

UNIX & LINUX

dd-style parameters to a bash script

Asked 7 years, 1 month ago Modified 7 years, 1 month ago Viewed 1k times



I would like to pass params to a bash script, dd-style. Basically, I want

19

```
./script a=1 b=43
```



to have the same effect as



4

```
a=1 b=43 ./script
```



I thought I could achieve this with:

```
for arg in "$@"; do
    eval "$arg";
done
```

What's a good way of ensuring that the `eval` is safe, i.e. that `"$arg"` matches a static (no code execution), variable assignment?

Or is there a better way to do this? (I would like to keep this simple).

bash

Share Edit Follow

edited Jun 4, 2015 at 16:35

asked Jun 4, 2015 at 16:33



PSkocik

26.4k

12

77

137

This is tagged with bash. Do you want a Posix compliant solution, or will you accept bash solutions?
– [rici](#) Jun 4, 2015 at 18:16

What the tag says is what I mean :) – [PSkocik](#) Jun 4, 2015 at 18:17

Well you could just parse it as a pattern with a `=` separator and do the assignment with a more carefully constructed `eval`. Just for safety, for private use, I'd do it as you did it. – [orion](#) Jun 5, 2015 at 17:39

6 Answers

Sorted by:

Highest score (default)



You can do this in bash without eval (and without artificial escaping):

16

```
for arg in "$@"; do
    if [[ $arg =~ ^[:alpha:]_[:alnum:]* ]]; then
        declare +i +a +A "$arg"
    fi
done
```

Edit: Based on a comment by Stéphane Chazelas, I added flags to the declare to avoid having the variable assigned being already declared as an array or integer variable, which will avoid a number of cases in which declare will evaluate the value part of the key=val argument. (The +a will cause an error if the variable to be set is already declared as an array variable, for example.) All of these vulnerabilities relate to using this syntax to reassign existing (array or integer) variables, which would typically be well-known shell variables.

In fact, this is just an instance of a class of injection attacks which will equally affect eval - based solutions: it would really be much better to only allow known argument names than to blindly set whichever variable happened to be present in the command-line. (Consider what happens if the command line sets PATH, for example. Or resets PS1 to include some evaluation which will happen at the next prompt display.)

Rather than use bash variables, I'd prefer to use an associative array of named arguments, which is both easier to set, and much safer. Alternatively, it could set actual bash variables, but only if their names are in an associative array of legitimate arguments.

As an example of the latter approach:

```
# Could use this array for default values, too.
declare -A options=[bs]=[if]=[of]=
for arg in "$@"; do
    # Make sure that it is an assignment.
    # -v is not an option for many bash versions
    if [[ $arg =~ ^[:alpha:]_[:alnum:]* &&
        ${options[${arg%*=}]+ok} == ok ]]; then
        declare "$arg"
        # or, to put it into the options array
        # options[${arg%*=}]=${arg#*=}
    fi
done
```

Share Edit Follow

edited Jun 4, 2015 at 20:08

answered Jun 4, 2015 at 18:26



rici

9,282

1

34

36

- 1 The regex seems to have the brackets wrong. Perhaps use this instead: `^[[:alpha:]]_[[:alnum:]]*= ?` – [lcd047](#) Jun 4, 2015 at 18:38
- 1 @lcd047: `foo=` is the only way to set `foo` to the empty string, so it should be allowed (IMHO). I fixed the brackets, thanks. – [rici](#) Jun 4, 2015 at 18:47
- 3 `declare` is about as dangerous as `eval` (one may even say worse as it's not as apparent that it is as dangerous). Try for instance to call that with `'DIRSTACK=$(echo rm -rf ~)'` as argument. – [Stéphane Chazelas](#) Jun 4, 2015 at 19:03
- 1 @PSkocik: `+x` is "not `-x`". `-a` = indexed array, `-A` = associative array, `-i` = integer variable. Thus: not indexed array, not associative array, not integer. – [lcd047](#) Jun 4, 2015 at 19:20
- 1 Note that with the next version of `bash`, you may need to add `+c` to disable compound variables or `+F` to disable floating ones. I'd still use `eval` where you know where you stand. – [Stéphane Chazelas](#) Jun 4, 2015 at 19:35

A POSIX one (sets `$<prefix>var` instead of `$var` to avoid problems with special variables like `IFS` / `PATH` ...):

9

```
prefix=my_prefix_
for var do
  case $var in
    (*=)
      case ${var%*=} in
        "" | *[!abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]* ) ;;
        (*) eval "$prefix${var%*=}=${var#*=}"
      esac
    esac
  done
```

Called as `myscript x=1 PATH=/tmp/evil %=3 blah '=foo' 1=2`, it would assign:

```
my_prefix_x <= 1
my_prefix_PATH <= /tmp/evil
my_prefix_1 <= 2
```

Share Edit Follow

answered Jun 4, 2015 at 19:32



[Stéphane Chazelas](#)

473k 84 929 1377

lcd047's solution refactored with a hardcoded DD_OPT_ prefix:

6

```
while [[ $1 =~ ^[[:alpha:]]_[[:alnum:]]*= ]]; do
    eval "DD_OPT_${1%%=*}"="${1#*=}"; shift;
done
```

[frostschutz](#) deserves the credit for most of the refactoring.

I put this in a source file with as global variable:

```
DD_OPTS_PARSE=$(cat <<'EOF'
while [[ $1 =~ ^[[:alpha:]]_[[:alnum:]]*= ]]; do
    eval "DD_OPT_${1%%=*}"="${1#*=}"; shift;
done
EOF
)
```

eval "\$DD_OPTS_PARSE" does all the magic.

A version for functions would be:

```
DD_OPTS_PARSE_LOCAL="${PARSE_AND_REMOVE_DD_OPTS/DD_OPT_/local DD_OPT_}"
```

In use:

```
eval "$DD_OPTS_PARSE_LOCAL"
```

I made a [repo](#) out of this, complete with tests and a README.md. Then I used this in a Github API CLI wrapper I was writing, and I used the same wrapper to setup a github clone of said [repo](#) (bootstrapping is fun).

Safe parameter passing for bash scripts in just one line. Enjoy. :)

Share Edit Follow

edited Apr 13, 2017 at 12:36



Community Bot

1

answered Jun 4, 2015 at 18:44





PSkocik

26.4k

12

77

137

-
- 1 but you can get rid of `*=` and stop substituting `key/val` where there is no `=`. (since you were refactoring) :P – [frostschutz](#) Jun 4, 2015 at 18:50 
-
- 1 in fact you can get rid of the for loop and the if and use while `$1` instead, since you're shifting and all... – [frostschutz](#) Jun 4, 2015 at 18:52
-
- 1 Heh, the proof that brainstorming works. :) – [lcd047](#) Jun 4, 2015 at 19:24
-
- 1 Harvesting morning ideas: you can even get rid of `key` and `val`, and just write `eval "${1%%=*}"="\${1#*=}"`. But that's pretty much as far as it goes, `eval "$1"` as in [@rici's](#) `declare "$arg"` won't work, obviously. Also beware of setting things like `PATH` or `PS1`. – [lcd047](#) Jun 5, 2015 at 4:30 
-
- 1 Thank you - I thought that was the evaluated variable. I appreciate your patience with me - that's pretty glaringly obvious. Anyway, no - other than that imagined one, it looks good. You know though you could extend this to work in any shell with `case`. Probably it doesn't matter, but just in case you didn't know... – [mikeserv](#) Jun 7, 2015 at 15:54
-

5

Classic Bourne shell supported, and Bash and Korn shell still support, a `-k` option. When it is in effect, any 'dd-like' command options anywhere on the command line are converted automatically into environment variables passed to the command:

```
$ set -k
$ echo a=1 b=2 c=3
$
```

It's a bit harder to be convincing that they're environment variables; running this works for me:

```
$ set -k
$ env | grep '^[a-z]=' # No environment a, b, c
$ bash -c 'echo "Args: $" >&2; env' a=1 b=2 c=3 | grep '^[a-z]='
Args:
a=1
b=2
c=3
$ set +k
$ bash -c 'echo "Args: $" >&2; env' a=1 b=2 c=3 | grep '^[a-z]='
Args: b=2 c=3
$
```

The first `env | grep` demonstrates no environment variables with a single lower-case letter. The first `bash` shows that there are no arguments passed to the script executed via `-c`, and the environment does contain the three single-letter variables. The `set +k` cancels the `-k`, and shows that the same command now has arguments passed to it. (The `a=1` was treated as `$0` for the script; you can prove that, too, with appropriate echoing.)

This achieves what the question asks — that typing `./script.sh a=1 b=2` should be the same as typing `a=1 b=2 ./script.sh`.

Be aware that you run into problems if you try tricks like this inside a script:

```
if [ -z "$already_invoked_with_minus_k" ]
then set -k; exec "$0" "$@" already_invoked_with_minus_k=1
fi
```

The `"$@"` is treated verbatim; it is not re-analyzed to find assignment-style variables (in both `bash` and `ksh`). I tried:

```
#!/bin/bash

echo "BEFORE"
echo "Arguments:"
a1 "$@"
echo "Environment:"
env | grep -E '^[a-z]|already_invoked_with_minus_k)'=
if [ -z "$already_invoked_with_minus_k" ]
```

```
then set -k; exec "$@" already_invoked_with_minus_k=1
fi

echo "AFTER"
echo "Arguments:"
al "$@"
echo "Environment:"
env | grep -E '^[a-z]|already_invoked_with_minus_k='

unset already_invoked_with_minus_k
```

and only the `already_invoked_with_minus_k` environment variable is set in the `exec 'd` script.

Share Edit Follow

edited Jun 5, 2015 at 15:11

answered Jun 5, 2015 at 14:49



[Jonathan Leffler](#)

1,459 13 14

Very nice answer! It's interesting that this won't change PATH, though HOME is changable so there must be something like a blacklist (containing PATH at least) of env vars that'd would be too dangerous to set in this way. I love how this is ultra-short and answers the question, but I'll go with the sanitize+eval+prefix solution as it's still safer and thereby more universally usable (in environments where you don't want users to mess with the environment). Thanks and +1. – [PSkocik](#) Jun 7, 2015 at 10:19

My attempt:

2

```
#!/usr/bin/env bash
name='^[a-zA-Z][a-zA-Z0-9_]*$'
count=0
for arg in "$@"; do
    case "$arg" in
        *=*)
            key=${arg%%=*}
            val=${arg#*=}

            [[ "$key" =~ $name ]] && { let count++; eval "$key"="\$val; } ||
                break

            # show time
            if [[ "$key" =~ $name ]]; then
                eval "out=\${$key}"
                printf '%s| <-- |%s|\n' "$key" "$out"
            fi
            ;;
        *)
            break
            ;;
    esac
done
shift $count

# show time again
printf 'arg: |%s|\n' "$@"
```

It works with (almost) arbitrary garbage on the RHS:

```
$ ./assign.sh Foo_Bar33='1 2;3`4"5~6!7@8#9$0 1%2^3&4*5(6)7-8=9+0' '1
2;3`4"5~6!7@8#9$0 1%2^3&4*5(6)7-8=9+0=33'
|Foo_Bar33| <-- |1 2;3`4"5~6!7@8#9$0 1%2^3&4*5(6)7-8=9+0|
arg: |1 2;3`4"5~6!7@8#9$0 1%2^3&4*5(6)7-8=9+0=33|

$ ./assign.sh a=1 b=2 c d=4
|a| <-- |1|
|b| <-- |2|
arg: |c|
arg: |d=4|
```

Share Edit Follow

edited Jun 4, 2015 at 18:58

community wiki
8 revs, 2 users 98%
lcd047

shift will kill the wrong things if you don't break the loop at the first non-x=y parameter – [frostschutz](#) Jun 4, 2015 at 17:37

@frostschutz Good point, edited. – [lcd047](#) Jun 4, 2015 at 17:53

Nice job at generalizing it. I think it can be simplified a little. – [PSkocik](#) Jun 4, 2015 at 18:14

Did you get the chance to take a look at my edit? – [PSkocik](#) Jun 4, 2015 at 18:16

Please take a look at my edit. That's the way I like it (+maybe just `shift` instead of `shift 1`).
Otherwise thanks! – [PSkocik](#) Jun 4, 2015 at 18:25

Some time ago I settled on `alias` for this kind of work. Here's some of another answer of mine:

0

It can sometimes be possible to separate the evaluation and execution of such statements, though. For example, `alias` can be used to pre-evaluate a command. In the following example the variable definition is saved to an alias that can only be successfully declared if the `$var` variable it is evaluating contains no bytes that do not match ASCII alphanumerics or `-`.

```
LC_OLD=$LC_ALL LC_ALL=C
for var do    val=${var#*=} var=${var%*=}
    alias    "${var##*[_A-Z0-9a-z]*}=_$var=\$val" &&
    eval    "${var##[0-9]*}" && unalias "$var"
done;        LC_ALL=$LC_OLD
```

`eval` is used here to handle invoking the new `alias` from a quoted varname context - not for the assignment exactly. And `eval` is only called at all if the previous `alias` definition is successful, and while I know a lot of different implementations will accept a lot of different kinds of values for alias names, I haven't yet found a shell that will accept a completely empty one.

The definition within the alias is for `_$var`, however, and this is to ensure that no significant environment values are written over. I don't know of any noteworthy environment values beginning with a `_` and it is usually a safe bet for semi-private declaration.

Anyway, if the alias definition is successful it will declare an alias named for `$var`'s value. And `eval` will only call that `alias` if it also does not start with a number - else `eval` gets only a null argument. So if both conditions are met `eval` calls the `alias` and the variable definition saved in the `alias` is made, after which the new alias is promptly removed from the hash table.

Also useful about `alias` in this context is that you can print your work. `alias` will print a doubly-quoted *safe-for-shell-rexecution* statement when asked.

```
sh -c "IFS=\`
alias q=\"\$*\` q" -- \
some args which alias \
will print back at us
```

OUTPUT

```
q='some''''args''''which''''alias''''will''''print''''back''''at''''us'
```

Share Edit Follow

answered Jun 7, 2015 at 13:14



[mikeserv](#)

55.9k

9

103

216