How To Setup an Apache or Nginx “Tar Pit” to Block Unwanted Hackers and Rogue AI “The Easy Way”

Apache Volcano burst guard capture

2026-11-25

Stephen

Paul

Capture started. Send `<endcapture>` when you’re ready for the raw Markdown export.

Stephen

Okay, so I’ve done this before, but for the life of me, I can’t find the code now. What I would like to do is, basically once a minute, I need a diff of what the old Apache log contained, and what the changes are since the last time. Basically, I want it to parse. I want it to— what I need to do is I want to know if one unique IP, any particular IP, has made more than, say, 100 requests. And if it does, then I’m going to do a route IP to loopback address, right? And then once a minute, I’ll clear the route. So basically, if somebody comes in and they abuse it, they try to hack it, you know, send a bunch of JSON and, you know, a bunch of crap WordPress, that if they send more than 100 queries in that last minute, that it will route them to a loopback address so that they can’t route there. And then after a minute, it restores the route. What do you think? Can you help me with that?

Paul

I’ll check the Linux and Apache details, then put together a script that reads new log entries and temporarily blocks IPs that cross your threshold.

Paul

The script now counts newly appended entries in 60-second windows and defaults to a dry run. I'll test the 100-versus-101 cutoff, log rotation, and the firewall commands it generates.

Paul

Yes. For Apache Volcano, I'd use a **60-second nftables block** instead of a route to loopback. A route change affects all traffic back to that IP. The firewall rule can drop only incoming TCP traffic on ports 80 and 443, and each block expires automatically without a cleanup job. genui {"citation":{"refs":["turn281251373search0","turn768435616search0"]}}

Download apache-burst-guard.py

The script starts at the end of your access log, follows newly appended entries through normal log rotation, and counts requests per IPv4 or IPv6 address. At the end of each fixed 60-second window, **101 or more requests** from an IP triggers a report or a 60-second block. Every logged request counts, including repeated URLs and failed WordPress probes. A burst split across two windows can stay below the cutoff in both. genui {"citation":{"ref":"turn428282777search1"}}

Start with a dry run. Substitute the Quantumgod vhost access log if it has its own file. The script expects the IP in the first field.

```
sudo python3 ./apache-burst-guard.py /var/log/apache2/access.log
```

An IP over the limit produces a line like `192.0.2.101 137 requests in 60s: would block`. You can exempt an address or network by adding `--allow IP_OR_CIDR` for each one.

For unattended enforcement, install the script and save the unit below as `/etc/systemd/system/apache-burst-guard.service`. Use `--block` only when the IP in the log is also the packet source IP the server sees. If a CDN or reverse proxy supplies the visitor IP, enforce the limit at that layer. Apache can log a forwarded visitor IP even though the connection comes from the proxy. genui {"citation":{"refs":["turn281251373search2","turn281251373search4"]}}

```
sudo install -o root -g root -m 0755 \
  ./apache-burst-guard.py /usr/local/sbin/apache-burst-guard
```

```
[Unit]
Description=Apache per-IP request burst guard
After=apache2.service

[Service]
ExecStart=/usr/bin/python3 /usr/local/sbin/apache-burst-guard /var/log/apache2/access.log --
block
```

```
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload
sudo systemctl enable --now apache-burst-guard.service
sudo journalctl -u apache-burst-guard.service -f
```

`sudo nft list table inet apache_burst_guard` shows active blocks and their remaining time. To stop enforcement and lift all current blocks immediately, stop the service and run `sudo nft delete table inet apache_burst_guard`.

I tested the 100-versus-101 cutoff, log rotation, IPv4 and IPv6 counting, and exemptions. I verified the generated firewall commands with a mock. Live nftables enforcement on your server is still unverified.

Stephen

System load prior to tarpitting since evidently somebody (or something ahem, rogue AI) is hammer mysql

Stephen

```
top - 09:00:18 up  1:35,  2 users,  load average: 3.70, 2.55, 1.83
Tasks: 130 total,   6 running, 124 sleeping,   0 stopped,   0 zombie
%Cpu(s): 29.2 us, 67.8 sy,  0.0 ni,  0.0 id,  1.3 wa,  0.0 hi,  1.3 si,  0.3 st
MiB Mem :    956.3 total,    285.5 free,    330.9 used,    392.5 buff/cache
MiB Swap:    496.0 total,      0.2 free,    495.8 used.    625.4 avail Mem
```

Paul

I'll read this as the pre-tarpit baseline and identify what to measure next to see whether MySQL or the kernel is driving the load.

Paul

Pre-tarpit baseline at 09:00:18 — load averages **3.70 / 2.55 / 1.83**, with **0% CPU idle** and **67.8% in kernel time**. The spike was using all available CPU during that sample, with most of the time spent in the kernel. The `top` header alone can't tell us whether MySQL, Apache/PHP, or another process was driving it.

Swap was almost full, but Linux still reported **625 MiB available RAM**. To see whether paging is contributing *now*, watch `si` and `so` in `vmstat 1 10` and skip the first row, which averages activity since boot. genui {“citation”:{“ref”:“turn239867926search4”}}

While it’s spiking, I’d grab these three read-only snapshots:

```
top -b -d 1 -n 2 -o %CPU | head -n 45
vmstat 1 10
sudo mysql -e 'SHOW FULL PROCESSLIST\G'
```

The process rows show who’s using CPU, and MySQL’s process list shows what its sessions are doing and where they connect from. If busy queries mostly come from the local BookStack account and coincide with an Apache request spike, that supports the web-traffic explanation. Repeat the same checks after the tarpit kicks in and compare **`%sy`****, `%id`, and active queries** first. The load averages will take longer to reflect the change. genui {“citation”:{“ref”:“turn239867926search0”}}

Stephen

```
root@deepdal00:~# systemctl stop apache2
root@deepdal00:~# top -b -d 1 -n 2 -o %CPU | head -n 45
vmstat 1 10
sudo mysql -e 'SHOW FULL PROCESSLIST\G'
top - 09:07:19 up  1:43,  2 users,  load average: 2.21, 2.40, 1.93
Tasks: 123 total,   1 running, 122 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us, 12.5 sy,  0.0 ni, 62.5 id, 25.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    956.3 total,    290.2 free,    290.2 used,    415.7 buff/cache
MiB Swap:    496.0 total,      1.7 free,    494.3 used.    666.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0   25332   8200   4696 S   0.0   0.8   0:06.79 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_wo+
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     10 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     12 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     13 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     14 root      20   0       0      0      0 S   0.0   0.0   0:37.09 ksoftir+
     15 root      20   0       0      0      0 I   0.0   0.0   0:02.08 rcu_pre+
     16 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_exp+
```

```
     17 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_exp+
     18 root      rt   0       0      0      0 S   0.0   0.0   0:00.02 migrati+
     19 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 kprobe-+
     20 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_in+
     21 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     22 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kdevtmp+
     23 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     24 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tas+
     25 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tas+
     26 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kauditd
     27 root      20   0       0      0      0 S   0.0   0.0   0:00.41 pr/ttyS0
     28 root      20   0       0      0      0 S   0.0   0.0   0:00.00 khungta+
     29 root      20   0       0      0      0 S   0.0   0.0   0:00.00 oom_rea+
     32 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     33 root      20   0       0      0      0 S   0.0   0.0   0:00.43 kcompac+
     34 root      25   5       0      0      0 S   0.0   0.0   0:00.00 ksmd
     35 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     36 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     37 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     38 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 irq/9-a+
     39 root      20   0       0      0      0 I   0.0   0.0   0:00.00 kworker+
     40 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     41 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     42 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     43 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
procs -----------memory---------- ---swap-- -----io---- -system-- -------cpu-------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st gu
 1  0 506128 296688 128816 297508   59   84  3225   187 1100   13 14 18 63  5  0  0
 0  0 506128 296688 128816 297572    0    0    64     0  440  442  1  1 98  0  0  0
 0  0 506128 296688 128824 297572    0    0     0    20  410  397  0  1 99  0  0  0
 0  0 506128 296688 128824 297572    0    0     0     0  409  398  1  0 99  0  0  0
 0  0 506124 296688 128824 298600    0    0  1028     0  456  450  1  1 97  1  0  0
 0  0 506124 296688 128824 298600    0    0     0     0  415  397  0  1 99  0  0  0
 0  0 506124 296688 128824 298600    0    0     0     0  405  398  0  0 100  0  0  0
 0  0 506124 297028 128824 298600    0    0     0     0  420  410  1  1 98  0  0  0
^C
*************************** 1. row ***************************
     Id: 5
   User: event_scheduler
   Host: localhost
```

```
     db: NULL
Command: Daemon
   Time: 6136
  State: Waiting on empty queue
   Info: NULL
*************************** 2. row ***************************
     Id: 9955
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: init
   Info: SHOW FULL PROCESSLIST
root@deepdal00:~#
```

Paul

Stopping Apache brought `deepdal00` back to 97–100% CPU idle. In the one-second `vmstat` samples afterward, I/O wait stayed at 0–1% and swap-in and swap-out were both zero. MySQL's process list showed only the idle event scheduler and your diagnostic query. The nearly full swap was already occupied, with no active paging during that sample. genui {"citation":{"refs":["turn239867926search4","turn239867926search0"]}}

That makes Apache or work triggered by web requests the strongest lead. MySQL was quiet with Apache stopped. We still need a sample during the spike to tell whether PHP was driving MySQL hard or the work was elsewhere in the web stack. This gives you an **Apache-off baseline** to compare against when the site is running with the guard enabled.

You can check whether the guard's threshold would have caught the earlier spike without bringing Apache back up. On the Quantumgod vhost log, this ranks **requests per minute per IP** from 09:00 through 09:06. Use the time shown in Apache's log if it differs from the server console.

```
awk '$4 ~ /^\[25\/Nov\/2026:09:0[0-6]:/ {print substr($4,14,5), $1}' \
  /var/log/apache2/access.log |
  sort | uniq -c | sort -nr | head -20
```

If the highest counts are all **100 or below**, the spike may have been spread across IPs or driven by a smaller number of expensive requests. One correction for the next live sample. My earlier `head -n 45` command cut off the second `top` refresh. Use `top -d 1 -o %CPU` and grab the process rows after it refreshes.

Stephen

Good point, Paul. Those netcat commands and the nohup aren’t gonna do a darn thing, a bit of good if Apache’s not running. So I just restarted Apache, and I’m gonna watch it load, see if it goes up to two or three, and then I’ll rerun the command and send you the output, Paul.

Paul

Sounds good, Stephen. When it spikes again, grab `SHOW FULL PROCESSLIST` first, while the queries are still active. Then send the top few CPU process rows from `top` and the interval rows from `vmstat`.

We’ve got a clean comparison now. With Apache stopped, the box was **97–100% idle** and MySQL had no application queries running. If CPU idle drops sharply after the restart, grab the samples right then. The load average can take a while to catch up.

Stephen

How do I disable dry-run mode?

Paul

Add `--block` to the command. If the dry-run copy is still running in the foreground, stop it with `Ctrl+C`, then run:

```
sudo python3 ./apache-burst-guard.py /var/log/apache2/access.log --block
```

Use your Quantumgod vhost log path if it’s different. The startup line should say `(enforcing)`. The defaults are still **more than 100 requests per IP in a 60-second window**, followed by a **60-second block**.

If you’re using the systemd unit from earlier, it already includes `--block`.

Stephen

```
root@deepdal00:/var/log/apache2# for i in $(cat /tmp/ips.txt); do echo $i; grep $i access.log
| wc -l; done
[redacted]
```

Paul

`50.116.22.169`** is the standout, Stephen.** Your loop found **40,992** matching lines, compared with **269** for the next IP. I’d check whether `50.116.22.169` belongs to one of your own VMs, a reverse proxy, or a monitor before blocking it.

`grep $i` treats the dots as regex wildcards and can match anywhere on the line. Assuming the first field of `access.log` is the client IP, this counts exact matches and shows the busiest minutes:

```
awk -v ip='50.116.22.169' '$1 == ip {print substr($4, 2, 17)}' access.log |
  sort | uniq -c | sort -nr | head -15
```

Then check what it was hitting. This ranks the method, path, and response status, with query strings removed:

```
awk -v ip='50.116.22.169' '
  $1 == ip {sub(/\?.*/, "", $7); print $6, $7, $9}
' access.log | sort | uniq -c | sort -nr | head -20
```

If the busiest minutes overlap the **09:00 load spike**, the path and status breakdown should tell us where to look next. Repeated BookStack page requests make PHP and MySQL worth checking. Static assets, redirects, or repeated probes point toward a different explanation. The **40,992** total covers the whole log, so the per-minute counts are what tell you whether the **101-request** threshold would have caught it.

Stephen

```
root@deepdal00:/var/log/apache2# dig -x 50.116.22.169

; <<>> DiG 9.20.24-1ubuntu0.5-Ubuntu <<>> -x 50.116.22.169
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15570
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;169.22.116.50.in-addr.arpa.    IN      PTR

;; ANSWER SECTION:
169.22.116.50.in-addr.arpa. 300 IN      PTR     50-116-22-169.ip.linodeusercontent.com.

;; Query time: 36 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Wed Nov 25 09:21:02 CST 2026
;; MSG SIZE  rcvd: 107

root@deepdal00:/var/log/apache2#
```

Paul

That PTR looks like a Linode hostname. It gives us a hosting-provider clue, but it doesn't tell us who controls the VM or what made those requests. Since you run Linode VMs yourself, I'd compare `50.116.22.169` against your instance IPs before blocking it. On `deepdal00`, `ip -br addr` will tell you whether it's this box's own address.

The next useful check is what that IP requested **during the spike**. This shows the most repeated paths and their response codes from 09:00 through 09:06, using the timestamps in Apache's log:

```
awk '$1 == "50.116.22.169" && $4 ~ /^\[25\/Nov\/2026:09:0[0-6]:/ {
    sub(/\?.*/, "", $7)
    print $6, $7, $9
}' access.log | sort | uniq -c | sort -nr | head -20
```

If it comes back empty, those **40,992** matches may belong to a different time window. If one BookStack path dominates, we have a specific request to correlate with MySQL activity the next time load climbs.

Stephen

So it's possible that it's one of mine, I'll have to login to the Akamai/Linode port and search for that IP under Linodes. I can check that here in a bit

Paul

Yep, Stephen. Search for `50.116.22.169` under **Linodes** in Cloud Manager. If it matches one of yours, check what that VM is doing before deciding whether to exempt it. A proxy could account for traffic from many visitors, while a monitor or script could be making repeated requests on its own.

Stephen

```
root@deepdal00:/usr/local/sbin# apache-burst-guard.py /var/log/apache2/access.log --block
Watching /var/log/apache2/access.log (enforcing)
50.116.22.169 201 requests in 60s: blocked for up to 60s please hold
```

Paul

Holding, Stephen. The guard reported a 60-second block for `50.116.22.169` after counting 201 requests in one window.

Stephen

```
root@deepdal00:/var/log/apache2# ip r
default via 50.116.22.1 dev eth0 proto static
50.116.22.0/24 dev eth0 proto kernel scope link src 50.116.22.169
root@deepdal00:/var/log/apache2#
```

Paul

Confirmed, Stephen. `50.116.22.169` is `deepdal00`'s own IPv4 address. The route's `src 50.116.22.169` is the source address Linux prefers on `eth0`.

Stop the foreground guard with `Ctrl+C`, then restart it with that address exempted:

```
apache-burst-guard.py /var/log/apache2/access.log --block --allow 50.116.22.169
```

The existing block will expire on its own. To clear it immediately if it's still active:

```
nft 'delete element inet apache_burst_guard blocked_v4 { 50.116.22.169 }'
```

The **201 requests in a minute** still matter. The next thing to check is whether Apache logged the server's address because a local job was calling the site or because a proxy was forwarding requests. The paths and user agents on those log entries are the best clues you already have.

Stephen

apache-burst-guard.py /var/log/apache2/access.log –block –allow 50.116.22.169 completed, just monitoring for the moment. Load has returned to < 1 levels □□

Paul

Good, Stephen. **Load is back below 1**, and the guard now exempts `deepdal00`'s own IP. If load climbs again, check whether requests logged under `50.116.22.169` climb at the same time. That would give us a better lead on where those 201 requests came from.

Stephen

Revision #7
Created 2026-11-25 15:02:55 UTC by LoveSouls
Updated 2026-11-25 15:41:32 UTC by LoveSouls