

FIELD NOTES FROM A PRESALES SOLUTIONS ARCHITECT

The Outage That Wasn't a DDoS

A presales war story about a website that kept dying while the control panel stayed perfectly healthy — and the two numbers in the logs that nobody had compared.

Daanial Farrukh · 19 October 2026 · <https://quabin.com/outage-that-was-not-a-ddos-scrapers-php-fpm/>

The message came in the same way these messages always do:

"The site is down again. Webmail too. We think it's a DDoS."

Then, a minute later: "But WHM is working fine."

I asked one question: "How many PHP processes is the site allowed to run?"

Silence. Then: "What does that have to do with a DDoS?"

Everything, as it turned out. The site was a busy public portal on a cPanel server. Twice in two days it had become unreachable for real users for twenty to thirty minutes, then recovered on its own. Webmail on the same box died with it. The control panel on port 2087 never blinked. The team had already restarted Apache, checked the firewall, and started drafting a request for DDoS protection.

That's the trap. **An outage with no attacker, no error and no crash looks exactly like an attack.** It isn't. It hides in plain sight, in two numbers that live in different files.

Why "WHM Works" Tells You Nothing

The first thing I corrected was the assumption that the server was healthy because the control panel loaded.

WHM is served by `cpsrvd`, a separate daemon. It does not pass through Apache at all. So on a cPanel box, WHM on port 2087 being fast while the website on port 443 is dead means precisely one thing: **the problem is inside the web stack, and the server itself is fine.** That is a useful clue, not a reassurance.

Webmail was the second clue. On this server, webmail is proxied through Apache. When Apache has no free worker, webmail dies along with the site. Two symptoms, one bottleneck.

Absence of Errors Is Not Health

The team's strongest argument for "it must be an attack" was the error log. It was empty. Apache had not crashed, had not restarted, and had logged nothing during either incident.

```
tail -20 /usr/local/apache/logs/error_log
```

Nothing. And that is expected. **Saturation is not an error.** When every worker is busy, Apache does not fail — it waits. New connections queue. Users see a spinner, then a timeout. Nothing in that sequence is an error from Apache's point of view, so nothing gets logged.

If you only look for errors, you will never find a capacity problem. You have to look at the traffic.

Step 1: Count Requests Per Minute

The access log for the domain is where the story lives. On cPanel the HTTPS traffic is in the `-ssl_log` file, not the plain one.

```
awk '{print $4}' /usr/local/apache/domlogs/example.com-ssl_log | cut -d: -f2,3 | sort | uniq -c  
| tail -30
```

That prints a request count for every minute of the day. The outage window stood out immediately: a flat line of roughly 700 requests a minute, every minute, for twenty minutes. Then it fell to 20 in a single minute, and the site came back.

Here is the detail that settled the argument about causation: **real traffic reappeared in the same minute the flood stopped**. Images, scripts, the homepage, the first webmail login — all of it returned at the moment the flood ended, not one minute earlier. Whatever that traffic was, it was locking users out.

Step 2: Find Out What the Traffic Was Asking For

```
grep '10:1[0-9]:' /usr/local/apache/domlogs/example.com-ssl_log | awk '{print $7}' | sort |  
uniq -c | sort -rn | head
```

Every one of the 7,000 requests in that window — 100 percent — hit the same page: the public listing page of the portal. And every URL was unique. They looked like this:

```
/page-listing?list=36?list=42?list=42?list=42?list=42
```

A valid URL can contain only one `?`. Anything after the first one is the query string. These URLs had a second, a third, a tenth. They were malformed, and they were growing: each request was one `?list=` longer than the previous one.

That pattern has only one explanation. A crawler was taking each category link on the page and appending it to the address it was already on, instead of replacing the query string. Every page it loaded minted new links, every new link was unique, and the crawl never ended.

I want to be precise about who was at fault here, because the team's first instinct was to send a developer to "fix the links". **The site's links were correct**. The category links were clean absolute URLs. The click handler on the listing rows only filled a subscribe form; the navigation line in it was commented out and never ran. The concatenation was happening inside the crawler. The site's only failing was answering a malformed URL with 200 OK, which tells any crawler "valid, keep going".

Step 3: Rule Out the DDoS Properly

A DDoS is a volume problem from many sources. So I checked the sources.

```
grep '10:1[0-9]:' /usr/local/apache/domlogs/example.com-ssl_log | awk '{print $1}' | sort |  
uniq -c | sort -rn | head -5
```

The busiest IP address in the entire window had made three requests. Three. The load came from more than two thousand addresses in datacenter and proxy ranges, each making two or three requests. By volume per IP, nothing was happening. No firewall rate limit on earth would have triggered.

Then I checked the user agent.

```
grep '10:1[0-9]:' /usr/local/apache/domlogs/example.com-ssl_log | grep -o '"Mozilla[^\"]*"' |  
sort | uniq -c | sort -rn | head -3
```

7,024 of 7,027 requests shared a single string — one specific desktop Chrome build on one specific macOS version. Two thousand "different visitors" with an identical browser fingerprint is not two thousand visitors. It is one scraping operation rotating through a proxy pool, dressed as a browser.

Two more things confirmed it was a tool, not a crowd. Both incidents sat in a tight band of 676 to 718 requests per minute, every minute — organic crawling is never that even. And the tool did not identify itself as a bot, which means robots.txt was never going to help.

So: not a DDoS. One client, rate-limited by its own configuration, requesting about 11.7 pages per second.

Step 4: The Number Nobody Had Compared

Here is the question the team could not answer in that first message, and it is the whole story.

```
grep -E 'pm.max_children|pm.max_requests' /opt/cpanel/ea-php81/root/etc/php-fpm.d/example.com.  
conf
```

```
pm.max_children = 5  
pm.max_requests = 20
```

Five. The entire site was allowed five concurrent PHP processes. At roughly half a second per page, that is a hard ceiling of about ten PHP requests per second.

The scraper was asking for 11.7 PHP requests per second. More than 100 percent of the site's capacity, sustained for twenty minutes.

Now the chain is obvious. All five PHP-FPM slots fill. The next request waits for a slot. While it waits, it holds an Apache worker. Apache has 150 workers. Within a minute they are all holding a request that is waiting for PHP. Every connection through Apache — the site, the images, the webmail proxy — stalls. A few real requests trickle through when a slot happens to free up at the right moment, which is why the outage was severe but never quite total.

Apache never crashed because Apache was never the bottleneck. It was the waiting room.

Why 20 Percent More Traffic Took the Site Down

This is the part that genuinely surprised the team. The site normally served around 35,000 requests an hour at peak. The scraper added about 42,000 an hour. More, yes, but not ten times more. How does a modest increase cause a total outage?

Because the two numbers are not the same kind of request.

Normal traffic is mostly images, CSS and JavaScript. Apache serves those from disk without touching PHP. A busy hour of real visitors might generate only a few PHP requests per second. The scraper's traffic was **100 percent PHP**, and because every URL was unique, nothing was ever served from cache. Every single hit ran the application from scratch.

Compare PHP request rates, not total request rates. Total requests per hour is the number everyone quotes and the number that tells you nothing about PHP capacity.

A warning if you go looking for this on your own server: measure during the flood, not after. When I ran the listing page on the idle server it answered in half a second and looked perfectly healthy. The pages were never slow to process. They were waiting for a slot that never opened.

How to Actually Fix This

There were two independent problems. The scraper alone would have been harmless if the page did not mint infinite URLs. The infinite URLs alone would have been harmless without the scraper. Fixing either one breaks the outage. Fix both.

Step 1: Reject malformed URLs at the door.

Any request whose query string contains a second `?` is malformed by definition, so refusing it carries no risk to legitimate traffic. Two lines in the site's `.htaccess`:

```
RewriteEngine On
RewriteCond %{QUERY_STRING} \?
RewriteRule ^ - [F]
```

Once the server answers 403 instead of 200, the crawler has nothing new to follow, and the loop collapses on its own. No code change on the page is needed, because the page was never broken.

Verify it before you trust it:

```
curl -s -o /dev/null -w '%{http_code}\n' 'https://example.com/page-listing?list=36?list=42'
```

You want 403. Then run the same check against the clean URL with one `?` and confirm it still returns 200.

Step 2: Give PHP a realistic ceiling.

Five children is far too low for a site doing 35,000 requests an hour. Any genuine burst of real visitors would have caused the same outage with no scraper in sight. The server had almost 12 GB of RAM and about 9 GB free; one FPM child measured 16 MB idle, so budgeting 50 MB each leaves room for about 30. In WHM that is MultiPHP Manager > PHP-FPM pool options for the domain, or the same file we read above.

```
pm.max_children = 30
```

This change is about capacity, not about the scraper. At 30 children the ceiling rises from about 10 to about 60 PHP requests per second, so this particular scraper would no longer cause an outage — but a faster one still would. Step 1 is what stops the loop. Step 2 is what stops the next real traffic spike from looking like one.

Change `pm.max_requests` separately, later. The cPanel default of 20 makes every child respawn constantly; 500 is saner. Keep the two changes apart so you can tell which one did what.

Step 3: Put something back in front of the origin.

This domain had been switched to DNS-only on Cloudflare a few weeks earlier to fix a different performance problem, which quietly turned off the edge rate limiting and put the origin IP in public DNS. The scraper landed directly on the server with nothing in between. Whatever you use in front — Cloudflare, a load balancer, a WAF — make sure a per-client rate limit is actually active after every DNS change, not just the day it was set up.

The Log Gotchas That Cost an Hour

- Webmail requests on this server were in the main domain's `-ssl_log`, not in the webmail subdomain's log, which had been empty for months. Identify webmail hits by `roundcube` or `cpsess` in the URL.
- The live log is rotated around 17:00 and does not hold earlier days. A count of zero for an older window usually means rotation, not an outage. Confirm which hours a file covers before calling a gap real.
- Apache prints a "reached MaxRequestWorkers" line once per event, not once per minute. Grepping a single minute of the error log proves nothing. Tail it and read the timeline.

What I Tell Every Customer Now

Before anyone writes "DDoS" in an incident report, I ask one question:

"What is the busiest single IP in the outage window, and how many PHP workers is the site allowed?"

If the busiest IP made three requests and the answer to the second half is a single digit, it was not an attack. It was a ceiling. And the fix is two lines in a config file, not a procurement request.

I write about the questions I ask in real customer rooms — the ones that change architectures and surface the problems no one else noticed. If you found this useful, drop your email below and I'll send the next one when it's ready.

FAQs

Was it really not a DDoS? It took the site down.

No. A DDoS is a volume attack from many sources, and the firewall sees it as high request rates per IP. Here the busiest IP made three requests in twenty minutes. The outage came from one rate-limited scraper requesting about 11.7 PHP pages per second against a site that could process about 10. That is a capacity ceiling, not an attack.

Why did WHM keep working while the website was down?

WHM runs on its own daemon, `cpsrvd`, on port 2087, and does not pass through Apache. When Apache's workers are all busy waiting for PHP, the website and anything proxied through Apache (such as webmail) stall, but WHM is unaffected. "WHM works" only tells you the server is up, not that the web

stack is healthy.

Why was there nothing in the Apache error log?

Because nothing failed. When all workers are busy, Apache queues new connections rather than erroring. Saturation is silent. You find it in the access log by counting requests per minute and by looking at what those requests asked for, not in the error log.

Should the developer fix the page so the URLs stop growing?

Not in this case. The page's links were correct absolute URLs; the concatenation happened inside the crawler. The one thing the site should change is to answer a URL with a second question mark with 403 instead of 200, which a two-line rewrite rule does. That removes the signal that keeps the crawler looping.

What is a sensible pm.max_children value?

Measure one idle PHP-FPM child's memory on your server, budget about three times that per child to allow for real page loads, and divide the RAM you are willing to give PHP by that figure. On this server, 16 MB idle, a 50 MB budget and about 9 GB free pointed to roughly 30. Then confirm under real load, not on an idle box.

How do I know if my server is close to this ceiling right now?

Count PHP requests per second during your real peak, not total requests, and compare it with pm.max_children divided by your average PHP response time in seconds. If the peak PHP rate is more than about half of that ceiling, you are one scraper or one marketing campaign away from an outage.

Read online: <https://quabin.com/outage-that-was-not-a-ddos-scraper-php-fpm/>