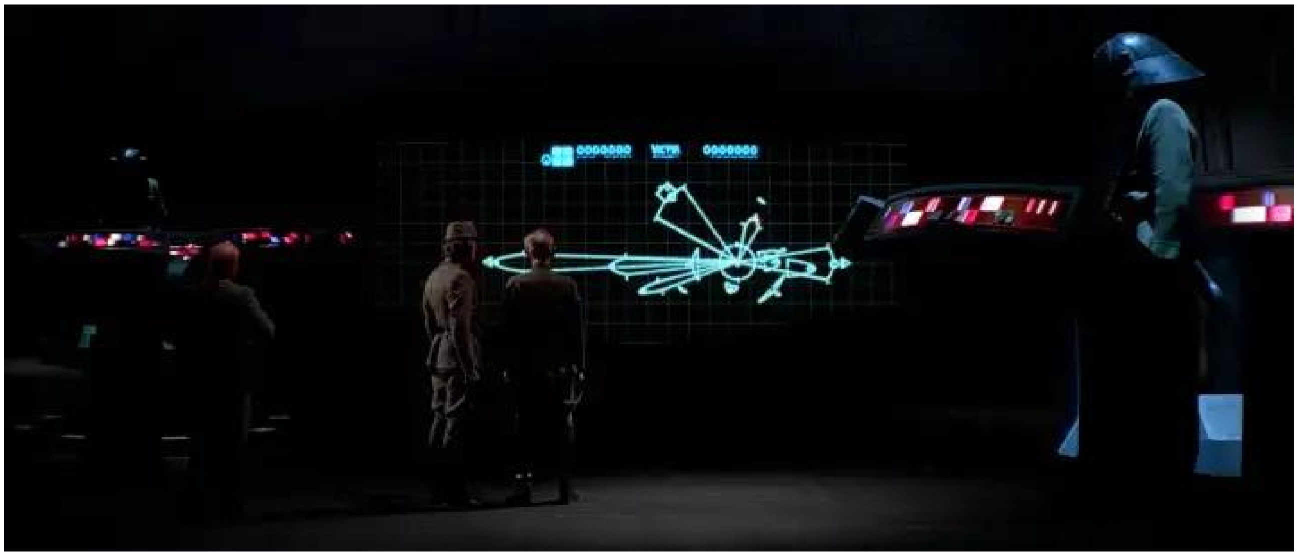
Published in .debug

Stephen Fox Follow
Feb 2, 2017 · 6 min read · Listen


The fateful moments aboard the Death Star prior to the power system running 'eval' against unsanitized user input.
(Image credit: Lusasfilm Ltd.)

The perils of Bash 'eval'

Reflection injection.

Bash is a "macro processor that executes commands", which can be used to solve problems in what amounts to concise, self-contained, text files. While this elegance is easily recognized by operations teams and system admins, it can be lost to developers that constantly work with object oriented programming languages. In my own experience of switching back and forth between Bash and Java, I felt compelled to apply object oriented programming principles to my Bash scripts.

The result of this was generally positive. For example, it led to separating massive blobs of code into concise functions, limiting the scope of variables, and creating more flexible logic.

Ironically, implementing those principles has sometimes led to my own detriment. I am almost universally told by other developers to compact my Bash code during code reviews. This is a fair criticism. When you spend enough time in development tools like IntelliJ or Visual Studio, the IDE practically does this for you before anyone else ever lays eyes on your code. With Bash — it is just you, and a text editor.

In the never ending cat and mouse game of compacting code, you will eventually find yourself reading about the 'eval' Bash builtin. Unfortunately, 'eval' is a very sharp knife, and many developers fail to explain why, let alone warn others of its dangers.

This post will explain and show why implementing 'eval' in your scripts is a really bad idea.

What is 'eval'?

To understand why 'eval' is dangerous, we must first understand what it is used for. As a tool, 'eval' is used to interpret (evaluate) strings into live logic. For example:

```
$ eval echo hello world
hello world
```

The command is not too dissimilar to reflection in programming. This lets you create variables dynamically at runtime. For example, you can create a variable whose name is the name of your current user:

```
$ echo "I am: ${root}"
I am:
```

```
$ eval 'export $(whoami)=$(whoami)'
$ echo "I am: ${root}"
I am: root
```

When you find yourself reading about ‘eval’, it is usually because your current logic has boxed you into one of the few use cases that ‘eval’ can quickly solve. Being a Bash builtin, it is reliably available. The command is also commonly recommended online. It can even be used to execute logic that would normally require several lines of code in a single line of code. These factors make implementing ‘eval’ into your scripts a very tempting proposition.

A significant portion of ‘eval’ examples fail to describe the dangers of implementing the command into your logic. If you are lucky, developers may drop subtle hints, such as “use it cautiously” because it is “evil”. As a Bash newbie, I can attest to weighing these vague warnings against finishing a script and moving onto the next project. Without an adequate explanation, ‘eval’ presents an easy crutch to logic that should be restructured or rewritten.

A really bad idea

Experienced Bash scripters will describe the outcome of implementing ‘eval’ in two different ways; a really bad idea, or consciously introducing a vulnerability into your script. While these concerns are exactly the same, neither actually *describe* why. In my opinion, an over-the-top implementation is needed to sufficiently illustrate why ‘eval’ is dangerous.

The example

Assume that you are looking to create a script that provides a basic, network-accessible API. The goal of the API in this example will be to manage a file named ‘file.txt’ using our own ‘update’ and ‘remove’ commands.

Design

The API will be accessible using HTTP on port 8080. In order to communicate with the API, users must know an “API key” (a special URL) to target. If the user does not provide a valid key, then the script will do nothing. If the user provides the key, but does not provide a valid command, then the script will do nothing. If the user provides a valid key and command, then the script will execute a command.

Implementation

We can use netcat (‘nc’) to listen on a port for HTTP GET requests. To make things easy, we will write the output of ‘nc’ to a file and parse the file for valid requests. I have implemented the example in a Bash script named **really-bad-idea.sh**:

Running it

To start it, execute:

```
$ ./really-bad-idea.sh

[INFO] netcat now listening on port 8080 as PID 768
[INFO] Make API calls to '<server-address>:8080/api/a15d24c3d4b585e84862dc46df98bd2f'
```

Note: *You can press Control + C to stop the script gracefully.*

Positive use case

First, let's take a look at the positive use cases. You can make requests to the script using `curl` like so:

```
# Client request:
# Note: Make sure to use the API key that the script spits out.

$ curl '127.0.0.1:8080/api/a15d24c3d4b585e84862dc46df98bd2f/update'
```

… you will see the following result on the server side:

```
# Server result:
```

```
[INFO] Received API call: '/api/a15d24c3d4b585e84862dc46df98bd2f/update'
[INFO] Updating file...
```

Saving a few lines of code

So far, the script meets the design requirements: it creates a public API that only validated users can utilize. Let's take a look at the code again, specifically the lines that validate the API key in `parse_incoming_data()` :

```
eval local key="$(echo "${line}" | cut -f2 -d' ')" \
    && [ "${key%/*}" == "${G_API_KEY}" ] \
    && execute_api_call "${line}"
```

I used 'eval' here to declare a local variable containing the request's URI. For context, a single HTTP call to netcat will spit out the following:

```
$ nc -l 8080

GET /api/key/uri HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/7.43.0
Accept: */*
```

Since we are interested in the `/api/key/uri` string, it makes sense to just use 'cut' to grab it since there will never be spaces in the string. The end result here is that I used 'eval' to save a few lines of code. Yah!

Not yay: Why 'eval' is a really bad idea

Note*: All of the following testing is done in a disposable Docker container. Be careful if you are fooling around with this script on an important machine. I am not responsible if you delete something important.*

If you recall, 'eval' executes strings and can permit us some reflection-like abilities. Looking at the code, recall the `control_c` function that handles user shutdown gracefully:

```
# control_c
# Executes logic to handle a Control + C press by the user.
control_c() {
    G_USER_REQUESTED_SHUTDOWN=0
    echo ''
    shutdown
}
```

What do you think will happen if we send a 'curl' call containing `$(control_c)` ? Let's find out:

```
# Client request:
$ curl '127.0.0.1:8080/$(shutdown)'

# Server result:
(...)
[ERROR] The listener has exited unexpectedly
```

Ut-oh. I do not think that was in the requirements! Well, what happens if we try touching a file?

```
# Client request:
$ curl '127.0.0.1:8080/$(touch${IFS}/not-good.txt)'

# Server result:
$ ls -ltr /

total 72
drwxr-xr-x   2 root root  4096 Nov  5 15:38 srv
drwxr-xr-x   2 root root  4096 Nov  5 15:38 opt
drwxr-xr-x   2 root root  4096 Nov  5 15:38 mnt
(...)
drwxrwxrwt   7 root root  4096 Feb  2 02:32 tmp
-rwxr-xr-x   1 root root  4163 Feb  2 02:34 really-bad-idea.sh
-rw-r--r--   1 root root     0 Feb  2 04:06 not-good.txt
```

As you can imagine, we can do some comical things like make the script delete itself:

```
# Client request:
$ curl '127.0.0.1:8080/$(rm${IFS}${PWD}/*.sh)'

# Server result:
$ ls /really-bad-idea.sh

ls: /really-bad-idea.sh: No such file or directory
```

While we are removing things we do not want, let's delete the file system:

```
# **WARNING**: THE FOLLOWING COMMAND WILL DELETE YOUR COMPUTER. DO NOT RUN THIS UNLESS YOU ARE IN A TEST ENVIRONMENT.

# Client request:
$ curl '127.0.0.1:8080/$(rm${IFS}-rf${IFS}/*)'

# Server result:
(...)
[INFO] Make API calls to '<server-address>:8080/api/104d01cba6f4b74051901ca500a2650a'
rm: cannot remove '/dev/console': Device or resource busy
rm: cannot remove '/dev/shm': Device or resource busy
rm: cannot remove '/dev/mqueue': Device or resource busy
rm: cannot remove '/dev/pts/ptmx': Operation not permitted
rm: cannot remove '/etc/hosts': Device or resource busy
rm: cannot remove '/etc/resolv.conf': Device or resource busy
rm: cannot remove '/etc/hostname': Device or resource busy
rm: cannot remove '/proc/fb': Operation not permitted
(...)
./really-bad-idea.sh: line 48: /tmp/listener.log: No such file or directory
./really-bad-idea.sh: line 54: /usr/bin/sleep: No such file or directory
./really-bad-idea.sh: line 48: /tmp/listener.log: No such file or directory
./really-bad-idea.sh: line 54: /usr/bin/sleep: No such file or directory
(...)
```

Closing arguments

Bash provides a powerful toolbox full of sharp knives. The ‘eval’ command is just one of those knives. If you must absolutely use Bash in a sensitive environment, please make sure you do everything possible to either isolate it from user input or sanitize the user input first.

Whatever you do, do not use ‘eval’ unless you want to end up like my Docker containers.

Docker Dev Ops Bash Security Information Security