caveat Emptor

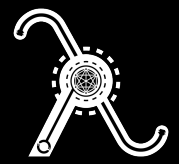


I'm not endorsing Bash for large-scale projects, difficult or performance critical tasks. If your project needs to talk to a database, object store, interact with a filesystem or dynamically handle block devices - you **SHOULD NOT** use Bash in the first place. You can. But you'll regret it - I speak from years of experience doing completely insane stuff in Bash for fun (certainly not for profit).

Bash is useful for one thing and one thing only: as *glue*!

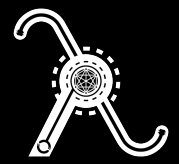
..and it's the glue that holds Linux distributions, Embedded Appliances and even Commercial networking gear together - so you better use the best glue on the market, right?

Do we really need another style guide?



- For starters: It's not only a style guide, but more on that later.
- A lot of the internet actually runs on poorly written Bash.
- Your company probably depends on a lot of Bash-glue.
- Everyone uses it on a daily basis to glue userland utilities together.
- Some scripts unintentionally look like they are submissions for an obfuscated code contest.
- There are some style guides (e.g. by Google) and tutorials – but nothing definitive.
- Most books on the subject are ancient and often reflect personal opinions of authors, outdated Bash versions and userland utilities and most haven't been updated in decades.
- I don't know a single good book on Bash. The best resource is still `http://wiki.bash-hackers.org`.

Working towards a community style guide



- I've started collecting style guides, tutorials, write-ups, tools and debugging projects during the last couple of years.
  ..chose the best ideas and clearest styles and combined them into one big community driven effort.
- People started contributing.
- Nothing is written in stone. Come up with a better idea for a certain topic and I'll gladly accept it.
- I've also included a lot of mistakes people do or even rely on when writing their (often production) scripts.
- I've also collected a lot of tricks and shortcuts I've learned over the years specific to bash scripting and the Linux userland.

Bad Example

Here's a cool and bad example at the same time. `rpm2cpio` reimplemented in bash.

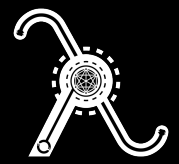
- As Debian package: `Installed-Size:  1044`
- As Bash script: `4`

Bad Example (cont.)

```
Line
   1 #!/bin/sh
   2
   3 pkg=$1
   4 if [ "$pkg" = "" -o ! -e "$pkg" ]; then
   5     echo "no package supplied" 1>&2
   6    exit 1
   7 fi
   8
   9 leadsize=96
  10 o=`expr $leadsize + 8`
  11 set `od -j $o -N 8 -t u1 $pkg`
  12 il=`expr 256 \* \( 256 \* \( 256 \* $2 + $3 \) + $4 \) + $5`
  13 dl=`expr 256 \* \( 256 \* \( 256 \* $6 + $7 \) + $8 \) + $9`
  14 # echo "sig il: $il dl: $dl"
  15
  16 sigsize=`expr 8 + 16 \* $il + $dl`
  17 o=`expr $o + $sigsize + \( 8 - \( $sigsize \% 8 \) \) \% 8 + 8`
  18 set `od -j $o -N 8 -t u1 $pkg`
  19 il=`expr 256 \* \( 256 \* \( 256 \* $2 + $3 \) + $4 \) + $5`
  20 dl=`expr 256 \* \( 256 \* \( 256 \* $6 + $7 \) + $8 \) + $9`
  21 # echo "hdr il: $il dl: $dl"
  22
  23 hdrsize=`expr 8 + 16 \* $il + $dl`
  24 o=`expr $o + $hdrsize`
  25 EXTRACTOR="dd if=$pkg ibs=$o skip=1"
  26
  27 COMPRESSION=`($EXTRACTOR |file -) 2>/dev/null`
  28 if echo $COMPRESSION |grep -q gzip; then
  29         DECOMPRESSOR=gunzip
  30 elif echo $COMPRESSION |grep -q bzip2; then
  31         DECOMPRESSOR=bunzip2
  32 elif echo $COMPRESSION |grep -q xz; then
  33         DECOMPRESSOR=unxz
  34 elif echo $COMPRESSION |grep -q cpio; then
  35         DECOMPRESSOR=cat
  36 else
  37         # Most versions of file don't support LZMA, therefore we assume
  38         # anything not detected is LZMA
  39         DECOMPRESSOR=`which unlzma 2>/dev/null`
  40         case "$DECOMPRESSOR" in
  41             /* ) ;;
  42             *  ) DECOMPRESSOR=`which lzmash 2>/dev/null`
  43                  case "$DECOMPRESSOR" in
  44                      /* ) DECOMPRESSOR="lzmash -d -c" ;;
  45                      *  ) DECOMPRESSOR=cat ;;
  46                  esac
  47                  ;;
  48         esac
  49 fi
  50
  51 $EXTRACTOR 2>/dev/null | $DECOMPRESSOR
```

https://trac.macports.org/attachment/ticket/33444/rpm2cpio

Common bad style practices



- overusing `grep` for tasks that Bash can do by itself.
- using bourne-shell backticks instead of `$()` for subshell calls.
  .. ever tried to nest backtick subshells? yea. you'll have to escape them. instead of e.g.:
  `$(util1 $(util2 ${some_variable_as_argument}))`.
- manual argument parsing instead of using the `getopts` builtin.
- using `awk` for arithmetic operations bash can do very well.
  .. same goes for `expr(1)`. please stop using it in bash scripts.
  .. same goes for `bc(1)`. please stop using it in bash scripts.

Common bad style practices (cont.)

- using the **`echo`** builtin where **`printf`** can (and probably should) be used.
- using **`seq 1 15`** for range expressions instead of **`{1..15}`**
- many **`coreutils`** you do not need & you save on subshell calls. .. a lot is set as a variable in your environment already (protip: see what **`env`** gives you to work with in the first place)
- worst of all: endless and unreadable pipe glue.....

Common bad style practices (cont.)

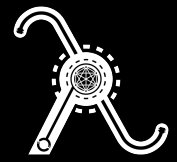
So what is more readable to you and probably the angry sysadmin that might take over your codebase at some point in time?

```
ls ${long_list_of_parameters} | grep ${foo} | grep -v
grep | pgrep | wc -l | sort | uniq
```

or

```
ls ${long_list_of_parameters}   \
    | grep ${foo}               \
    | grep -v grep              \
    | pgrep                     \
    | wc -l                     \
    | sort                      \
    | uniq
```

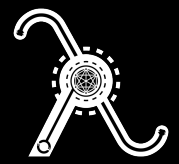
Debugging is a mess



One of the reasons nobody should aim for big projects in Bash is that it is terrible to debug, most of you will know this already.

This project aims to make it easier for you to debug your scripts. By writing beautiful, solid and testable code.

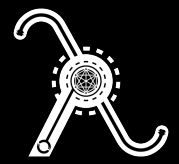
Modern Bash scripting



Most people don't know that there are a lot of useful paradigms and tools that are used for software engineering in serious languages available also to Bash.

Let's not kid ourselves: some Bash scripts will run in production, even for years. They'd better work. And not take your business offline.

Test Driven Development and Unit tests with Bash



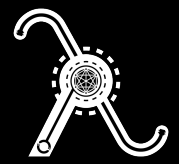
1. Sam Stephenson (of **rbenv** fame) wrote an automated testing system for Bash scripts called 'bats' using TAP (Test Anything Protocol): **https://github.com/sstephenson/bats**
2. **Sharness**: another TAP library. there's even a Chef cookbook for it: **https://github.com/mlafeldt/sharness**
3. **Cram**: a functional testing framework based on Marcurial's unified test format - **https://bitheap.org/cram/**
4. **rnt**: Automated testing of commandline interfaces - **https://github.com/roman-neuhauser/rnt**
5. **shUnit2**: is a xUnit framework (similar to PyUnit, JUnit et cetera) - **https://code.google.com/p/shunit2/**
6. **shpec**: Tests/Specs - **https://github.com/rylnd/shpec**

..there are more, but these I've found to be most useful.

Linting

- A online Bash style linter:
  `https://github.com/koalaman/shellcheck`
- Ubuntu ships with a tool called `checkbashisms` based on Debians `lintian` (portability).
- `shlint` tests for portability between zsh, ksh, bash, dash and bourne shell (if need be):
  `https://github.com/duggan/shlint`
- For Node fans: Grunt task that checks if a Bash script is valid (not anything else, btw):
  `https://www.npmjs.com/package/grunt-lint-bash`

Inter-shell portability



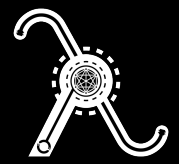
Personal opinion:

Inter-shell portability doesn't matter. I've spent years writing OS agnostic bourne-shell scripts. Today every modern OS ships with a reasonably recent version of Bash. These days Solaris (and FOSS forks like SmartOS) ship even with a GNU userland. Use Bash.
I love **`zsh`** and it can do a lot more. I still use Bash for (semi-) production scripts. They run basically everywhere when done right.

Defensive Bash programming

- As you would in every other language, write helper functions, test these functions.
- Set constants `readonly`.
- Write concise, well defined and tested functions for every action.
- Use the `local` keyword for function-local variables.
- Prepend every function with the `function` keyword.
- Return proper error codes and check for them.
- Write unit tests.
- Some people write a `function main()` as people would with Python. So one can import and test ones `main` call as well.

Defensive Bash programming (cont.)



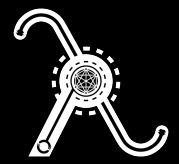
```
function fail() {
  local msg=${@}

  # handle failure appropriately
  cleanup && logger "my message to syslog"

  echo "ERROR: ${msg}"
  exit 1
}
```

et cetera

Defensive Bash programming (cont.)



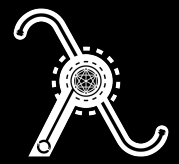
```
function linux_distro() {
  local releasefile=$(cat /etc/*release* 2> /dev/null)
  case ${releasefile} in
  *Debian*)              printf "debian\n" ;;
  *Suse*)                printf "sles\n"   ;;
  *CentOS* | *RedHat*)   printf "el\n"     ;;
  *)                     return 1          ;;
  esac
}

...
[[ $(linux_distro) ]] || fail "Unkown distribution!"
readonly linux_distro=$(linux_distro)
...
```

Defensive Bash programming (cont.)

```
function debian_version() {
  # convert debian version to single unsigned integer
  local dv=$(printf "%.f" $(</etc/debian_version))
  printf "%u" ${dv}
}
```

Defensive Bash programming (cont.)

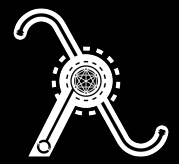


```
function is_empty() {
  local var=${1}
  [[ -z ${var} ]]
}

function is_file() {
  local file=${1}
  [[ -f ${file} ]]
}

function is_dir() {
  local dir=${1}
  [[ -d ${dir} ]]
}
```

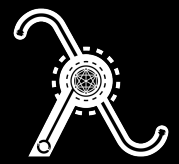
Signal Handling



- Bash supports signal handling with the builtin **trap**:

```
# call the fail() function if one
# of these signals is caught by trap:
trap ,fail "caught signal!", HUP KILL QUIT
```

Anonymous Functions (Lambdas)



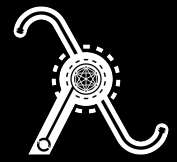
You'll probably never ever need this in Bash, but it's possible:

```
function lambda() {
  _f=${1} ; shift
  function _l {
    eval ${_f};
  }
  _l ${*} ; unset _l
}
```

Bash Profiling

- Sam Stephenson also wrote a profiler for Bash scripts: `https://github.com/sstephenson/bashprof`

Bash Debugging



Hopefully you'll write code that you do not have to debug often, but eventually you'll have to. There's only one real way to debug a Bash script unfortunately:

- `bash -evx script.sh`
- or setting `set -evx` in your script directly
- that being said, someone wrote a Bash debugger with `gdb` command syntax: `http://bashdb.sourceforge.net/`

Conclusion

- There's a lot more to tell (just ask me afterwards) – but this was supposed to be a lightning talk.
- All this, a lot of references and other projects are mentioned in my **`Community Bash Style Guide`** which is on GitHub.
- Please contribute in any way you can if you come up with useful Bashisms, tricks or find any cool projects.
- Any input is very much appreciated!

Fork and open Pull Requests, Issues or Complaints!

`https://github.com/azet/community_bash_style_guide`

Thanks for your patience. Are there any questions?

Twitter:

@a_z_e_t

E-Mail:

azet@azet.org

XMPP:

azet@jabber.ccc.de

GitHub:

https://github.com/azet

GPG Fingerprint:

7CB6 197E 385A 02DC 15D8 E223 E4DB 6492 FDB9 B5D5

[I have ECDSA (Brainpool) & EdDSA (Curve25519) subkeys as well.]