Blind Spots

WordPress Security From the Field

JUNE 2026 EDITION

What a fleet of monitored WordPress sites reveals that no single site can see.

A field report · logystera.com

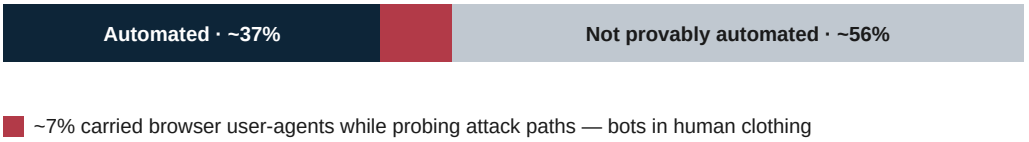
Produced from continuous observation of WordPress sites with Logystera. Site names, owners, addresses, and attacker network details have been removed; the mechanics, proportions, and timings are unchanged, because they're the evidence.

What this is, and what it isn't

This is not a survey. We did not crawl a million sites or buy a threat feed. This report comes from a fleet of WordPress sites that we watched continuously through the spring of 2026 — more than four million requests and background events, observed in place, over months rather than sampled in a single scan. Small businesses, a nonprofit, a couple of personal sites, our own. Real sites with real traffic and real admins who had, in almost every case, already done the responsible thing: installed a security plugin, put a WAF in front of the login page, locked down the endpoints the blog posts told them to lock down.

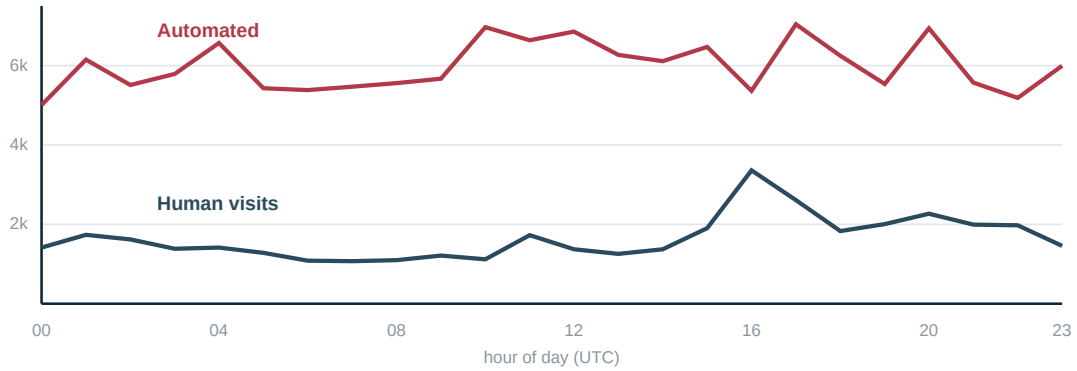
Start with a number you'd think would be simple: how much of this traffic is a person? It isn't simple — and that's the first finding. By user-agent, the cheap and honest signal, only about **a third of requests were flagged as automated**. But the user-agent is also the easiest thing in a request to fake, and attackers do: we logged **tens of thousands of requests carrying ordinary `Chrome` and `Firefox` strings while they probed for `.env` files, login forms, and `xmlrpc`** — bots, plainly, sitting in the "human" column. Count those, plus the scanners quietly walking dead 404 paths, and automated traffic is closer to **half of everything**. The share you can confidently call human is the number you can trust least.

> **How much of this traffic is a person?** Less knowable than it looks. By user-agent, about a third of requests were automated — a floor, because the cheapest disguise is a browser user-agent. We caught another tranche wearing ordinary browser strings while probing for login forms, `.env` files, and `xmlrpc`. The grey remainder is humans *and* the bots good enough to blend in with them — there is no clean line between the two.



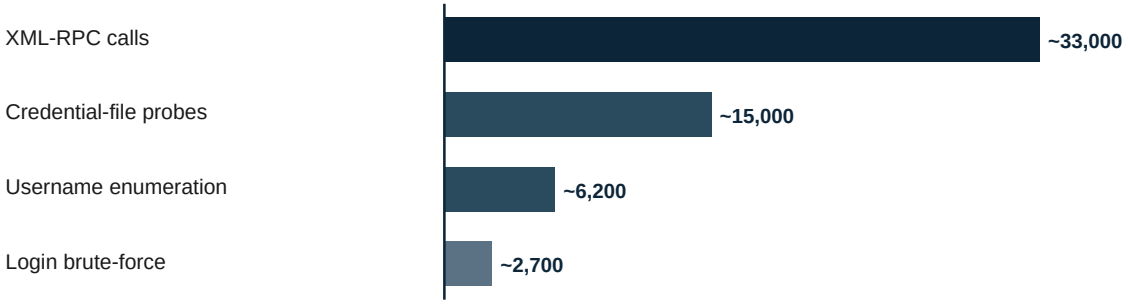
And it doesn't keep your hours. Split the same traffic by time of day and the two populations pull apart: human visits rise and fall on a daily curve — quiet before dawn, peaking late afternoon — while the automated traffic barely moves, holding a near-flat line around the clock. At every hour the machines outnumber the people, and before dawn they do it by roughly five to one. Your visitors sleep; the things probing your site do not.

> **Your visitors sleep. The probes don't.** Requests by hour of day (UTC) across the fleet. Human page-views (lower line) rise and fall on a daily curve; automated traffic (upper line) holds a near-flat line around the clock — and outnumbers humans in every single hour.



Whatever the exact split, the hostile share isn't random. Over a single month, four kinds of request account for most of the attack traffic — and notice that login brute-force, the attack everyone pictures, is the *smallest* of them. The cheap reconnaissance comes first.

> **What the hostile traffic is doing.** Attack requests across the monitored fleet over one month, by type. XML-RPC calls and credential-file scanning (`.env`, `wp-config`, `.git`) dominate; the brute-force login attempts everyone worries about are the smallest category.



We're not going to pretend these sites speak for all of WordPress — they speak for what we watched closely, and that's the point. What watching a *fleet* buys you, rather than a single site, is a pair of vantage points no individual admin has. The first is **horizontal**: from inside your own site you see the attacks aimed at you — never the campaign sweeping a hundred sites that you happen to be number forty in. The second is **vertical**: you see your security dashboard report "handled," never what your defense actually *did* when the attack arrived.

This report is about both blind spots — the campaign you can't see across, and the defense you can't see through.

Attackers run lists, not targets

Here is the thing a single site can never show you. Across the fleet, **more than nine thousand distinct addresses** sent attack traffic — login probes, username enumeration, `xmlrpc` calls, credential-file scans. Most hit one site and moved on. But **more than 1,100 of them attacked more than one of these sites — and close to 500 hit three or more.** The sites that share these attackers share nothing else: different owners, different hosting companies, different countries, no link between them but the software they run.

From inside any one of those sites, that overlap is invisible. The admin sees a handful of failed logins on a Tuesday and reasonably files it under background noise. They cannot see that the same address spent the previous fortnight working through a string of other unrelated sites, because they can only see their own.

> **The same attackers, across unrelated sites.** Of the distinct addresses that sent attack traffic, how many hit more than one of the sites we watch — sites with no shared owner, host, or network. A single-site view counts each of these as one quiet incident.

The numbers undercount, if anything — they only include addresses we could attribute, and they stop at the edge of our fleet. The behaviour behind them is what matters, and it has a shape:

- **The methodical sweep.** One address spent a month working through site after site, five to ten days apart, and at each new one it re-probed for *that site's* specific plugins — checking which membership, e-commerce, or community plugins were installed — before it tried a single login. To each site's admin that's one slow afternoon of odd requests. Seen across all of them, it's a tool walking a list and adapting to each target.
- **The cross-host credential run.** Another address ran a six-week login-guessing campaign that moved between sites on entirely different hosting providers, in different countries, without a pause in method. Hosting boundaries that feel like walls from the inside are nothing to a script working an address list.
- **Escalation when nothing pushes back.** One site went from a few dozen login attempts from a handful of addresses to nearly four hundred from over 150 in six days — all against a single username. Nothing resisted, so the botnet scaled. The escalation curve is only legible if you're watching the username, not the IPs, because the username is the constant and the addresses are disposable.
- **Targeting tuned to you.** We watched a campaign guess usernames from each site's *own domain name* — not a dictionary, not `admin`, but a string built from the target — and rotate it across two dozen addresses so no single one ever tripped a per-IP rate limit. The countermeasure most sites rely on was anticipated and designed around.

None of this is exotic, and none of it is news to the attackers. It's only news to the defenders, because the defender's vantage point — one site, its own logs — is structurally the wrong one for seeing it. The one thing you *can* do without the cross-site view is assume it exists: treat the address that probed you once as one stop on a list, and size your defenses — lockouts, rate limits, a hidden login URL — for the botnet behind it rather than the single request in your log. Beyond that, there is no `curl` command at the end of this section, and that is the point — the campaign itself is the one thing you cannot see for yourself.

The door you locked, and the door you didn't

The reconnaissance those sweeps run starts with a simple question — *who has an account here?* — and WordPress, by default, answers it for strangers. The well-known route is the REST endpoint: an unauthenticated request to `/wp-json/wp/v2/users/` returns the login names and author slugs of every account on the site. On one site in the fleet, a single unauthenticated request returned three real usernames — the exact strings a login-guessing run needs. Most security plugins don't close this by default, and most admins don't know it exists.

The admins who *did* know had closed it. Several sites returned a correct `401 Unauthorized` on `/wp-json/wp/v2/users/`. Enumeration handled, the dashboard would say.

Except WordPress exposes the same usernames through a second route, and almost nobody closes that one. The legacy `/?author=1`, `/?author=2`, `/?author=3` URLs each issue a `301` or `302` redirect — and the redirect *target* is `/author/<the-real-login-name>/`. The username is right there in the `Location` header. A scanner walking `/?author=N` harvests exactly what it would have gotten from the REST endpoint, against sites whose admins believe they've shut enumeration down.

We watched this exact split — REST locked, author-redirect open — recur on about half the sites we monitored, in separately configured installations that had nothing in common but the platform they run. On one, the redirect leaked the site's real login name directly. On another, a partial fix had landed: one author ID still redirected and leaked while the rest returned a clean `404` — a defense applied to every door but one. The tell, every time, is the HTTP status: a `301`/`302` on `/?author=N` (rather than a `404`) confirms both that the user ID exists *and* what the username is.

This is partial mitigation that reads as complete. The plugin's enumeration toggle goes green. The REST endpoint genuinely is closed. And the exposure is unchanged, because the second route was never part of the toggle.

> ***Tonight, on your own site:*** *`curl -I 'https://yoursite/?author=1'`. A `404` is what you want. A `301` or `302` means your usernames are public, no matter what your REST endpoint does. Close both: lock the REST route and add a rule that turns `/?author=N` into a `404`.*

The WAF said *blocked*. The clock said *executed*.

That was the horizontal blind spot — the campaign you can't see across. The rest of this report is the vertical one: what your own defenses actually did, underneath the dashboard that says they worked.

A web application firewall in front of your login page is good hygiene, and on the sites that had one, it mostly did its job. But "the WAF returned an error code to the attacker" and "nothing happened" are two different statements, and the gap between them was where two of the more unsettling findings in this report lived.

The first is about what the WAF hides from *you*. One site was hit by a textbook brute-force: a few hundred requests to the login page from a single host inside a couple of minutes. The site's WAF handled it perfectly — rate-limited the requests, then locked the offending address out for thirty days after the second bad username. No compromise, site never wobbled. And the monitoring saw *almost nothing*: a

ten-minute quiet gap, then a single "slow response" blip as the site flushed its backlog. Zero brute-force alerts.

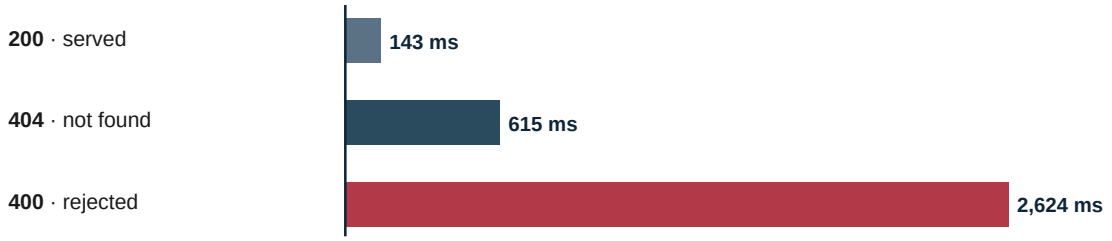
Here's why, and it's structural: a WAF that handles the login page at the HTTP layer returns its block *before* WordPress core ever runs the authentication code. The failed-login hook never fires. From the application's point of view, the attack didn't happen — there's nothing to alert on. Which sounds fine until you say it the other way around: **the better the site is defended at the edge, the less anyone downstream can see.** Every well-WAF'd site is a future "why didn't you warn me?" conversation, because the attacks that get blocked are exactly the attacks that become invisible.

The second finding is about what the WAF hides about the *attack*. One site took a steady run of SQL-injection attempts against its admin AJAX endpoint, spread across a hundred different plugin actions. Some we could read straight from the request URL — classic time-blind probes like `?meta_ids=1 AND (SELECT 3066 FROM (SELECT(SLEEP(6))))` and `UNION SELECT` attempts. Others arrived as POST bodies, which we don't record, visible only as the same endpoint answering after a long pause. Every one of them came back `HTTP 400`. Rejected. Blocked. Customer safe, if you read the firewall log in isolation.

But those rejected requests didn't come back quickly. The `400`s on that endpoint averaged around two and a half seconds, and the slowest ran past eleven. A genuine rejection is a few milliseconds — so a time-blind `SLEEP(6)` payload that takes six-plus seconds to return its `400` tells you the database executed the delay *before* the layer above it issued the rejection. The request was rejected at the HTTP boundary; the injected SQL had already reached the database and run. How much further it got depends on how each plugin builds its queries — the timing proves the payload reached the database layer, not how much of it executed. That uncertainty is the finding: "blocked" at the HTTP layer is not the same as "didn't run."

`HTTP 400` is a statement about the response. It is not a statement about whether any code ran first — and the only reason anyone could draw that distinction here is that the timing was recorded alongside the status at all. Without it, this is a clean line in a firewall log: attack blocked, nothing to see.

> **Rejection isn't free.** Average response time by status class on one site over the period. A request returned as `400` — "rejected" — ran on average roughly eighteen times longer than a real `200` page, because the database executed the injected work before the rejection came back. (The `404`s are their own story: a 404-monitoring plugin doing a database lookup on every miss.)



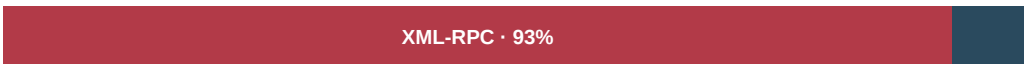
> ***A query you can run on your own logs:*** *average response time grouped by status code. If your `400`s and `404`s are anywhere near as slow as your `200`s, something heavy is running before the rejection — and on an injection target, "something heavy" may be the payload itself.*

Two hundred thousand login attempts. Almost none at the login.

One admin had done exactly what every hardening guide says to do: moved the WordPress login off `/wp-login.php` to a secret URL. It worked, in the narrow sense — the secret address caught a *handful* of attempts in two months. Nobody was finding it.

And the brute-force against the site didn't budge. Over the same two months it absorbed **more than 200,000 failed login attempts** — and **93% of them arrived through `xmlrpc.php`**, the legacy remote API that hiding the login page doesn't touch and most admins forget is even switched on. A single XML-RPC request can carry a whole batch of password guesses, so a few hundred addresses produced two hundred thousand attempts without ever once looking for the login form.

> **Where 200,000 login attempts actually landed.** One site, two months. The admin had moved the real login to a secret URL — it caught six attempts. Ninety-three percent of the brute-force came through `xmlrpc`, which no login-hiding trick covers. The default `/wp-login.php` the admin thought they'd retired absorbed the rest.



The default `/wp-login.php` the admin thought they'd retired · 7%

The secret login URL the admin actually moved to: 6 attempts in two months.

This is the trap in one move. From the admin dashboard it reads as "200,000 failed logins" — a login page under siege — so the natural response is to harden the login, which this admin already had. The login was barely involved. The attack was coming through a door on the other side of the house, and nothing on the dashboard said so.

And it wasn't one site. The same shape — login hidden, `xmlrpc` wide open — repeated across multiple monitored installations: the door everyone is told to lock, locked; the same window left open behind it, every time.

> ***On your own site:*** *the login URL is the famous thing to harden; `xmlrpc.php` is the one to actually check. If you don't use the WordPress mobile app or Jetpack, disable it; if you do, block the `system.multicall` and `wp.getUsersBlogs` methods. And watch where your failed logins land, not just how many there are.*

The guard that can't call for help

The most uncomfortable pattern in the fleet was the security tool as the liability — not because the tools were bad, but because nobody had asked the second question: *if this fails, how would I know?*

On one site, the answer was: you wouldn't. The site's security plugin ran scheduled scans, and its only out-of-the-box way to tell the admin anything — findings, warnings, "I'm broken" — was email. The host had disabled PHP's `mail()` function, which is a common and otherwise reasonable shared-hosting hardening step. So every scheduled scan crashed at the moment it tried to send its report, threw a single

fatal error, and stopped. No findings. No failure notice. No admin warning. For weeks. The thing that failed was the thing that was supposed to tell you it failed, and it had no second channel to say so. We only caught it because we watch for PHP fatals directly, with no dependency on any plugin's own reporting path.

That generalizes well past WordPress. The backup tool that emails its failure reports. The uptime monitor that pages you through a service it doesn't itself ping. The error tracker that needs the same database connection it's trying to report an error about. Any guard whose alarm runs through the thing it's guarding can fail silently for as long as it takes a human to wander past and notice.

The other shape of the same problem is the security tool as the load. On one site, the security plugin's scanner ran more than a dozen times in a single day, tens of seconds each, totaling **several hundred thousand database queries** — more work, that day, than the actual website did. It starved the site's scheduled-task capacity, blocked other cron jobs from running on time, and inflated the host's database load, all in the name of protection that was only marginally better run that often than it would have been run once. "Install it and forget it" is the advice everyone gives for security plugins. It's also how a useful tool quietly becomes the heaviest thing on the server.

And — the version of this we'll describe by shape rather than detail, because the plumbing was our own — one site was *paying* for a tunable-rules add-on while a coordinated brute-force ran against it for hours. Subscription active. Fifteen hundred failed logins. Zero alerts. The protection had been purchased but, through a silent misconfiguration, never actually attached to the site. The billing record and the running defense disagreed, and the only way to catch that disagreement was to observe the site from outside both. "The customer paid for it" is not the same sentence as "the alert will fire," and the distance between those two sentences is invisible from inside the product.

The fix that hides the break

The last pattern is the gentlest-looking and, in some ways, the most insidious: a fix that works just well enough to keep you from noticing what it's fixing.

One site's scheduled-task system — WordPress's cron — was, functionally, dead. Hooks were stuck in the queue for years; at the extreme, a task was **more than five years overdue**. The admin had no idea, because they'd installed a band-aid plugin whose entire job is to retroactively run scheduled posts that missed their slot. And it worked: posts got published, eventually. So the underlying breakage stayed invisible, papered over by a plugin firing a hundred-plus times a day, burning a couple of minutes of server time daily to compensate for a system that should have just run on schedule. The band-aid's *firing rate* was the real diagnostic — a plugin working overtime is sometimes the clearest evidence that something underneath it is broken — but you only see that if you're counting how often the cure gets applied.

The starkest version was a corrupt database row on one subsite of a multisite network. For three weeks, every public URL on that subsite returned a healthy `200`. Every uptime monitor was green. Every performance check passed. And the whole time, a corrupted user-roles record was throwing thousands of errors a week during background tasks — invisible, because the corruption only surfaced during the role-init step that cron and the admin panel run, never during a normal page load. It stayed latent until someone finally opened the admin screen for that specific subsite and hit the same fatal that cron had

been hitting silently since the start. Three weeks of accumulating failure that no uptime, status, or response-time monitor would ever have caught, because none of them were watching the one thing that was actually failing: work that's supposed to happen quietly in the background.

The lesson both sites teach is the same. "It still loads" and "it still publishes" are not health. They're the absence of a *symptom you happen to be looking at*. The failure was real, present, and compounding the entire time — it just hadn't picked a channel you were monitoring yet.

What this means if you run a WordPress site

None of the sites in this report were negligent. That's the part worth sitting with. They had the plugins, the firewall, the hardening. What they didn't have was the right vantage point — and almost no single-site admin does.

Two blind spots, one cause. **Horizontally**, you can only see your own site, so the campaign you're one target in looks like background noise. **Vertically**, you can only see your dashboard, so a defense that reports "handled" gets believed even when it let the payload run, crashed in March, or was never attached. Neither gap is about missing tools. Both are about where you're standing.

You can't manufacture the horizontal view on your own — that's the honest limit of a single site. But the vertical checks you *can* run, and they're worth an afternoon:

- **Usernames, both doors.** `curl -I 'https://yoursite/wp-json/wp/v2/users/'` should be `401`. `curl -I 'https://yoursite/?author=1'` should be `404`, not a redirect. Check *both*.
- **Slow rejections.** Group your access-log response times by status code. A `400` or `404` that's as slow as a real page is a sign something heavy — possibly hostile — is running before the rejection.
- **The guard's second channel.** Whatever tool you rely on for security alerts: how does it reach you, and what happens if that channel breaks? If the answer is "email, and I'm not sure," that's the gap.
- **Background work.** Your scheduled tasks fail silently by design. Find out whether yours are running on time, and treat a "compensating" plugin (one that retroactively fixes missed jobs) as evidence of a problem, not a solution to one.
- **Paid protection.** If you pay for a security feature, confirm it's *attached and firing*, not just billed. Trigger a benign version of what it's supposed to catch and check that it caught it.

We watch sites for a living, so we'll say the obvious thing plainly: the reason these gaps stayed open for weeks or months is that no one was standing where they could see them. Not on the status page, not in the scan report, not in the billing record — in the actual behavior of the actual defense under actual traffic, and in the pattern that only appears when you can see more than one site at once.

The last word

A defense you haven't verified is a hypothesis, not a control. And an attacker you've only seen once is a campaign you're not seeing the rest of.

So run the checks more than once. Every failure in this report has a clock attached: a control that passed one month crashed the next, a paid defense detached itself the day someone touched a config, a campaign didn't stop just because you stopped watching it. None of them announced the moment they went quiet — so a single audit, however careful, only describes the afternoon you ran it. Put the quick vertical checks on a standing schedule — monthly is plenty for most sites — and run them again whenever the ground shifts: a plugin update, a host migration, a new security tool, a change to who can log in. The full pass is worth a quarter's attention. The gap all of these problems lived in is the distance between "was true once" and "is true now" — and the only way to keep it closed is to keep looking.

Written from the field by the team at Logystera — June 2026.

We do the two things this report is about: watch each site from the inside, and watch the whole fleet from the outside. The blind spots close where those two views meet.