



PRODUCTION
GRADE
BASH
SCRIPTS

KUMAR ASHWIN

\$ whoami

Lead Research & Consulting / Security Engineering at RedHunt Labs

I've been a speaker/trainer at BlackHat, Nullcon, c0c0n, etc.

My work sits at the intersection of code, and security - Offensive + Defensive - mostly in AI Security (obviously) & Supply Chain Security.

Loves to travel across the world when I am not writing bash scripts.

I also do Internet Scanning for Fun & Profit.



me @ 5am today
finishing these slides
:)

What do you mean by
production grade bash scripts???

Once upon a time...



be like alex!!

Alex

Intern - DevOps

Love creating bash scripts and drinking
matcha and playing **Pickleball**!

because it worked!

next morning...

On-Call Engineer's Thoughts

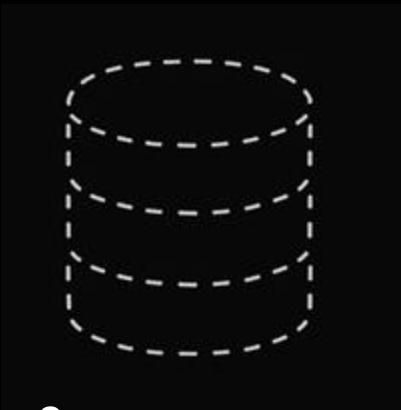
did we get hacked?

is this the correct
grammar?

is life a joke?

is AWS down?

where is the log file?



did our server go down?

am i dumb?

what caused this?

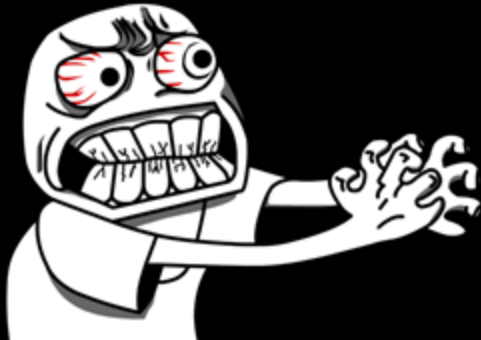
let's the debugging games begin!!

there are no logs!!!

cleanup.sh

...

```
echo "Cleanup Completed Successfully! ✓" > /tmp/log
```



logging to a file

If something breaks, the log should know before you do.

Instead of manually redirecting each command's output, you can route everything with a single line:

`cleanup.sh`

```
exec > >(tee -a "$LOG_FILE" | logger -t tag-name -s) 2>&1
```

It works at the script level, so every line - every error, every echo - gets captured. Perfect for debugging, postmortems, or just knowing what actually happened.

what really makes a good log line?

- No spaces between fields - use a **delimiter**
- Consistent key=value pairs for structured data
- Fixed field order (timestamp, level, then data)
- Unique and searchable field names (e.g., user_id, not just id)
- Machine readability first, human readability second
- No unstructured errors - wrap them with context (reason=timeout instead of "it broke")
- Avoid redundancy - don't repeat info like the timestamp or log level already shows

add more structure to the log

cleanup.sh

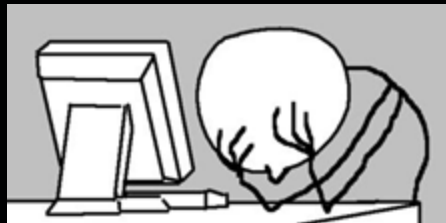
```
LOG_FILE="./script.log"
```

```
log() {  
    local level="$1"  
    shift  
    local ts  
    ts="$(date '+%Y-%m-%dT%H:%M:%S%z')"  
    local fields="$*"   
    local message="$ts::$level::$fields"  
  
    # Color-coded terminal output  
    case "$level" in  
        ERROR) echo -e "\033[0;31m$message\033[0m" >&2 ;;  
        WARN)  echo -e "\033[0;33m$message\033[0m" >&2 ;;  
        *)     echo "$message" >&2 ;;  
    esac  
  
    # Log to file  
    echo "$message" >> "$LOG_FILE"  
}
```

log

```
2025-04-16T13:37:00+0000::[INFO]::event=task_start,user_id=123,task_id=456  
2025-04-16T13:37:01+0000::[ERROR]::event=file_open_failed,file=config.json,reason=NoSuchFile
```

variables were not quoted!



cleanup.sh

```
#!/usr/bin/env bash
```

```
DB_BACKUP_DIR="db dumps"
```

```
DB_DIR="postgresql/data"
```

```
TARGET_DIR=$DB_BACKUP_DIR
```

```
echo "Cleaning old database backups from $TARGET_DIR"
```

```
rm -rf $TARGET_DIR/*
```

quoting variables

```
rm -rf $TARGET_DIR/*
```

TARGET_DIR="db dumps", this becomes:

```
rm -rf db dumps/*
```

And now you've run two separate commands:

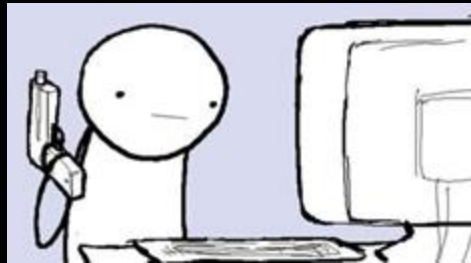
```
rm -rf db  
rm -rf dumps/*
```

The fix is simple:

cleanup.sh

```
TARGET_DIR="db dumps"  
rm -rf "$TARGET_DIR"/*
```

no dependency check!



cleanup.sh

...

...

PG_CTL="/usr/bin/pg_ctl_default"

...

\$PG_CTL stop -D "\$DB_DIR" -m fast 2>/dev/null || echo "[WARN] pg_ctl failed (ignored)"

echo "Remove DB Dumps, as backup is complete!"

...

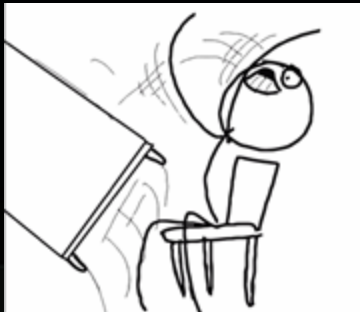
check for dependency

cleanup.sh

```
require() {  
    command -v "$1" >/dev/null 2>&1 || {  
        echo "Missing dependency: $1" >&2  
        exit 1  
    }  
}
```

```
require pg_ctl_default
```

it still fails silently!



`cleanup.sh`

...

```
BACKUP="./var/backups/postgres/backup today.tar.gz"
```

```
DB_DIR="./var/lib/postgresql/12/main"
```

```
TARGT=$BACKUP
```

```
echo "[INFO] Attempting graceful DB stop (best-effort)..."
```

```
pg_ctl stop -D "$DB_DIR" -m fast 2>/dev/null || echo "[WARN] pg_ctl stop failed (ignored)"
```

```
echo "[INFO] Running backup integrity pipeline: gzip -t $BACKUP | awk 'END{print \"ok\"}'"
```

```
gzip -t $BACKUP | awk 'END{print "ok"}'
```

```
echo "[INFO] Pipeline exit status (reported): $?"
```

...

Use Strict Mode

Shell doesn't assume anything is dangerous. You probably should.

Strict mode gives you a safety net:

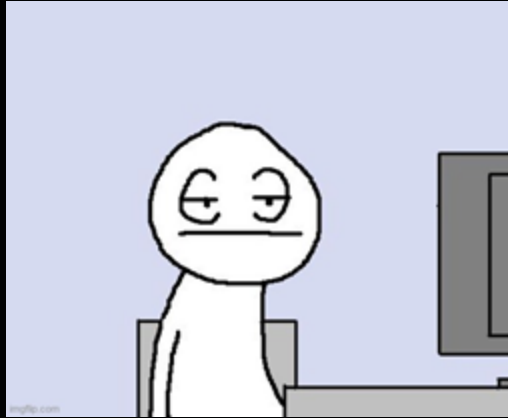
```
set -euo pipefail
```

-e: Exit immediately if any command fails

-u: Error on using unset variables

-o pipefail: Fail if any command in a pipeline fails

now, let's talk some other practices!



add a debug mode

Sometimes you want to see everything. Sometimes you don't.

A debug mode lets you toggle verbose output without editing your script every time.

add a progress loader

Because silence feels like failure.

Long-running commands can make users wonder if the script hung or crashed. A simple loader adds just enough feedback to show that something's happening without cluttering the terminal.

It's a small **UX upgrade**, especially in scripts that handle provisioning, backups, or large data transfers.

No output doesn't have to mean no activity.

create a resumable script

Because rerunning the whole thing shouldn't feel like starting over.

cleanup.sh

```
if [ ! -f /tmp/setup.step1.done ]; then
    echo "Running step 1..."
    # some long-running command
    touch /tmp/setup.step1.done
fi
```

If your script processes a long input file - say, a list of hosts or user IDs - resumability means not starting from the top again. Instead of deleting lines from the input file, a cleaner pattern is to track progress in a separate file.

clean up after your mess - use Trap

Because your script should clean up after itself, even when it crashes.

If your script creates temporary files, background jobs, or mounts anything, it should also clean up - no matter how it exits. That's where trap comes in.

The `trap` builtin lets you register a clean-up function that runs on `EXIT`, `INT`, or `ERR`.

Trap isn't just for temp files. Use it to:

- Kill background jobs
- Unmount things
- Stop services
- Log exit status

add meaning to your exits

Because not all failures are created equal.

Every script exits with a status code. By default, 0 means success, and anything else means failure. But if you're building scripts for automation, chaining, or CI/CD, you should make exit codes meaningful.

Instead of a generic exit 1,
define what each failure means:

cleanup.sh

```
EXIT_OK=0  
EXIT_USAGE=64  
EXIT_DEPENDENCY=65  
EXIT_RUNTIME=66
```

feeling cheated?



[Home](#) » [Posts](#)

Production Grade Bash Scripts

A deep dive into writing resilient, production-grade shell scripts, because sometimes your one-liner ends up running in prod.

April 16, 2025 · 11 min · Kumar Ashwin



Production Grade Bash Scripts

For scripts that accidentally matter

For scripts that accidentally matter

krash.dev/posts/writing-production-grade-bash-script/

thank you!

you can find me at
kumarashwin.com

