Eval command and security issues

The `eval` command is extremely powerful and extremely easy to abuse.

It causes your code to be parsed twice instead of once; this means that, for example, if your code has variable references in it, the shell's parser will evaluate the contents of that variable. If the variable contains a shell command, the shell might run that command, whether you wanted it to or not. This can lead to unexpected results, especially when variables can be read from untrusted sources (like users or user-created files).

Examples of bad use of eval

"eval" is a common misspelling of "evil".

One of the most common reasons people try to use `eval` is because they want to pass the name of a variable to a function. Consider:

```
# This code is evil and should never be used!
fifth() {
    _fifth_array=$1
    eval echo "\"The fifth element is \${$_fifth_array[4]}\""    # DANGER!
}
a=(zero one two three four five)
fifth a
```

This breaks if the user is allowed to pass arbitary arguments to the function:

```
$ fifth 'x}"; date; #'
The fifth element is
Thu Mar 27 16:13:47 EDT 2014
```

We've just allowed arbitary code execution. Bash 4.3 introduced *name references* to try to solve this problem, but unfortunately they **don't** solve the problem! We'll discuss those in depth later.

Now let's consider a more complicated example. The section of this FAQ dealing with spaces in file names used to include the following "helpful tool (which is probably not as safe as the \0 technique)".

```
Syntax : nasty_find_all <path> <command> [maxdepth]
```

```
# This code is evil and must never be used!
export IFS=" "
[ -z "$3" ] && set -- "$1" "$2" 1
FILES=`find "$1" -maxdepth "$3" -type f -printf "\"%p\" "`
# warning, BAD code
eval FILES=($FILES)
for ((I=0; I < ${#FILES[@]}; I++))
do
    eval "$2 \"${FILES[I]}\""
done
unset IFS
```

This script was supposed to recursively search for files and run a user-specified command on them, even if they had newlines and/or spaces in their names. The author thought that `find -print0 | xargs -0` was unsuitable for some purposes such as multiple commands. It was followed by an instructional description of all the lines involved, which we'll skip.

To its defense, it worked:

```
$ ls -lR
.:
total 8
drwxr-xr-x  2 vidar users 4096 Nov 12 21:51 dir with spaces
-rwxr-xr-x  1 vidar users  248 Nov 12 21:50 nasty_find_all

./dir with spaces:
total 0
-rw-r--r--  1 vidar users 0 Nov 12 21:51 file?with newlines
$ ./nasty_find_all . echo 3
./nasty_find_all
./dir with spaces/file
with newlines
$
```

But consider this:

```
$ touch "\"); ls -l $'\x2F'; #"
```

You just created a file called `"); ls -l $'\x2F'; #`

Now FILES will contain `""); ls -l $'\x2F'; #`. When we do `eval FILES=($FILES)`, it becomes

```
FILES=(""); ls -l $'\x2F'; #"
```

Which becomes the two statements `FILES=("");` and `ls -l /`. Congratulations, you just allowed execution of arbitrary commands.

```
$ touch "\"); ls -l $'\x2F'; #"
$ ./nasty_find_all . echo 3
total 1052
-rw-r--r--   1 root root 1018530 Apr  6  2005 System.map
drwxr-xr-x   2 root root    4096 Oct 26 22:05 bin
drwxr-xr-x   3 root root    4096 Oct 26 22:05 boot
drwxr-xr-x  17 root root   29500 Nov 12 20:52 dev
drwxr-xr-x  68 root root    4096 Nov 12 20:54 etc
drwxr-xr-x   9 root root    4096 Oct  5 11:37 home
drwxr-xr-x  10 root root    4096 Oct 26 22:05 lib
drwxr-xr-x   2 root root    4096 Nov  4 00:14 lost+found
drwxr-xr-x   6 root root    4096 Nov  4 18:22 mnt
drwxr-xr-x  11 root root    4096 Oct 26 22:05 opt
dr-xr-xr-x  82 root root       0 Nov  4 00:41 proc
drwx------  26 root root    4096 Oct 26 22:05 root
drwxr-xr-x   2 root root    4096 Nov  4 00:34 sbin
drwxr-xr-x   9 root root       0 Nov  4 00:41 sys
drwxrwxrwt   8 root root    4096 Nov 12 21:55 tmp
drwxr-xr-x  15 root root    4096 Oct 26 22:05 usr
drwxr-xr-x  13 root root    4096 Oct 26 22:05 var
./nasty_find_all
./dir with spaces/file
with newlines
./
$
```

It doesn't take much imagination to replace `ls -l` with `rm -rf` or worse.

One might think these circumstances are obscure, but one should not be tricked by this. All it takes is one malicious user, or perhaps more likely, a benign user who left the terminal unlocked when going to the bathroom, or wrote a funny PHP uploading script that doesn't sanity check file names, or who made the same mistake as oneself in allowing arbitrary code execution (now instead of being limited to the www-user, an attacker can use `nasty_find_all` to traverse chroot jails and/or gain additional privileges), or uses an IRC or IM client that's too liberal in the filenames it accepts for file transfers or conversation logs, etc.

The problem with bash's name references

Bash 4.3 introduced `declare -n` ("name references") to mimic Korn shell's `nameref` feature, which permits variables to hold references to other variables (see FAQ 006 to see these in action). Unfortunately, the implementation used in Bash has some issues.

First, Bash's `declare -n` doesn't actually avoid the name collision issue:

```
$ foo() { declare -n v=$1; }
$ bar() { declare -n v=$1; foo v; }
$ bar v
bash: warning: v: circular name reference
```

In other words, there is no safe name we can give to the name reference. If the caller's variable happens to have the same name, we're screwed.

Second, Bash's name reference implementation still allows *arbitrary code execution*:

```
$ foo() { declare -n var=$1; echo "$var"; }
$ foo 'x[i=$(date)]'
bash: i=Thu Mar 27 16:34:09 EDT 2014: syntax error in expression (error token is "Mar 27 16:34:09 EDT 2014")
```

It's not an elegant example, but you can clearly see that the `date` command was actually *executed*. This is not at all what one wants.

Now, despite these shortcomings, the `declare -n` feature is a step in the right direction. But you must be careful to select a name that the caller won't use (which means you need *some* control over the caller, if only to say "don't use variables that begin with _my_pkg"), and you must reject unsafe inputs.

Examples of good use of eval

The most common correct use of `eval` is reading variables from the output of a program which is **specifically *designed* to be used this way**. For example,

```
# On older systems, one must run this after resizing a window:
eval "`resize`"

# Less primitive: get a passphrase for an SSH private key.
# This is typically executed from a .xsession or .profile type of file.
# The variables produced by ssh-agent will be exported to all the processes in
# the user's session, so that an eventual ssh will inherit them.
eval "`ssh-agent -s`"
```

`eval` has other uses especially when creating variables out of the blue (indirect variable references). Here is an example of one way to parse command line options that do not take parameters:

```
# POSIX
#
# Create option variables dynamically. Try call:
#
#    sh -x example.sh --verbose --test --debug

for i; do
    case $i in
       --test|--verbose|--debug)
            shift                   # Remove option from command line
            name=${i#--}            # Delete option prefix
            eval "$name=\$name"    # make *new* variable
            ;;
    esac
done

echo "verbose: $verbose"
echo "test: $test"
echo "debug: $debug"
```

So, why is this version acceptable? It's acceptable because we have restricted the `eval` command so that it will **only** be executed when the input is one of a finite set of known values. Therefore, it can't ever be abused by the user to cause arbitrary command execution -- any input with funny stuff in it wouldn't match one of the three predetermined possible inputs.

Note that this is **still frowned upon**: It is a slippery slope and some later maintenance can easily turn this code into something dangerous. Eg. You want to *add a feature* that allows a bunch of different --test-xyz's to be passed. You change `--test` to `--test-*`, without going through the trouble of checking the implementation of the rest of the script. You test your use case and it all works. Unfortunately, **you've just introduced arbitrary command execution**:

```
$ ./foo --test-'; ls -l /etc/passwd;x='
-rw-r--r-- 1 root root 943 2007-03-28 12:03 /etc/passwd
```

Once again: by permitting the `eval` command to be used on unfiltered user input, we've permitted arbitrary command execution.

AVOID PASSING DATA TO EVAL AT ALL COSTS, even if your code seems to handle all the edge cases today.

If you have thought really hard and asked #bash for an alternative way but there isn't any, skip ahead to "Robust eval usage".

The problem with declare

Could this not be done better with `declare`?

```
for i in "$@"
do
    case "$i" in
        --test|--verbose|--debug)
            shift                   # Remove option from command line
            name=${i#--}            # Delete option prefix
            declare $name=Yes       # set default value
            ;;
        --test=*|--verbose=*|--debug=*)
            shift
            name=${i#--}
            value=${name#*=}        # value is whatever's after first word and =
            name=${name%%=*}        # restrict name to first word only (even if there's another = in the value)
            declare $name="$value"  # make *new* variable
            ;;
    esac
done
```

Note that `--name` for a default, and `--name=value` are the required formats.

`declare` does work better for some inputs:

```
griffon:~$ name='foo=x;date;x'
griffon:~$ declare $name=Yes
griffon:~$ echo $foo
x;date;x=Yes
```

But it can still cause arbitary code execution with array variables:

```
attoparsec:~$ echo $BASH_VERSION
4.2.24(1)-release
attoparsec:~$ danger='( $(printf "%s!\n" DANGER >&2) )'
attoparsec:~$ declare safe=${danger}
attoparsec:~$ declare -a unsafe
attoparsec:~$ declare unsafe=${danger}
DANGER!
```

Robust eval usage

Almost always (at least 99% or more of the time in Bash, but also in more minimal shells), the correct way to use `eval` is to produce abstractions hidden behind functions used in library code. This allows the function to:

- present a well-defined interface to the function's caller that specifies which inputs must be strictly controlled by the programmer, and which may be unpredictable, such as side-effects influenced by user input. It's important to document which options and arguments are unsafe if left uncontrolled.
- perform input validation on certain kinds of inputs where it's feasible to do so, such as integers -- where it's easy to bail out and return an error status which can be handled by the function caller.
- create abstractions that hide ugly implementation details involving `eval`.

Generally, `eval` is correct when at least all of the following are satisfied:

- All possible arguments to `eval` are guaranteed not to produce harmful side-effects or result in execution of arbitrary code under any circumstance. The inputs are statically coded, free from interaction with uncontrolled dynamic code, and/or validated throughly. This is why functions are important, because YOU don't necessarily have to make that guarantee yourself. So long as your function documents what inputs can be dangerous, you can delegate that task to the function's caller.
- The `eval` usage presents a clean interface to the user or programmer.
- The `eval` makes possible what would otherwise be impossible without far more large, slow, complex, dangerous, ugly, less useful code.

If for some reason you still need to dynamically build bash code and evaluate it, make certain you take these precautions:

1. Always **quote** the `eval` expression: `eval 'a=b'`
2. Always **single-quote** code and expand your data into it using `printf`'s `%q`: `eval "$(printf 'myvar=%q' "$value")"`
3. Do NOT use dynamic variable names. Even with careful `%q` usage, this can be exploited.

Why take heed? Here's how your scripts can be exploited if they fail to take the above advice:

- If you don't single-quote your code, you run the risk of expanding data into it that isn't `%q`'ed. Which means free executable reign for that data:

  ```
  name='Bob; echo I am arbitrary code'; eval "user=$name"
  ```

- Even if you `%q` input data before treating it as a variable name, illegal variable names in assignments cause bash to search `PATH` for a command:

  ```
  echo 'echo I am arbitrary code' > /usr/local/bin/a[1]=b; chmod +x /usr/local/bin/a[1]=b; var='a[1]' value=b; eval "$(printf '%q=%q' "$var" "$value")"
  ```

- For a list of ways to reference or to populate variables indirectly **without** using `eval`, please see BashFAQ/006.
- For a list of ways to reference or to populate variables indirectly **with** `eval`, please see BashFAQ/006#eval.
- More examples

CategoryShell