mySites.guru

Your .htaccess Won't Stop a Joomla Hack

A hardened .htaccess feels safe, but Joomla attacks ride straight through index.php. Here is why the file protects far less than most site owners think.

Phil E. Taylor | 28 August 2026

A provocative, click-bait claim? Yes. Also true: your `.htaccess` file does almost nothing for your Joomla site's security.

That will annoy a lot of people, because hardening the `.htaccess` is the first thing most site owners and half the "secure your Joomla" guides reach for. It feels like locking a door. In practice it is closer to padlocking your fridge and fitting a deadbolt to your toilet door, then calling it home security. The burglar cannot drink your beer and he cannot use your toilet. He is still standing in your kitchen, and he was never especially interested in either. The lock is real. The reassurance is real. The security is mostly theatre.

A lot of people will not read this entire post, for sure, and will be angered at the very notion that I have written it, and will attempt to discredit parts of it. That does not change the hard truths it tells.

Let me be precise about what I am and am not saying. I am not saying a `.htaccess` is useless, or that you should throw it away. It is good at a short list of jobs. I am saying that the thing most people believe it does, stand between an attacker and a vulnerability, is not the thing it actually does, and that believing otherwise is how sites get owned while their owners feel protected. I will show you why using Joomla's own documentation and Joomla's own code as the evidence, and then tell you what actually keeps a site standing.

Joomla is about to bet harder on the .htaccess

The reason to say all this now is that Joomla is moving in exactly the opposite direction. An ongoing surge in third-party Joomla extension exploits has the community rattled and looking for an answer, and one answer is gathering real support: ship a stricter `.htaccess` by default so that a fresh Joomla install is locked down out of the box.

There are two threads to this. The first is already on the table as Brian Teeman's open pull request #48076, targeting a future Joomla 6.2. It adds a directory-scoped allow-

list to `htaccess.txt` for the `images`, `media` and `files` folders: a defined set of safe extensions is permitted, and everything else in those three directories gets a 403.

```
RewriteRule ^(?:images|media|files)/ - [F,L]
```

The allow-list in front of that deny rule runs to `7z|avif|bmp|css|csv|doc|docx|eot|gif|gz|htm|html|ico|jpg|js|json|mp3|mp4|pdf|png|svg|txt|webp|woff|woff2|xls|xlsx|xml|zip` and a handful more. It is a sensible, tightly-scoped idea aimed squarely at the folders attackers love to drop shells into, and on its own, as one narrow layer, I have no quarrel with it. The danger is not that narrow rule. It is treating a few whitelisted folders as if they were a security perimeter. The second thread is the one to watch: a Security Strike Team discussion has moved further, toward a default `.htaccess` that restricts which entry points are allowed to run at all, front-controller style, by default. David Jardin, a member of the Joomla Security Strike Team, offered to prepare that pull request.

The instinct is right. Something does need to change, and the people involved are competent and not inventing the problem. But restricting entry points in the `.htaccess` defends at the wrong layer, and the community's own discussion surfaced exactly why, in several ways worth pulling apart one at a time.

It breaks legitimate tools

When the entry-point restriction was debated, the counter-examples came from the most respected corners of the ecosystem: Akeeba's `kickstart.php` and its backup-restore script are designed to run as standalone PHP outside the Joomla application, and Joomla's own updater has `index.php` hand off directly to the updater's `extract.php` under the right conditions. (And yes, recent Joomla versions can now route the core auto-update through `index.php` itself; that does not change the fact that for well over a decade the upgrade required direct access to `com_joomlaupdate`'s own extract and finalise scripts, and plenty of installs still lean on exactly that path.) A blanket "only approved entry points may run" rule risks breaking the exact tools people reach for when a site is already on fire.

And it reaches well beyond recovery tools. A great deal of ordinary e-commerce depends on an external service posting directly to a script on your site: PayPal IPN callbacks, WorldPay and other payment-gateway notifications, webhook receivers, shipping and CRM integrations, every one of them a server-to-server request that arrives with no browser, no session, and often no trip through `index.php` at all. Block the endpoint the gateway was told to call and the notification is simply lost, so the customer is charged, the order never completes, and nobody notices until the refunds and chargebacks start.

You can rewrite all of it, of course. Every one of those endpoints could be re-plumbed to enter through Joomla's approved `index.php` and satisfy the whitelist. But there is a decade of working scripts already in production that a single line added to a default `.htaccess` would break the instant it shipped, with no error, no migration path and no warning. And the rewrite is not free even when you do it. Routing a lean, standalone callback through `index.php` means booting the entire Joomla stack to serve it: loading the framework, firing every system plugin, running all the bootstrap code a full page view runs, just to record one payment notification that a small standalone script currently handles in a fraction of the CPU and memory. You would be trading a fast, isolated request for a slow, heavy one on every callback, and calling the result more secure.

It only works on Apache

As Teeman himself noted of the Windows equivalent on that very pull request, "As the web.config already has weaker settings than htaccess I would suggest that is considered in its own pr" - an open admission that the non-Apache half of Joomla's own shipped hardening is weaker and, in his words, that he cannot test it. Every Joomla site on IIS, and every site fronted by Nginx or Cloudflare, gets a different and lesser version of this protection, or none of it. A default lock that works one way if your house was built by Apache, a weaker way if it was built by IIS, and not at all if it was built by Nginx or Caddy, is not a default. It is a coin toss decided by a hosting choice your site owner made years ago for unrelated reasons.

And there is no parity across the web servers Joomla claims to support. The project hardens the Apache `.htaccess` while, by its own admission, the `web.config` it ships for IIS has weaker settings, and Nginx, Caddy and every other web server get nothing at all. Nor is there a word of guidance on the layer that now fronts a huge share of Joomla sites: how to correctly configure Cloudflare, which by itself sits in front of more than a fifth of the entire web, is left completely unaddressed, even though the edge is exactly where this kind of hardening actually belongs. A single release can therefore lock down the wrong things on Apache and, by pouring all its attention there, leave every non-Apache site with a wider relative attack surface and no equivalent protection. Hardening one server family while neglecting the rest is not defence in depth. It is defence in one place.

It fixes nothing behind the doors that stay open

A default rule that blocks some entry points does absolutely nothing about the vulnerable code sitting behind the entry points that have to remain open. Joomla's own `com_ajax` is the clearest example: a general-purpose dispatcher, reached through `index.php` by design, that any plugin or module can hook into, and one no entry-point whitelist can ever close because a large part of the front end depends on it. Several of the vulnerabilities we have disclosed rode straight through it: an unauthenticated file write in JoomShaper's Helix3 and an unauthenticated menu write that became stored XSS in Helix Ultimate were both reachable through `com_ajax` with no token and no permission check at all. The whitelist has to leave that door open, so it does nothing for any of them. And nearly every exploit in the current surge arrives through the one entry point that can never be closed: `index.php`.

It gives novice users no safe way to change it

Even where the shipped rule is right, the people it reaches cannot safely adapt it, and that matters the moment their site needs it to behave even slightly differently. Editing a `.htaccess` is not for the faint-hearted: it is regular-expression wizardry that even seasoned web developers get wrong, and a single mistaken character does not degrade gracefully, it returns a `500 Internal Server Error` for the entire site. Akeeba's Admin Tools Professional solved exactly this years ago with its `.htaccess`

Maker, a checkbox interface that lets a non-expert choose behaviours instead of hand-writing PCRE. There is no proposal to give the core Joomla administrator anything like it. So a site owner who needs to allow one legitimate extension, or unblock one path the default blocked, is given a file of raw regex and left to edit it by hand, which is how these files end up either breaking the site or being deleted wholesale in frustration.

A hardened .htaccess protects less than you think

Here is the uncomfortable mechanic that the whole argument rests on. Almost every serious Joomla vulnerability of the last few years, SQL injection, unauthenticated file upload leading to remote code execution, arbitrary file read, arbitrary file deletion, is exploited through a normal, approved request to `index.php`. Put bluntly: **many of the recent Joomla vulnerabilities walk straight in through that whitelisted front door**, the one door every hardening scheme is forced to leave open. It is the main Joomla entry point, the front controller that every legitimate page view already flows through. Your `.htaccess` exists to route requests *to* `index.php`. Its central job, the SEF URL rewriting that makes `/about-us` resolve to `index.php?option=com_content&view=article&id=42`, is to make sure requests reach that front controller. It cannot then protect you from an attack that arrives as a perfectly ordinary request to the one endpoint it is built to feed.

Think about what a `.htaccess` can actually see and act on. It works at the level of the request line and headers: the path, the query string, the method, the user-agent, the referrer. It has no idea what the PHP behind that request is going to *do* with those values. A gallery component, a forms extension, a page builder, any of them can contain a flaw that an attacker triggers with a request that is, at the HTTP level, indistinguishable from legitimate traffic. The rewrite rules wave it through, because at the layer where `.htaccess` operates there is nothing wrong with it. The vulnerable PHP runs. No `.htaccess` rule was ever going to sit in the way, because the request never looked wrong at the only layer `.htaccess` can inspect.

Technical detail: what a .htaccess can and cannot see

Joomla's SEF rewriting is the block that ends with `RewriteRule (.*) index.php [L]`. Every pretty URL, say `/blog/my-article`, is rewritten to `index.php?option=com_content&view=article&id=42` and passed to Joomla's front controller. The `.htaccess` operates entirely on the request line and headers: the method, the path, the query string, the user-agent, the referrer. It never sees, and by design cannot see, which component the router will dispatch to, which `task` will run, whether the code casts `id` to an integer before it hits the database, or whether the extension calls `authorise()` before it writes a file.

By the time any of those decisions are made, the request is already several stack frames deep inside PHP, well past the layer the `.htaccess` controls. Trying to filter the attack at the `.htaccess` means guessing, from the request line alone, what the PHP a dozen frames later is going to do wrong. You cannot pattern-match your way to that from the outside.

This is not theoretical, and it is not rare. It is the same pattern behind attacks that ride in through the CMS's own AJAX and API endpoints, where a component verifies a CSRF token but never checks who is calling or whether they are allowed to. It is why a single POST to an approved SP Page Builder endpoint could drop a live web shell, returning a clean 200, immediately followed by a request to the planted PHP file. The perimeter was never breached. Nothing was forced. The attacker used the door, held open by design, that your **`.htaccess`** exists to keep open.

How do Joomla attacks get past your .htaccess?

They do not get *past* it. They go *through* it, using the exact route it is built to permit. It is worth walking through the two most common Joomla compromise patterns in detail, because once you see the mechanics you stop believing a rewrite rule could have helped.

Unauthenticated file upload to remote code execution

A front-end task in a vulnerable extension accepts a file it should have rejected, or writes it to a folder it should never have written to. The request is something like **`index.php?option=com_vulnerable&task=upload`**, a POST with a multipart body. To your **`.htaccess`** this is a request to **`index.php`** with a query string and a file attached, which is exactly what a legitimate contact-form submission or media upload looks like. It passes it on without hesitation. The extension then writes an attacker-controlled

`.php` file into a web-served directory, and the attacker requests it. Now they have code execution on your server. We have documented this precise class over and over: the JCE profile-upload attack that walked in through a legitimate editor endpoint, and fully unauthenticated remote code execution straight through the com_ajax endpoint in the Novarain Framework, where six weeks after the patch nearly half the affected sites in our data were still exposed. In none of these did a `.htaccess` rule have any part to play, because the malicious request was, byte for byte at the HTTP layer, an ordinary one.

SQL injection

A parameter that should be an integer, an article ID, a category filter, a sort order, is concatenated raw into a database query instead of being bound or cast. The attacker sends their payload as a value on a normal page URL: `index.php?option=com_something&id=1 UNION SELECT ...`. There is no `.htaccess` rule on earth that can reliably tell a malicious `id` value from a legitimate one, because both are just text in the query string of a request to `index.php`. You could try to blocklist SQL keywords, and people do, and it is the same losing game we will come to below: attackers encode, comment-split, and case-vary their way around any keyword list you invent, while your rule starts throwing 403s at legitimate searches that happen to contain the word “select” or “union” or an apostrophe.

Unauthenticated data exposure through a broken access check

Not every compromise ends in code execution. Some just hand over data that should have stayed private, and a `.htaccess` is even more useless against these because there is not even a stray file to block. We recently found an invoice download in Events Booking that returned any registrant’s invoice to an anonymous visitor: full name, postal address, email and the amount they paid, chosen by a sequential ID number in the URL. The request is an ordinary GET to `index.php`, indistinguishable at the HTTP layer from an actual attendee fetching their own receipt. Your `.htaccess` looks at it, sees a normal request to `index.php`, the one file it is built to whitelist so Joomla can run at all, and waves it straight through. There is no upload to inspect, no `.php` file to deny, no SQL keyword to match, nothing for a rule to grab hold of. An attacker just

counts the ID up from one and collects every invoice on the site. A hardened `.htaccess` would not have stopped a single one of those downloads, because the whole attack rode in through the exact front door the file exists to hold open.

When we went back over a full month of Joomla security disclosures, nearly every one shared a single root cause: a public endpoint that trusts anonymous input. A firewall or a hardened `.htaccess` can, at the very best, stop an already-uploaded PHP file from *executing* in an uploads folder, which is worth having as one layer. Neither one fixes the underlying bug, and neither one stops the request that triggers it. The vulnerability is in the code the request reaches, not in the route it took to get there.

What if an attacker can delete or overwrite your Joomla .htaccess?

Now it gets worse, because a `.htaccess` is not a wall. It is a file. Files can be deleted. Files can be overwritten. And plenty of vulnerabilities let an attacker do exactly one of those things.

An arbitrary-file-deletion bug removes your `.htaccess` in the time it takes to send one request, and takes every protective rule in it with it - like the one I recently reported, responsibly, in Joomla core's own updater, and which the project fixed. Go figure: the same project now proposing to lean harder on the `.htaccess` had a flaw that could have deleted that very file. (More on that shortly.) From that moment the folder it was guarding is wide open, and the deletion may not even be noticed for weeks. An upload-anywhere bug is arguably worse, because the attacker does not need to delete anything: they write a new `.htaccess` of their own, straight over the top of yours. That replacement can do anything Apache allows. It can re-enable PHP execution in your uploads directory with a single `AddHandler` or `php_flag engine on` line, turning a folder of harmless images into a place their shell will run. It can add a `RewriteRule` that cloaks your site, showing Googlebot spam and showing you a clean page. It can 301-redirect every visitor to a phishing or malware network while your homepage still looks fine to you. The file you trusted to protect the folder becomes the mechanism that betrays it.

Technical detail: how a planted .htaccess turns images into code

An attacker who can write a `.htaccess` into a folder does not need to touch your server's PHP handler configuration at all. One line does it: `AddType application/x-httpd-php .jpg`, or `AddHandler application/x-httpd-php .gif`, or `SetHandler application/x-httpd-php`, or on some stacks `php_flag engine on`. Any one of those tells Apache to run every matching file in that directory through the PHP interpreter, so the `holiday.jpg` they uploaded through your vulnerable contact form is now live, executable code.

This is exactly why the one useful hardening rule for an uploads folder is a `.htaccess` that *denies* PHP execution there, and exactly why an upload-anywhere bug that lets the attacker overwrite that rule with their own is the whole game. The defence and the weapon are the same file, and the attacker only needs to win the race for it once.

This is not hypothetical. Take the SP Page Builder zero-day, an unauthenticated file upload that lets a guest write files into a Joomla site. On a site that has done the "recommended" hardening, direct execution of **`.php`** is denied, so a plain **`shell.php`** upload would only ever return a 403. The attacker knows this, so they do not upload **`.php`** at all. They write their payload as a **`.json`** file, which slips past both the filename filter and the **`.php`** deny rule, and they drop a **`.htaccess`** beside it that teaches Apache to run every **`.json`** file in that folder through the PHP interpreter:

```
AddType application/x-httpd-php .json
AddHandler application/x-httpd-php .json
<FilesMatch "\.json$">
    SetHandler application/x-httpd-php
</FilesMatch>
```

Now **`payload.json`** is not data any more, it is executable code, and it runs on the very next request. The carefully written rule that denied **`.php`** was never in the fight, because the attacker simply chose an extension it never named. **`.json`** looks like configuration, like a language file, like something no security guide ever thought to block, and that is precisely why it was picked. You guarded one door; they walked through the one beside it and relabelled it on the way in.

There is a subtler and more damning version of this, and it needs no upload-anywhere bug and no deletion primitive at all. Your protective rule and the attacker's **`.htaccess`**

use the exact same mechanism, and Apache always hands the win to whichever one sits *deeper* in the tree. Apache assembles the **`.htaccess`** rules for a request by walking from your document root down to the folder holding the file, applying each in turn, so the most specific, deepest file has the final word. Your "no PHP in **`/images/`**" rule lives in the root, one or more levels up. The attacker is writing *into* **`/images/`**, the very folder your vulnerable form let them reach. They drop a one-line **`.htaccess`** beside their uploaded file that re-enables execution, and because their file is deeper than yours, it overrides yours for that folder. Your root rule is never touched. It is simply out-nested. Put a lock on the fridge and I will drill through the back of it, take a beer, and fit my own lock to the hole I made. Yours is still there. Still shiny, still locked, still on the front of a fridge I am no longer using the front of.

The symmetry is the whole trap. The **`AllowOverride`** setting that lets *your* **`.htaccess`** deny PHP in that folder is the identical setting that lets *their* **`.htaccess`** allow it straight back. It is one switch, set in the server config for the entire tree, and you cannot have it on for your deny rule and off for their allow rule. The only version of this block a dropped **`.htaccess`** cannot reverse is one written at the server layer: **`php_admin_flag engine off`** in a **`<Directory>`** block, which **`.htaccess`** is expressly forbidden from countermanding, or **`AllowOverride None`** on that path so the attacker's file is never read at all. Which is to say, once again, that the only real protection lives at the layer the root-**`.htaccess`** hardening guide, and the shared-hosting user pasting it in, cannot reach.

Technical detail: the deepest .htaccess wins

When several `.htaccess` files apply to one request, Apache does not pick one, it merges them from the document root down to the requested file's directory and applies the deepest one last. A root rule such as `<FilesMatch ".php$"> Require all denied </FilesMatch>` guarding `/images/` is therefore overridden by a `<FilesMatch ".php$"> Require all granted </FilesMatch>` an attacker drops inside `/images/`, because Apache's default `AuthMerging Off` uses only the closest context's authorization directives, and the attacker's context is closer. The same holds for `php_flag engine on` under mod_php, or an `AddHandler application/x-httpd-php .php` / `SetHandler` line that re-routes execution: the deeper file has the final say.

This block can only be made irreversible where a `.htaccess` cannot reach or override it: `php_admin_flag engine off` (forbidden in `.htaccess`, honoured only from `httpd.conf` or a vhost), or the deny placed in a server-config `<Directory>` block with `AllowOverride None` on that path, so no `.htaccess` in the folder is read at all. Both are server-layer controls. An

attacker standing in your uploads folder is always deeper in the tree than your root rule, and when Apache decides a contest by depth, the one writing into the leaf folder wins by default.

I know this class of bug intimately, because I have reported it in Joomla core itself.

CVE-2026-23898 - reported by Phil Taylor

CVE-2026-23898 is a high-severity arbitrary file deletion vulnerability (CWE-73, CVSS 4.0 score 8.6) in `com_joomlaupdate`, the component responsible for keeping Joomla up to date. A lack of input validation in the autoupdate server mechanism allowed deletion of files the PHP process can reach, not just Joomla files, anything the web server user has access to. It affected Joomla 4.0.0 through 5.4.3 and 6.0.0 through 6.0.3, and was fixed in 5.4.4 and 6.0.4.

The point for this article is blunt: the very file you rely on for hardening is one of the first an attacker with a deletion primitive would remove, and the component that could have deleted it was Joomla's own updater. If your entire security posture is a file, your entire security posture is one file-deletion bug away from gone.

So the defence you spent a careful afternoon perfecting can be undone in a single request by a bug in an extension you had forgotten was installed, or, as CVE-2026-23898 shows, by a bug in Joomla core. The lock is only ever as strong as the weakest extension's ability to reach the door frame, and in a typical Joomla site with twenty or thirty third-party extensions, a great many of them can reach the door frame. Betting your security on the integrity of a file that dozens of code paths can write to or delete is not a strong position.

Would a stricter default .htaccess have stopped last month's hacks?

No. Not one of them. I went back through every Joomla SQL injection and every unauthenticated upload-to-remote-code-execution flaw we confirmed in the last month, and a stricter default `.htaccess`, the kind now being proposed, one that restricts entry points and whitelists a handful of locations, would not have prevented a single one. That is not a rhetorical flourish. It is the direct consequence of the mechanic above: those exploits all arrive as ordinary requests to `index.php`, and `index.php` is

the one entry point any workable whitelist has to keep open. You would have been compromised on exactly the same day, in exactly the same way, with or without the extra rules.

What a stricter `.htaccess` changes is not *whether* the intruder gets in, but what he finds once he is inside, and even that he routes around in seconds. Picture the break-in properly. He is already through the front door, because the SQL injection or the upload flaw let him walk in through `index.php`. Now he wants to leave himself a web shell somewhere it will run. The media cupboard is bolted, the `tmp` room is locked, a couple of the internal doors are shut, because your shiny default `.htaccess` whitelisted three folders and denied the rest. So he does the digital equivalent of finding the downstairs toilet locked and simply relieving himself in the garage instead. He writes his shell into one of the many writable locations your three-folder whitelist never covered. Or he skips dropping a file altogether: he uses the same hole to add himself a Super User account, or to inject a malicious template override that Joomla itself will execute, and runs his code straight through the front controller. The locked internal doors did not stop him. They only decided which room he wrecked.

And some of last month's flaws do not even fit that picture, because nothing is ever dropped, executed or hidden. Take the Events Booking invoice leak we disclosed. A stricter default `.htaccess`, even the entry-point-restricting kind now being proposed, would have waved it straight through, because the whole attack is a single ordinary GET to `index.php`, the one endpoint the rule is built to allow. There is no shell, no `.php` file, no payload of any kind for a rule to catch. The attacker just increments the ID in the web address and the server hands back another registrant's invoice: full name, postal address, email, and the amount they paid. So the strictest whitelist you can write still leaves the personal and financial data of every attendee readable by anyone on the internet, because reading it never looked like an attack in the first place. It looked exactly like a visitor collecting their own receipt.

Technical detail: what a whitelist of locations actually blocks

A restricted-entry-point `.htaccess` typically does two things: rewrite unknown paths to `index.php`, and deny direct execution of `.php` files outside a small allow-list, using a `<FilesMatch>` block or a `RewriteRule` that 403s any `.php` request that is not `index.php`. That stops exactly one move: browsing directly to a shell at `/images/evil.php`. It does

nothing about the SQL injection, the upload flaw, the add-an-admin, the template-override injection, or the PHP object injection, because every one of those executes *through* `index.php`, which is allow-listed by necessity.

Nor does it stop a shell written into any directory the rule did not enumerate, and Joomla writes to far more than three folders: `tmp/`, `cache/`, `administrator/cache/`, `logs/`, per-extension upload paths, avatar directories, and more. Whitelisting `images`, `media` and `files` is not a perimeter. It is three locked doors in a house full of open windows, and the attacker only has to notice one window.

And "whitelist a few example locations" is more fragile than it sounds because Joomla and its extensions scatter writable, sometimes web-served, directories all over the tree. Why lock the fridge and leave the freezer standing open beside it? A default that only reasons about three of them hands users a powerful feeling of coverage that the filesystem does not back up. Worse, every legitimate extension that writes or serves from a non-whitelisted path now breaks with a bare 403 and no clue as to why, which is the definition of a silent breaking change. And a user who cannot read the `.htaccess` well enough to add a precise exception has exactly one obvious fix when their gallery or forms extension stops working: delete the whole file. One broken feature, and every rule you shipped to protect them goes in the bin together.

There is a further problem the whitelist creates rather than solves, and it follows directly from what it chooses to allow. The whole point of that allow-list is to permit images, PDFs and text files while blocking executables. But an attacker who can write those does not need code execution to get value out of your server. A file he can upload anonymously and have you serve from your own domain is free, hard-to-trace storage that points at you rather than at him: a malware payload, the images for someone else's phishing page, a text file staged for the next step of an attack, all hosted by you, delivered under your reputation, and traced back to your IP. We have watched exactly this in the unauthenticated upload flaws we reported in Events Booking and Membership Pro, where dropping a file was the foothold, not the finish. Whitelisting the "safe" extensions does not close that door, it blesses it: the folder is still a drop box, the file is still served, and your webspace is still doing an attacker's storage for him.

The exception: when something did block one, it was not the .htaccess

In fairness, one of the vulnerabilities we disclosed was stopped on some sites before it ever reached the vulnerable code, so it is worth being precise about what stopped it. It was Admin Tools' web application firewall catching the SQL injection payload at runtime, not the `.htaccess` that the same product's `.htaccess` Maker generates. Those are two separate features that happen to ship inside one extension, and people conflate them constantly. The WAF is PHP, running inside the request, able to inspect parameter values after Joomla has parsed them, which is exactly the layer a `.htaccess` cannot reach. Give the credit to the thing that earned it.

And even that is thinner than it looks, because it is a toggle. The SQLi filter sits on a settings screen, alongside other filters that occasionally throw a false positive on a legitimate request, and it is one click away from being switched off by an administrator who does not know what SQL injection is and only knows that turning that thing off made the complaint stop. A protection that depends on a checkbox nobody understands staying ticked is not a protection you can count on. It also never fixed the vulnerable query sitting behind it, which is still there, still waiting, on every site where the box gets unticked.

Now for the part that should worry you, because it is how most of these break-ins actually finish. Your beautifully hardened live site does not live alone. It shares a hosting account, a filesystem, and very often a single PHP process, with every other site in that account. And one of those other sites is the dev copy you spun up on a subdomain two years ago, never updated, still running **`test`** / **`test`** as the Super User password because it was "only a test site". The attacker does not pick a single one of your expensive new locks. He walks into the neighbour you forgot about, through the connecting door, and comes out inside your live site's files. Your garage had an internal door into the kitchen, and you left the garage unlocked.

The weakest site in your webspace is the way in

Running many domains and many different applications, a Joomla site here, a couple of WordPress sites there, an old Drupal thing nobody remembers installing, inside a single webspace is a serious mistake. Do not do it. The reason is simple. If an attacker compromises just one file, they can infect and reinfect every file and every site in that webspace, and in the worst case they can delete all of them. Hack one, hack all. This argument alone should end the "let's just ship a stricter default **`.htaccess`**" conversation, because it is a dominant real-world compromise path and a **`.htaccess`** can do nothing about it.

A quick definition, because the webspace is the unit that actually matters here and almost nobody thinks in it. On a cPanel server, a *webspace* is a single cPanel account together with its `public_html` folder. Everything below that folder shares the same space, however many separate domains, subdomains and applications you have pointed at it:

```
./public_html/dev
./public_html/old
./public_html/new
./public_html/anotherdomain.com
./thisdomain.com
./subdomain.domain.com
```

Those are not six sites in any sense the operating system recognises. They are six folders belonging to one user, and that user is the one PHP runs as. They share a filesystem, and very often a single PHP-FPM pool. To the server there is no live site and no dev site, no Joomla and no WordPress, no client A and no client B. There are just folders, and they all belong to the same person. The `.htaccess` on your best-defended site has no vote in any of this, because the file that gets compromised is not going to be on your best-defended site.

The consequence is brutally simple. Whoever compromises the *weakest* site in that account very often gets read and write access to *all* of them, because to the operating system they are all just files owned by the same user. The attacker who cannot find a bug in your fully patched, carefully hardened live Joomla does not sit there hammering it. He enumerates the account, finds `dev.yoursite.com` running a three-year-old Joomla with a known unauthenticated RCE and `test` / `test` credentials, pops it in seconds, and from that foothold reads your live site's `configuration.php`, connects to the very same database, writes a shell into your live document root, or just inserts himself an administrator row. Every `.htaccess` rule on your live site is beside the point, because he never sent a single HTTP request to your live site at all.

Technical detail: one Unix user, every site

On typical shared hosting, all of an account's sites run as a single Unix user, often through one PHP-FPM pool, with `open_basedir` either unset or scoped to the whole account rather than per-vhost. So PHP running on the compromised dev site can `fopen()`, `include()` and `unlink()` files belonging to the live site, because at the OS level they are all the same user's files. The live site's `configuration.php` is readable to that user by necessity, so its database name, user and password are one `file_get_contents()` away, and the database is usually reachable from any site in the account.

No web request is ever made to the live site, so no rule in its `.htaccess`, restrictive or not, is ever consulted. Real isolation has to happen at the hosting layer, with separate users per site, per-site PHP pools, and CageFS-style jails. That is not something any `.htaccess` can provide, and no default Joomla ships can create it.

This bites hardest with development sites, and they deserve singling out, because the pattern is so reliable it is close to a law. A dev site should never share a webspace with a live one. It always goes the same way. The dev copy is spun up in a hurry, it does its job, and then it is abandoned. Nobody updates it, because it is "only a test site" and keeping it patched was never anybody's job. It falls out of date. It turns vulnerable. And because it is sitting in a folder called **`dev`**, or **`new`**, or **`old`**, or **`staging`**, or **`test`**, an attacker's scanner finds it without needing any cleverness whatsoever, since those are the first names on the list it was going to try anyway. Then it is brute force against **`test`**/**`test`**, or a vulnerability disclosed last month that this copy will never be given the patch for, and they are in. Not in the dev site. In the webspace, and your live site is in the webspace.

I have watched this at scale, and the scale is the part people refuse to believe until it happens to them. Some GoDaddy Reseller plans pack hundreds of sites into a single account, which is to say a single shared webspace. In one case, one old and vulnerable site led to more than a hundred sites being hacked at once. Every file was infected with a different hacker string, so there was no pattern to match and no find-and-replace that would clean it. And the cleanup was not just hard, it was impossible in place: not one of those sites could be cleaned until each had been moved into its own isolated webspace first, because anything cleaned would be reinfected within minutes by the compromised sites still sitting beside it. The infection was not in a site. It was in the space.

So the best practice here is one line long, and it is not a rewrite rule. Give every site its own isolated webspace: one site, one account, one Unix user, one **`public_html`**. It

costs more than cramming a dozen sites into the plan you are already paying for, and it is worth every penny, because it is the difference between “hack one, hack all” and “hack one, hack one”.

A default `.htaccess` that restricts entry points does absolutely nothing about this, and it cannot, because the attack never touches your site’s entry points. This is exactly why we care so much about knowing the version and patch state of *every* site in an account, not just the flagship one. The abandoned, unmonitored, unpatched site is not a low-priority afterthought to clean up later. It is very often the actual front door, standing wide open, while you are busy polishing the locks on a door nobody is even trying.

Who decided viagra was a dangerous word?

Here is where the theatre becomes impossible to unsee. Joomla’s own htaccess security examples documentation still recommends a “Basic antispam Filter” that forbids any request whose query string contains a hardcoded list of pharmaceutical and drug words:

```
########## Begin - Basic antispam Filter, by SigSiu.net
## This code uses PCRE and works only with Apache 2.x.
RewriteCond %{QUERY_STRING} \b(ambien|blue\spill|cialis|cocaine|erectile)\
RewriteCond %{QUERY_STRING} \b(erections|hoodia|impotence|levitra|libido)\
RewriteCond %{QUERY_STRING} \b(lipitor|phentermin|tramadol|ultram|valium)\
RewriteCond %{QUERY_STRING} \b(viagra|vicodin|xanax|huronriveracres|troyha
RewriteRule .* - [F]
########## End - Basic antispam Filter
```

That `[F]` means forbidden. Any request whose query string trips one of those words gets a 403, no page, no explanation. Now ask the obvious question that nobody who pasted this rule ever seems to ask. What if your site *is* a pharmacy? A men’s-health clinic? A GP surgery whose article search passes the query as `?q=erectile+dysfunction`? A sexual-health charity? An addiction-recovery service whose visitors are literally searching for “valium”, “xanax” and “tramadol” because those are the things they are trying to stop? Every one of those sites now serves 403

errors on legitimate searches, and the owner has no idea why, because nobody reads the `.htaccess` they pasted in three years ago, and a 403 on a search result is exactly the kind of intermittent failure that never gets diagnosed.

So ask the deeper question. Why does the Joomla project get to decide, by default, which words your visitors are allowed to type into a search box on your own website? Nobody voted for that. It is a values judgement about content, baked into a security file, shipped to everyone. And then look closer at the list, because it gives the game away. `huronriveracres`, `troyhamby`, `sandyauer`, `ypxaieo`, `unicauca`: those are not drugs. They are fossilised referrer-spam and comment-spam tokens from mid-2000s blocklists, from the MT-Blacklist and referrer-spam era, the same corpus that circulated in things like the old "Ultimate htaccess Blacklist". They mean nothing today. They are in a Joomla security example in 2026 because they were copied forward, file to file, site to site, for close to twenty years by people who never questioned a single line of what they were pasting. The block even has the tell "I removed some common words, tweak to your liking", the fingerprint of a hand-me-down list nobody actually owns.

Every blocklist loses to the person you are blocking

There is a deeper problem here, and it does not stop at the pharmaceutical filter. It applies to the user-agent blocks, the bad-referrer blocks, the SQL-keyword blocks, and every other "deny these known-bad strings" rule that these hardening guides ship. **The attacker controls the request, completely.**

A blocklist can only ever contain the bad things you already know about. That is its fundamental, unfixable weakness. The instant an attacker uses a user-agent string, a query word, a referrer or a payload encoding that you did not put on your list, and doing so costs them nothing, a single character change, a different scanner, a fresh string, they sail straight through as if the rule were not there. Blocking `libwww` or `wget` or `python-requests` in your user-agent rules stops only the lazy, and the lazy simply set their user-agent to `Mozilla/5.0` and try again. This is the security anti-pattern that has been understood for decades: you cannot enumerate all the badness in the world, you can only enumerate the tiny, stale subset you have personally heard of, which means a deny-list is permanently one step behind whoever it is pointed at.

Technical detail: why a regex blocklist never holds

Take a filter that blocks the string `union select`. The attacker sends `uni%6fn%20sel%65ct` (URL-encoded), or `union/**/select` (an inline SQL comment splitting the keyword), or `UnIoN SeLeCt` (case variation), or double-URL-encodes it so your rule sees a harmless `%2575nion` that Apache only decodes to `union` after your `RewriteCond` has already passed it, or splits the payload across two request parameters your single-line regex was never going to join up.

The same is true of user-agent and referrer blocks: those are attacker-supplied strings, changed with a one-line header edit. A blocklist is, at best, a list of the disguises you have already seen someone wear. The attacker's entire job is to turn up in one you haven't, and there are infinitely many of those.

And it fails in both directions at once, which is the worst possible outcome. It never stops a competent attacker, because they just pick a string that is not on your list. And it *does* keep blocking your real visitors, your real customers, your real search traffic, because they occasionally trip a fossilised entry that nobody remembers adding and nobody dares remove. No security benefit against anyone who is actually trying, plus a permanent trickle of broken legitimate traffic and mystery 403s. If you are going to control access at all, you allow-list what is known-good, or better still you fix the code so it does not matter what the input is. You do not play whack-a-mole with a list of bad words and call it hardening.

Burning CPU to block a threat PHP deleted in 2013

The same Joomla documentation ships a rule to "Disallow PHP Easter Eggs", complete with references for further reading:

```
## Disallow PHP Easter Eggs (fingerprinting the PHP version).
## See http://www.0php.com/php_easter_egg.php and http://osvdb.org/12184
RewriteCond %{QUERY_STRING} \=PHP[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a
RewriteRule .* - [F]
```

PHP Easter eggs were the magic GUID query strings, like `?=PHPE9568F34-D428-11d2-A769-00AA001ACF42`, that made old PHP versions echo back a hidden logo or the credits

page, which a scanner could use to fingerprint the exact PHP version. There are three separate things wrong with still defending against them in 2026, and together they make this rule a small monument to how `.htaccess` hardening actually ages.

First, the threat no longer exists. PHP removed the Easter eggs entirely in **PHP 5.5, released on 20 June 2013**. The magic GUIDs "no longer have any special functionality" from that release onward. Joomla 4 requires PHP 7.2.5 as an absolute floor, and Joomla 5 and 6 require PHP 8.x. There is no PHP version that any supported Joomla can even run on that still has the feature this rule is guarding against. It is a regular expression evaluated against the query string of every single request, forever, to defend against something that was deleted from the language more than a decade ago.

Second, even back when the Easter eggs were real, a rewrite rule was never the right fix. The correct mitigation was, and is, one line in `php.ini`:

```
expose_php = Off
```

That single directive suppresses the Easter-egg GUIDs *and* the `X-Powered-By: PHP/8.x` response header, at the engine layer, once, with zero per-request cost. It is a server-layer concern, and it belongs in server-layer configuration, not in a per-request regular expression at the back of the stack. The `.htaccess` version does a worse version of the same job, more expensively, on every hit, and only covers the GUID half while leaving the `X-Powered-By` header it was worried about untouched.

Third, the rule's own references are dead. It points you at `osvdb.org/12184` for more information, and OSVDB, the Open Sourced Vulnerability Database, shut down permanently on 5 April 2016. The other reference, `0php.com`, is a long-defunct mid-2000s site. So here is a rule, shipped in current Joomla documentation, that mitigates a class of bug PHP deleted 13 years ago, at the wrong layer, citing two reference sites that no longer exist. That is not an outlier. That is what an awful lot of `.htaccess` hardening amounts to: a pile of rules nobody understands, guarding threats that are gone, copied between sites by people who assume that more rules must mean more safety.

And every one of those dead rules has a running cost. Apache's own documentation is blunt about it: when `.htaccess` is enabled, "httpd will look in every directory for `.htaccess` files", and it does so "whether or not you actually even use them". Worse, it walks the *entire* directory tree to assemble the rules: a request under `/www/htdocs/example/` forces Apache to look for `.htaccess` in `/`, `/www/`, `/www/htdocs/` and `/www/htdocs/example/`, "4 additional file-system accesses, even if none of those files are present", on every single request. More rules, in more files, deeper trees, is slower and occasionally less secure, not more.

What Joomla actually ships, and why nobody reads it

To be fair to Joomla, it does try, and the effort is not the problem. The problem is that the effort reaches nobody, because the thing it produces is unreadable to almost everyone who receives it.

Joomla ships a preconfigured `htaccess.txt` in the site root, which you have to manually rename to `.htaccess`, and which is tied to switching on SEF URLs in Global Configuration, so it is not even active on a default install. In current Joomla that file runs to over 150 lines: mod_rewrite exploit-prevention rules, a custom redirect section, the SEF URL rules for both the API and the front end, GZIP and Brotli compression, and content-type and SVG security headers. The documentation warns you, in bold, not to edit `htaccess.txt` directly, because a Joomla update will overwrite it and discard whatever you changed without telling you. There is a post-installation message announcing new `.htaccess` features, with advice that, in practice, almost nobody follows, because a 150-line `.htaccess` is regex wizardry that only a small minority of Joomla users can actually read, let alone safely modify. The net result is a powerful, fragile file that is shipped disabled, must not be edited where it lives, and is understood by a fraction of the people who depend on it.

There is also a more defensible road the proposal does not take, which is telling in itself. Joomla already ships small, tightly-scoped `.htaccess` files inside individual folders. `libraries/.htaccess`, for one, is three lines that simply deny direct access to that directory. That per-folder approach, a specific deny rule in the specific folder that

needs it, is far sounder than one sprawling regex at the root, and yet there is no real discussion of leaning on it: of shipping targeted deny rules in the handful of writable directories that actually matter, rather than a giant front-door filter. But even the better idea hits the same wall as everything else here. Those subfolder files do nothing at all on Nginx, Caddy, or behind a cache, so on the majority of the web that is no longer plain Apache they are wasted bytes radiating the same false sense of security as the big file at the top.

Akeeba's Admin Tools is the closest anyone in the ecosystem comes to doing this properly, and it deserves credit. Its `.htaccess` Maker gives you checkbox toggles with real documentation attached to each option, so you are choosing behaviours rather than hand-writing PCRE you do not understand. To its further credit, it does not pretend the world is only Apache: it also ships an Nginx Maker and a `web.config` maker. But look closely at what deploying the non-Apache versions actually takes, because it exposes the whole problem. Admin Tools' own Nginx documentation says it in capitals: the generated config has to be manually included in your site's server definition, and then "modifying the NginX configuration has NO EFFECT until you reload or restart the NginX server". A site owner on shared hosting cannot reload Nginx. They have no shell, no root, no access to the server definition file. So the one server family where the average Joomla user can actually deploy the output themselves is Apache, and only because Apache re-reads `.htaccess` on every request with no reload required, which, as we just saw, is the exact behaviour that also makes it a measurable performance drain. Do this properly across web servers and you need server-level access most Joomla users do not have. Do it the easy way and you are back to Apache alone. The best tool in the ecosystem cannot escape the fact that the format only self-deploys on one server.

Admin Tools' HTAccessMaker is good, and still not the answer

I just called it the closest anyone comes to doing this properly, so let me make good on that, because the `.htaccess` its HTAccessMaker writes is very good, and it mutes a

great deal of what I have argued above. I would rather concede that openly than pretend the best version of the thing is as weak as the worst.

It ships none of the fossils. No pharmaceutical blocklist, no dead Easter-egg regex, no twenty-year-old referrer-spam tokens. What you get instead is the short list of jobs a `.htaccess` is actually good at, done cleanly and from checkboxes rather than hand-cut PCRE: force HTTPS, disable directory listings, block direct access to `configuration.php-dist` and `htaccess.txt`, neutralise SVG script execution, set the clickjacking, MIME and reflected-XSS headers, and restrict the administrator area by IP with proper support for reading the real visitor address back from Cloudflare, Sucuri or BunnyCDN. It previews the exact file before writing it and backs the old one up to `.htaccess.admintools` first, which defuses the “one typo and the whole site 500s” objection better than anything else in the ecosystem.

And here is the detail that should settle the argument rather than soften it: Akeeba is honest about the limits in its own documentation. Of the user-agent blocklist it ships, the docs say in capitals that it “IS NOT A SECURITY FEATURE”, because the client controls that header, which is exactly the point I made earlier about every blocklist. When the best tool in the space tells you, in its own manual, that a chunk of what people treat as hardening is not security, this article is not a fringe position. It is the vendor’s own words.

So use it. If you have Admin Tools Professional, its HTAccessMaker is the right way to do the narrow jobs, and I would far rather you ran it than hand-edited anything yourself. But notice the three things it cannot change. It is still a `.htaccess`, so every structural limit above still applies: the file itself cannot see the SQL injection or the unauthenticated upload arriving through `index.php` (Admin Tools’ separate web application firewall can and does detect and block a SQL injection at runtime when you have that feature switched on, but that is the PHP WAF earning the credit, not the `.htaccess` it also generates, the exact distinction I drew earlier), it runs on Apache alone, and a deeper `.htaccess` dropped into your uploads folder still out-nests it. It is a commercial product, a paid Professional feature that the free edition and every non-Apache site never get. And it is only ever as good as the administrator ticking the boxes, which is the quiet catch in any checkbox tool: it can express a decision safely,

but it cannot make the decision for you or understand your server on your behalf. Akeeba says as much, warning that depending on your server settings some options "may be incompatible with your site", and when they are you get "a blank page or an Internal Server Error 500 error page" on every request. Some options assume Apache modules that are not always switched on: a prepackaged stack like WAMPServer ships with `mod_rewrite` off by default, and a rule that needs it will 500 the whole site until someone works out why. The preview, the backup and the delete-it-over-FTP escape hatch all exist because a wrong choice takes the site down.

A superb `.htaccess` is still a `.htaccess`. HTAccessMaker makes the narrow layer about as good as the narrow layer can be. It does not turn it into a different layer.

Why Joomla code needs direct PHP access

The flip side of "only approved entry points may run" is that a great deal of legitimate software has a real, designed need for direct PHP access somewhere in the webspace, and locking that down by default breaks working sites and working tools.

Start with the assumption the whole proposal rests on: that a webspace is a Joomla-only place. In the real world it almost never is. An agency runs all sorts of things inside one account: a Joomla site, a couple of WordPress installs, a static microsite, a standalone PHP form handler, a scheduled script, an old bespoke app nobody has touched in years. The account is reused for many purposes and Joomla is only one tenant in it. A default that treats every `.php` file in the space as either Joomla's or an attacker's is being narrow-visioned about how these servers actually get used, and it breaks the neighbours it was never entitled to make assumptions about in the first place.

The Joomla project does not have to look far for examples, because its own ecosystem is full of them, which is why the counter-examples in that entry-point discussion came from the most respected developers in the community, not from cowboys. We already met the two sharpest cases at the top of this article: Akeeba's `kickstart.php`, a standalone script that has to run *outside* Joomla because it rebuilds a site where Joomla may be broken or absent, and Joomla's own updater handing `index.php`

straight to `extract.php` to unpack a new copy of the very software it is replacing. Those are not sloppy design. They are recovery tools doing the one job you cannot route through a running Joomla, and a blanket "only approved entry points may run" rule breaks them by definition. That is the awkward truth sitting underneath the proposal: the direct-access pattern it wants to forbid is one the project itself depends on.

Third-party developers hit the softer version of this constantly. Explaining to a customer that they must edit a `.htaccess` to whitelist an endpoint, just so a plugin's own PHP file can be reached directly, is a hard support conversation, and the framing that they must "downgrade their security" to make the plugin work is misleading and unfair. It implies the plugin is insecure, when the direct access may be a perfectly safe, intended feature of how it is built. The access is part of the design, not a hole in it, and there is a real difference between a file that is reachable and a file that is *exploitable*.

Tools like mySites.guru sit squarely in this category, and here the direct access is not a convenience, it is the entire point. The connector is written to be semi-independent of Joomla so that it keeps working when Joomla does not, and that independence matters most at the exact moment you need it: when the site is hacked, when it is down, when core files have been deleted, when the database is unreachable, when Joomla is throwing a fatal error on every page. An external monitor that could only ever talk to you *through* a healthy Joomla would go blind at the precise instant Joomla broke, which is the one moment you most need eyes on the site. Direct access to its own dedicated endpoint is what lets an independent watcher stay independent of the thing it is watching. Forbid that by default and you do not make the site more secure, you blind the tools that were going to tell you it had been attacked.

.htaccess is an Apache feature, not a web standard

Step back far enough and the whole argument looks shakier still, because `.htaccess` is not a standard at all. It is an Apache implementation detail, one web server's particular way of doing per-directory configuration, and Apache is no longer how most of the web is served. Plenty of houses have no fridge to lock and no toilet door to bolt.

Nginx, Caddy, Cloudflare, Node and every other server that never adopted Apache's per-directory model are all perfectly happy homes that simply do not have the fixture you are selling a lock for, and no amount of insisting on the lock will conjure one into their kitchen.

By W3Techs' current figures, both Nginx and Cloudflare now sit above Apache in web-server usage, with LiteSpeed close behind. Roughly three quarters of the web is not "plain Apache reading your `.htaccess` on every request", and of the major players:

- **Nginx** does not read `.htaccess` at all, and this is by design, not an oversight. Its own documentation states that Nginx does not support per-directory configuration files, so every rule has to be converted into central configuration that is loaded once, not re-read from the filesystem on every request.
- **Cloudflare** sits in front of your origin entirely, and is configured at the edge through its own rules engine. It has never heard of your `.htaccess` and never will.
- **Caddy** serves automatic HTTPS and HTTP/2 out of the box from a single central Caddyfile, with no per-directory model. Its maintainers have explained why they will not add `.htaccess` support: it would mean building an Apache configuration parser in Go, recursively scanning directories for these files on every request, and running dynamic handlers based on their contents, all of which runs directly counter to how a modern, performant server is designed. It is not a small ask and it is not on the roadmap.
- **LiteSpeed** does read `.htaccess`, and this is the one that confuses people into thinking the format is a standard. It is not. LiteSpeed chose to honour Apache's rewrite rules as a compatibility convenience, to make migration off Apache painless. Those are still Apache configuration files being tolerated by a different server, not a neutral cross-server standard that everyone agreed to implement.
- **IIS** uses `web.config`, which Joomla ships as `web.config.txt`, and which, by the Joomla team's own admission, has weaker settings and lags behind the `.htaccess` rules.

And even that asterisk on LiteSpeed needs an asterisk of its own. LiteSpeed Enterprise does re-implement Apache's rewrite engine and honours a good deal of a `.htaccess`:

rewrite rules, headers, expiry, basic auth, `php_value`, re-read from the file per request much as Apache does it. But the compatibility is partial, and worse, it fails with no error. Per-directory `mod_security` rules are ignored, `mod_deflate` and gzip directives are dropped in favour of native vhost settings, `SetEnv` values may never reach PHP, and a whole list of Apache modules is passed over without a word. A rule that protects you on Apache can sit in the exact same file on LiteSpeed doing nothing at all, with no error to tell you it has stopped working. The free, open-source sibling is worse still: in practice OpenLiteSpeed does not read `.htaccess` files at all, treating that as an Enterprise-only feature, while its admin panel cheerfully offers `.htaccess` options that produce no result, so your rewrites have to be hand-configured as static contexts or they simply never run. "LiteSpeed supports `.htaccess`" turns out to mean one paid edition supports most of it and discards the rest with no error, while the free edition barely reads it. A security file whose rules half-apply depending on which build of which non-Apache server happens to be underneath is the exact opposite of a control you can rely on.

That is the point people miss when they treat the `.htaccess` as "the security file". More than one server tolerating a format does not promote it to a standard; it just means Apache's format leaked into a couple of neighbours. Anything you build on top of it is, by definition, Apache-specific, and increasingly a minority case.

WordPress, for comparison, powers an enormous share of the web on a default `.htaccess` of just seven rules:

```
# BEGIN WordPress
RewriteEngine On
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
# END WordPress
```

Seven rules. No pharmaceutical blocklist, no Easter-egg regex, no fossilised referrer-spam tokens. WordPress leans on secure-by-design core and its plugin ecosystem, for better and for worse, rather than pretending the rewrite file is a firewall. It is not that WordPress got security right and Joomla got it wrong; both have plenty of plugin and extension vulnerabilities. It is that WordPress never asked its `.htaccess` to do a job the `.htaccess` cannot do.

The ground is shifting under Apache, and Cloudflare is hiding how fast

I quoted a single snapshot of those figures earlier. Look at the twelve-year trend instead and the direction is unmistakable. As I write this, W3Techs has Nginx at 31.5%, Cloudflare's own server at 29.0%, Apache at 23.1% and LiteSpeed at 15.1%. A year ago Apache was on 26.0% and Cloudflare on 24.0%: in twelve months Apache shed nearly three points while Cloudflare gained five, and that is not a blip, it is the shape of the whole decade. Apache sliding, Nginx and the edge rising. The one server the entire `.htaccess` argument depends on is the one steadily losing ground.

But those figures deserve less confidence than the people quoting them tend to give them, mine included, and the reason cuts against my own argument as hard as it cuts for it. These surveys identify a site's web server from the `Server` response header, and Cloudflare rewrites that header to `cloudflare` for every site it proxies, whatever is actually running at the origin behind it. A site served by Apache behind Cloudflare is therefore counted as Cloudflare, not Apache. Every Apache origin hiding inside that 29% Cloudflare bucket is an Apache install the survey never sees. The consequence is that Apache is almost certainly more common than 23% makes it look, quite possibly more common than Nginx, and we simply cannot know by how much, because the proxy that masks it now fronts more than a fifth of the web. We will never really know the true numbers.

Which sounds like it rescues the `.htaccess`, and does the opposite. Every one of those hidden Apache sites has a Cloudflare edge sitting in front of it, and that helps the file not at all: the edge can answer a request from cache without it ever reaching

Apache or its `.htaccess`, and the edge is where the real hardening belongs, the very layer Joomla offers not one line of guidance for. The masking does not prove your `.htaccess` still matters. It proves it now sits even further back, behind even more that runs before it, than the raw share numbers admit.

On a fresh Apache server, your .htaccess is ignored by default

Here is a fact that surprises most Joomla site owners: on a brand-new Apache install, your `.htaccess` does nothing at all. Not “less than you think”, literally nothing. Since Apache 2.3.9, released back in 2011, the shipped default has been `AllowOverride None`, which tells Apache not to read `.htaccess` files anywhere on the server. Every rule you carefully wrote, every SEF rewrite, every hardening directive, every deny rule, is sitting in a file Apache has been explicitly instructed never to open.

Those rules only begin to work the moment a server administrator opens the main Apache configuration, `httpd.conf`, a vhost file, or `apache2.conf` on Debian and Ubuntu, and changes the `<Directory>` block covering your document root to `AllowOverride All`, or at least to the specific override classes your rules require. This is exactly why “I renamed `htaccess.txt` to `.htaccess` and my SEF URLs still don’t work” is one of the most common questions a new-server Joomla install produces. The rewrites are not broken and the file is not wrong. Apache was simply never told to read it, so it doesn’t.

Sit with what that means for the mental model. The single most important decision about whether your `.htaccess` runs at all is not made in your `.htaccess`. It is made in server configuration that you usually cannot see, that on shared hosting you cannot change, and about which you are never notified. Your “security file” is opt-in, off by default on a fresh server, and someone else entirely holds the switch.

Technical detail: AllowOverride and the override classes

`AllowOverride` lives in a `<Directory>` block in the main Apache config, never in the `.htaccess` itself, and its default since Apache 2.3.9 (2011) is `None`, meaning matching `.htaccess` files are not even opened. To make Joomla’s SEF rewrites run you need at least

`AllowOverride FileInfo` (which permits `mod_rewrite` directives); a full hardening `.htaccess` typically also needs `Options`, `Limit` and `AuthConfig`, which is why hosts usually just grant `AllowOverride All` and be done with it.

On managed shared hosting the control panel sets this for you, so you never see it. On an unmanaged server or a fresh VPS, nothing does, so a freshly renamed `.htaccess` is completely inert until you edit the server config by hand. And because `AllowOverride` is evaluated per-directory from the top of the tree down, a single broad `AllowOverride None` higher up disables every `.htaccess` beneath it in one line.

And the switch flips both ways, at times you do not choose. A host tuning for performance may set **`AllowOverride None`** across the whole box, because Apache's own documentation recommends exactly that, moving rules up into the main config, for speed, and in doing so retires every **`.htaccess`** on the server. A migration to a new server, a control-panel rebuild, a move onto a hardened base image, any of these can reset **`AllowOverride`** to its default and switch your entire hardening layer off without a single character of your **`.htaccess`** ever changing. You would not see an error. The rules would simply stop applying, and unless you were actively testing for them, you would never notice. A security control that defaults to off, and that someone else can switch off again at any time without telling you, is not a foundation to build on.

Your .htaccess is buried behind proxies you don't control

There is one more layer to this that almost nobody accounts for, and it has become the norm rather than the exception. Most serious sites no longer serve requests from a single Apache instance. They serve them from a stack.

A very common modern arrangement is Cloudflare in front of Nginx in front of Apache with PHP-FPM behind it, or Nginx as a caching reverse proxy in front of Apache, which is now a standard cPanel configuration. In every one of those stacks, your Apache **`.htaccess`** sits at the very back, and it only has any say once a request has already travelled through everything in front of it. It governs one hop of the journey, the Apache leg, and it is blind and powerless for the rest.

Two consequences follow, and both undercut the mental model of "my **`.htaccess`** guards the front door". First, whatever the front proxy allows through reaches Apache

regardless of what your `.htaccess` would have wanted at the edge; the `.htaccess` cannot filter a request before it has been proxied to Apache, because it is not the first thing the request meets, it is nearly the last. Second, and more damaging, a request that the front layer can answer from cache never reaches Apache at all. If Cloudflare or the Nginx reverse-proxy cache has a copy of the response, it serves it directly, and your `.htaccess` does not execute: not its redirects, not its blocks, not its security headers, not its carefully crafted deny rules. A security rule that only runs on cache misses is a security rule with holes in it that move around depending on what is currently cached.

So the picture people have in their heads, a single gatekeeper `.htaccess` standing at the entrance inspecting every visitor, is wrong twice over on a modern stack. There are one, two, sometimes three layers in front of it that it has no control over and cannot even see, and behind some of those layers, for cached responses, it does not run at all. It is not the front door. It is a door somewhere near the back of the house, that only some visitors ever walk through.

Even hackers use .htaccess, usually better than you do

There is an irony sitting underneath all of this, and it is a real one: attackers are frequently far more fluent in `.htaccess` than the site owners defending with it. To a competent attacker it is not a shield at all. It is one of their favourite tools.

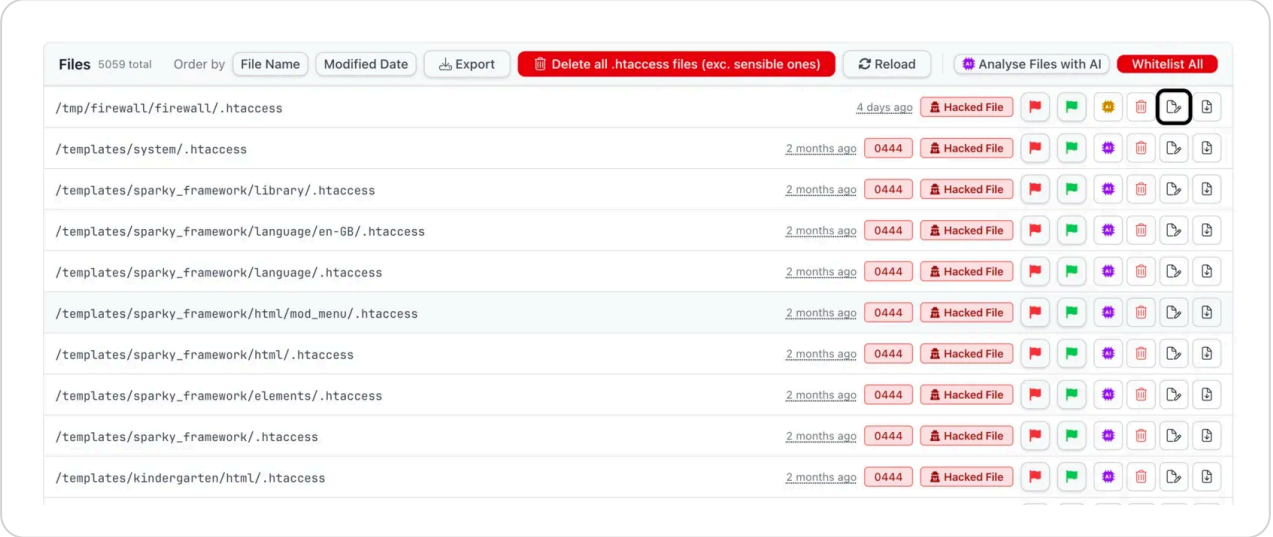
Once they have a foothold, a `.htaccess` is how they make themselves comfortable. They drop their own into an uploads folder to force PHP processing where your rules forbade it, so that the innocuous-looking `.jpg` they uploaded runs as code. They plant one in a deep subdirectory to cloak and redirect: real visitors and Googlebot get sent to a spam or malware network, while you, visiting your own homepage, are served a clean page and see nothing wrong for months. They use it to password-protect the directory holding their own backdoor, so that even if you find the folder you cannot easily see inside it. We have cleaned up sites with thousands of malicious .htaccess files that attackers weaponised against the owner, scattered through every writable directory, and we have watched backdoors rebuild every one of those files within minutes of a cleanup, because the entry point that let them in was never actually closed. The same

file you trusted as your defence is, in a compromised site, the attacker's preferred instrument. They understand the format better than you do, and they are using it against you.

Your .htaccess might be the hack itself

Here is the turn that catches people out. On a compromised site, the `.htaccess` is not automatically your file. Some of the most common Joomla infections we clean up right now ship their own, uploaded by the attacker and doing the exact opposite of what you assume a `.htaccess` is for.

We flag every one of them. On the site below, the audit found `.htaccess` files marked as hacked scattered through the template tree and inside a folder the attacker had helpfully named `firewall`, several of them set to `0444` so they are read-only and harder to delete.



Open one and the intent is obvious. This is the contents of that `/tmp/firewall/firewall/.htaccess`:

/tmp/firewall/firewall/.htaccess

```
<FilesMatch ".(py|exe|phtml|php|PHP|Php|PHp|pHp|pHP|pHP7|PHP7|phP|PhP|php5|suspected)$">
Order allow,deny
Deny from all
</FilesMatch>
<FilesMatch "^(index.php|cache.php)$">#
Order allow,deny
Allow from all
</FilesMatch>
```

Read it the way Apache does. The first block denies execution to every spelling of a PHP file it can think of, `php`, `PHP`, `pHp`, `php5`, `phtml`, plus `.suspected` and several more. The second block then carves two exceptions back out: `index.php` and `cache.php` are allowed to run. That is not a defence. It is an attacker switching PHP execution off for the whole directory and back on again only for their own two droppers. Every scanner's shell, every rival attacker's backdoor, every file but theirs is now inert, while their `index.php` and `cache.php` keep serving. It is persistence and turf control, written in the exact syntax you were told to trust.

This is why we treat a new or changed `.htaccess` as something to audit, not something to believe. The file that is supposed to be your defence is, often enough, the clearest signal that the site is already owned.

More reasons the .htaccess is the wrong thing to trust

Beyond the headline problems, a pile of smaller ones compounds, and any single one of them should give you pause before you treat this file as your security.

One bad line takes the whole site down

A `.htaccess` is parsed on every request, and if it contains a single directive Apache cannot honour, a rule from a module the host has not loaded, a typo, a directive the host has explicitly forbidden, Apache does not skip it and move on. It returns a `500 Internal Server Error` for every request to that directory and everything beneath it: your pages, your images, your CSS, all of it, gone at once. The file you added to make

the site safer is one fat-fingered line away from taking the entire site offline. It fails to *broken*, not to *safe*, which is the opposite of what a security control should do.

> **Technical detail: how one line 500s everything**
>
> Apache parses the applicable `.htaccess` files while it assembles the response, so a directive it cannot process is a fatal parse error for that request, not a skipped line. Add `Header set X-Frame-Options SAMEORIGIN` where `mod_headers` is not loaded, a `php_value` where PHP runs as FPM or CGI rather than `mod_php`, or a `SetEnv` where `mod_env` is absent, and every request under that directory returns `500 Internal Server Error`, including static assets that never touch Joomla at all.
>
> There is no partial application and no graceful fallback: the entire file is rejected, so a rule you pasted in for security can black-hole the whole site until someone reads the Apache error log and finds the one offending line. A security file that can take the site down on a typo is a liability sitting inside your defence.

Your ruleset is downloadable, and it maps your defences

Every editor and hardening tool leaves backups behind: `.htaccess.bak`, `.htaccess.orig`, `.htaccess.save`, cPanel's own copies, Akeeba's `.htaccess.admintools`. Those backup files are almost never themselves protected from direct browser access, so an attacker can often just request one and read, in plain text, exactly which paths you guard, which files you hide, and which protections you lean on. Your secret hardening becomes a reconnaissance document, served straight to the person you were trying to hide it from.

It is not in source control, so it drifts unnoticed

The `.htaccess` lives on the server, edited by hand, almost never committed to git, rarely reviewed by anyone. Your dev, staging and production copies drift apart, nobody can diff the current state against a known-good baseline, and Joomla's own warning that a core update will overwrite `htaccess.txt` guarantees the rot. Security configuration you cannot see the history of, and cannot diff, is security configuration you do not really control.

It is blind to the vectors behind most compromises

A phished or reused Super User password, a nulled or pirated extension shipping a backdoor, a legitimate extension whose update server was compromised: all of these arrive through approved, authenticated, entirely expected channels. A large share of real-world Joomla infections begin exactly there, and no `.htaccess` rule has any opinion whatsoever about a valid login or a signed update. You cannot rewrite-rule your way out of a stolen password.

A default lockdown punishes the careful and waves the malware through

This is the sting hiding inside a “secure by default” restrictive `.htaccess`. The extensions that break under it are the well-behaved ones that put their files where they said they would. Malware does not read your whitelist; it writes wherever the process can write. So the rule inconveniences legitimate developers and their users far more than it ever inconveniences the attacker it was aimed at.

A dead rule looks exactly like a working one

`.htaccess` directives are order-sensitive and dependency-sensitive. A hardening rule pasted below the SEF rewrite’s `[L]` flag, or one that quietly needs a module the server has not loaded, simply never fires, and nothing anywhere tells you. You cannot unit-test it, you cannot see it on a dashboard, and “I added the rule” is not the same statement as “the rule runs”. A good share of the hardening people believe they have is dead code they will never find out is dead.

It keeps no record of its own tampering

When an attacker drops or overwrites a malicious `.htaccess`, the file raises no alarm about itself. No log line, no alert, no diff, nothing that says “your security file just changed underneath you”. A control that cannot tell you it has been subverted is the exact opposite of what you want, and it is the whole reason the thing that actually catches these break-ins is file-change monitoring watching the `.htaccess`, not the `.htaccess` watching for itself.

The most dangerous thing your .htaccess does is reassure you

Everything up to here has been about what a `.htaccess` cannot stop. The subtler and more expensive damage is what it makes you *feel*. A hardened `.htaccess` is enormously reassuring, and that reassurance is very nearly unearned, which makes false confidence the real cost of the whole exercise, well ahead of the CPU it burns.

Watch how it plays out in practice. You run Admin Tools, you work through the toggles, you get a satisfying "your site is hardened" summary, and a warm feeling of having done your security homework settles in. Then the critical extension update arrives, the one that actually closes the hole an attacker is about to use, and the urgency to apply it is gone, because you are hardened now. The update can wait until the weekend, or the end of the sprint, or the next time you happen to log in. The hardening did not make you safer. It made you slower to do the one thing, patching, that would have. You swapped a small, real risk reduction for a large, invisible increase in how long you leave known holes standing open. That is a bad trade, and it is a trade almost nobody realises they are making.

It gets worse when the reassurance is institutional rather than personal, which is what a "secure by default" `.htaccess` shipped in Joomla core would become. Push a stricter default onto a million installs and you also push a message to a million site owners: security is handled, the project has taken care of it, you can relax. For the one narrow risk the rule addresses, that message is true. For the SQL injection, the unauthenticated upload, the add-an-admin, the abandoned dev site next door, everything that actually gets sites owned, it is false, and now it is being broadcast at scale with the authority of the core team behind it. A green tick that covers a sliver of your real risk but reads as "you are protected" is not neutral. It actively suppresses the behaviour, patching and monitoring, that covers everything else.

> **Warning: hardening is not the same as being patched**
>
> The most common pattern we see in the hours before a compromise is not an unhardened site. It is a hardened site running an extension three versions behind, whose owner deferred the update because the site "was secure". Hardening and patching are not substitutes for one another. If you only have time for one this week, patch: the update actually closes the

hole, while the `.htaccess`, at its very best, makes one narrow corner of the site slightly less convenient to abuse after the break-in has already happened.

The false confidence also starves the one thing that actually catches a breach. If you believe your **`.htaccess`** is standing between attackers and your site, you will not spend money or attention on file-change monitoring, because why would you, the wall is up. So when an attack routes through **`index.php`** past every rule you own, and it will, nothing is watching for the shell it drops. The compromise your hardening was never going to prevent now also goes undetected, because your hardening persuaded you detection was unnecessary. That is how a site ends up serving pharmacy spam to Google for six months: not because nobody hardened it, but because somebody hardened it and then stopped looking.

Real security is uncomfortable on purpose. It assumes the wall will be bypassed, it keeps patching as though there were no wall, and it watches for the breach as though it were inevitable, because it is. A **`.htaccess`** that lulls you out of doing those three things has cost you far more than it ever protected.

And if you are thinking that this is why you bought a firewall rather than relying on a text file, the next two sections are for you, because the trap has a second door.

Not all WAFs are equal - the most popular, by definition, slow down your sites a little!

Earlier I gave a web application firewall the credit for the one block that actually happened, and said it was the WAF rather than the **`.htaccess`** that caught it. So it is worth being as precise about firewalls as I have tried to be about everything else here. “WAF” is one word covering several quite different things sitting at different layers of the stack, and the layer decides both what the thing can catch and what it costs you to run.

Start with the kind most Joomla sites actually run, because it is the kind that gets sold to them: a PHP web application firewall, installed into Joomla as a component or a system plugin. It is the most popular sort by a distance, and it slows your site down by

definition. That is a structural fact about where it sits rather than a criticism of how it is written. To inspect a request at all it must first be running, which means the PHP process has already started and Joomla's framework has already booted far enough to load system plugins, which is most of the fixed cost of a page view before any of your actual content is involved. Only then does the rule set run: a long list of regular expressions, evaluated in loops, against every parameter, every header and every cookie, on every single request that arrives, including the overwhelming majority that were always perfectly ordinary.

That cost is not incidental to the protection, it *is* the protection. The reason a PHP WAF can catch a SQL injection payload that your `.htaccess` never could is that it is deep enough inside the request to see the parameter values after Joomla has parsed them, and being that deep is exactly what makes it expensive. Every request pays for the inspection, and your customer's request pays the same as the attacker's. Popularity then compounds it: the most-installed security extension in the ecosystem is, by definition, the one charging that per-request tax to the most sites.

The best place for a web application firewall is the server layer, or better still the edge, in front of your server entirely: ModSecurity as an Apache or Nginx module, or a real edge WAF like Cloudflare. Put it at the edge and a malicious request is inspected and rejected in a datacentre near the attacker, before it has touched your server at all, let alone started a PHP process or booted Joomla. Your origin spends nothing on it, and your legitimate visitors are not paying a regex tax on every page view for the privilege. It also cannot be got at: a WAF at the edge sits outside the folder tree, outside the application, and outside the reach of whoever compromises the site. It is not a checkbox on a settings screen inside the very thing being attacked, so it cannot be unticked by a frustrated administrator, switched off by a rogue Super User, or overridden by a dropped `.htaccess`.

Technical detail: how far a request travels before each WAF sees it

An edge WAF inspects the request at a datacentre near the visitor and drops it there: your server never receives it, so there is no connection to your origin, no Apache worker, no PHP process and no database query. A server-level WAF such as ModSecurity sees the request once it has reached your box and been parsed by the web server, and inspects it in compiled C before PHP is involved, costing you a little CPU and no PHP. A PHP WAF inside Joomla

sees it last of all, once your server has accepted the request, started a PHP process, and booted the framework far enough to run system plugins.

So the further back the firewall sits, the more of your stack an attacker gets to spend before being told no, and the more of it your real visitors spend too.

A Joomla extension WAF only sees what index.php sends it, and the hacker's most important requests never go near it

The layer problem has a sibling, and this one is worse, because it is a coverage problem. A web application firewall installed as a Joomla extension is not the god layer you are looking for, and the reason has nothing to do with the quality of its rules. It only ever inspects the requests that Joomla hands it. It is a system plugin. It runs because Joomla ran. Its entire field of view is the set of requests that were routed through `index.php` and got far enough into the application to fire a plugin event. Anything that reaches PHP by another path is, to that firewall, silent.

Take our own connector, because it is the cleanest example I can give you and it is one I am responsible for. It is a PHP file sitting in a subfolder of your site, and it is requested directly, by design, for a reason worth saying out loud: it has to keep working on the day Joomla does not. When mySites.guru talks to that file, the request never touches `index.php`, never boots the Joomla application, and never fires a plugin event. It is therefore never seen by RSFirewall!, never seen by Admin Tools' WAF, and never seen by any other PHP firewall installed as a Joomla extension. Those firewalls are not ignoring the request. They are not running at all. Nothing woke them up.

That is fine for us, because the connector authenticates every request itself and is built to stand on its own, which is the same standard I am asking you to hold everything else to. The point is what it proves about the firewall. There is a live PHP endpoint in your webspace, being requested every single day, that your security extension has no visibility of whatsoever. And if that is true of a file you installed on purpose and know about, ask yourself what else it is true of.

The answer is: every directly-requested `.php` file on the site. Akeeba's `kickstart.php`. The updater's `extract.php`. Any extension that ships a standalone endpoint. Any leftover script from an old installation, a migration tool someone uploaded once, a phpinfo file from a debugging session in 2019, the whole forgotten dev copy sitting in `./public_html/old`. None of it routes through `index.php`, so none of it is inspected, and the firewall's dashboard will still cheerfully report that it is protecting your site.

And then there is the one that actually costs you the site. Go back to the two-stage attack. Stage one is the exploit, and yes, that usually does arrive through `index.php`, which is why a PHP WAF can sometimes catch it, as one did for us. Stage two is the attacker requesting the file they just planted: a plain GET to `/images/evil.php`. That request does not go through `index.php`. It does not boot Joomla. It does not fire a plugin event. Your WAF is not consulted and cannot be. So the firewall you are trusting gets exactly one chance to stop the whole attack, the request that plants the shell, and it only has to miss once. From that moment it is blind to every request the attacker makes for as long as they care to keep making them. The backdoor they run commands through for the next six months is invisible to your web application firewall, because your web application firewall lives inside the application, and the backdoor does not need the application.

Yes, I know about auto_prepend_file, and no, it does not rescue this

Some firewalls can hook every PHP script rather than only the ones Joomla routes, using PHP's `auto_prepend_file` directive to force their own entry point to execute before any script on the site runs. Credit where it is due: that is the right instinct, and it does close the blind spot described above. The shell at `/images/evil.php` would then be inspected too, because the firewall runs before it does. There are two problems with reaching for it as the answer.

The first is the bill, and it is the previous section's argument made worse. You have gone from "run the whole rule engine on every request Joomla handles" to "run the whole rule engine before every PHP file on the server executes", including plenty that Joomla knows nothing about and that were never worth inspecting. Everything on the box gets slower, not just Joomla. The second problem is the one that settles it: you can only do this if you are permitted to. `auto_prepend_file` is set in `php.ini`, in a `.user.ini`, or through `php_value` in a `.htaccess` under `mod_php`, and on most shared hosting you have no `php.ini` of your own, `php_value` 500s the entire site the moment PHP is running as FPM or CGI rather than `mod_php` (which is now the norm), and whether `.user.ini` is exposed to you at all is your host's decision, not yours. So the fix for the extension WAF's blind spot turns out to need

exactly the server-layer access that most Joomla site owners do not have. And if you *do* have that access, the question answers itself: you had the server layer all along, so put the firewall there in the first place, where it is cheaper, sees everything, and cannot be switched off from inside the site.

So the question worth asking out loud about any security extension is a coverage question rather than a rules question: what proportion of the PHP that can execute on your server does this thing actually see? For a Joomla extension WAF the answer is "the proportion that goes through `index.php`", and that excludes every standalone endpoint, every forgotten script, every file belonging to the neighbouring site in your webspace, and every request to the backdoor once it exists. A WAF at the server or the edge has none of those blind spots, for the same structural reason it costs you nothing: it is in front of the server, so it sees every request that arrives, no matter which file the request was for or whether Joomla was ever going to run.

To be clear, because these two sections have been hard on them: none of this is a case against web application firewalls. Like the `.htaccess`, they have a real place, and I would far rather you ran one than did not. A PHP WAF inside Joomla still sees things nothing else in your stack can see, it is still the thing that blocked a live SQL injection for us, and on a shared host with no edge and no ModSecurity available to you it may be the only firewall you are permitted to have. Run it.

The point is the same one, aimed at a different object: know what your protection actually covers, and do not let it do your thinking for you. How a Joomla extension WAF really works deserves an article of its own, and if I write it, a fair amount of what is above will turn up in it again almost word for word, because the same trap is sitting there waiting. A firewall that shows you a green dashboard while a direct request to a PHP file in a subfolder strolls past it unexamined is selling you exactly what a hardened `.htaccess` sells: the feeling of having been protected, which has never been the same thing as the fact of it.

Secure by design beats a bigger moat

So what is the alternative to piling rules onto the gate? Stop thinking about moats, and start thinking about the house.

> **Why I don't lock the garden gate**
>
> I don’t lock my garden gate when I go on holiday. Not because I have dug a moat and padlocked the gate, but because the house itself is in order and secure by design. Windows closed and locked. Doors deadbolted. Lights on a timer. Post office told to hold the mail, so no parcels pile up on the porch announcing to the whole street that nobody is home.

A `.htaccess` bristling with rules is the chained garden gate and the padlock on the fridge. It looks impressive from the pavement and it changes almost nothing, because the way in was never the gate, and the prize was never the beer. The burglar walks up the drive that has to be there, past the gate, to the front door that has to open. Secure by design is the house: a core and extension stack that validates its own input, checks its own permissions, casts its own integers, and rejects unsafe uploads in its own code, so that it does not *matter* how a request arrived or what it carried.

This is exactly why some properly secure architectures need direct access and are not weakened by it. When we built the mySites.guru API around OAuth2, PKCE and per-request access checks, the security lives in the design of the thing, in what every request is required to prove before anything happens, not in a wall of `.htaccess` restrictions bolted on around the outside. And there is a nuance most hardening guides miss entirely: some Joomla exploits only became remote code execution on the newest Joomla, because every earlier version blocked the unsafe upload in core by default. The thing that actually protected those older sites was secure-by-design core behaviour, an upload filter in Joomla’s own code, not a line in anyone’s `.htaccess`.

Friction is the other reason lockdown fails in the real world, and it is worth saying because the Joomla community is about to walk into it. We learned this lesson the hard way with two-factor authentication and passkeys: pile on enough friction and frustrated users switch the protection off entirely, choosing convenience over a safety measure that keeps getting in their way. This is where the fridge lock finally gives itself away.

Hunting for your keys and scanning your eyeball every time you want milk for your coffee is a cost you pay every single morning, and the burglar it defends against is hypothetical and has not turned up yet. Nobody keeps that up. The lock gets propped open, then taken off, and the person who removes it is not being careless, they are being rational about a tax they pay daily for a benefit they have never once observed. A `.htaccess` so aggressive that it 403s your own staff, breaks a legitimate extension, or blocks a real customer search, gets loosened, commented out, or deleted the first time it causes a support ticket. Security that people turn off is not security. It is a feature with a short half-life.

Real security lives in the server layer, not the app's folder tree

There is a deeper architectural point underneath everything above, and once you see it you cannot unsee it. A `.htaccess` is a configuration file that lives *inside your application's folder tree*, in the same browsable directories as your images and your PHP. That is the wrong place for security to live, and the whole industry has spent the last decade moving configuration in the opposite direction: out of the app tree, into the server, the environment, and the platform.

We already learned this lesson with plain configuration, long before anyone argued about security. The Twelve-Factor App, about as close as modern web development has to a shared rulebook, says config belongs in *the environment*, injected by the server or the process, not in files that sit alongside your code. The reasoning is the same one: a file in the app tree can be read, checked in, copied, downloaded, or edited by the wrong person. A `.env` file dropped into your document root is the textbook anti-pattern, a plain-text file full of database passwords sitting in the web-served tree, and the thing most often "protecting" it from being downloaded is a `.htaccess` rule, one that can be absent, mis-ordered, or 500 the whole site on a bad line. You end up using one app-tree file to guard another, when the real answer is to keep the secrets out of the tree entirely.

Technical detail: where config and controls actually belong

The Twelve-Factor App's config rule is to store configuration in environment variables set by the deploy environment, not in files committed with the code, because files in the tree get read, served, or checked in by accident. In practice that means credentials live in Apache `SetEnv` / `SetEnvIf` at the vhost level, Nginx `fastcgi_param`, a systemd `Environment=` line, or a container's env, none of which sit inside the web-served folder tree at all.

The containment controls that actually limit a breach live at that same layer and are equally out of a `.htaccess`'s reach: per-account Unix users and per-site PHP-FPM pools, `open_basedir`, CageFS jails, a read-only application filesystem, and firewall rules. You cannot express a single one of them as a directive in a file in your webroot, which is exactly why they hold: an attacker who reaches your folder tree still cannot rewrite the rules that are keeping them boxed in.

Set those same values at the server layer instead and they never touch the browsable tree. Now apply the identical principle to *security*, because it is the identical principle. The controls that actually contain a Joomla compromise all live at the server layer, and none of them are things you can write in a file in your webroot: separate Unix users and separate PHP-FPM pools per site, **`open_basedir`** and CageFS jails, a read-only application filesystem, network firewall rules, a real WAF at the edge. These are what stop a break-in on one site from reading every other site's **`configuration.php`**, what stop a web shell from spawning a system shell, what stop a leaked credential from reaching the database. Every one of them sits at the layer a **`.htaccess`** cannot touch. The closer a control lives to the server and the further from your browsable folder tree, the harder it is to read, to tamper with, or to expose by accident, and the more it is actually worth.

One server-layer setting is worth calling out because it is the exception rather than an example. **`disable_functions`** in **`php.ini`** looks like a real control and is really a blocklist, which means it inherits every weakness this article has already pinned on blocklists, only moved to the server. It does nothing until an attacker is already executing PHP on your box, at which point, by definition, you are already compromised. And once they are in, disabling **`exec`** and **`system`** just sends them to **`proc_open`**, **`popen`**, **`pcntl_exec`**, an **`LD_PRELOAD`** trick, or simply writing a file and letting something else run it. It is the fridge lock of the server layer: it inconveniences a burglar who is already standing in your kitchen, and only until he notices the jewellery is in the next room. Worth having as a last thin layer, never mistaken for the thing keeping him out.

Now here is the uncomfortable part, and it is the reason `.htaccess` hardening will not simply go away. Joomla is mass-market software, and most Joomla sites live on shared hosting where the owner has no access to the server layer whatsoever: no vhost config, no systemd, no firewall, no per-account isolation they control, often not even a real `php.ini` of their own. For those users the `.htaccess` is not a bad choice among good ones. It is the *only* lever they are given, the single scrap of server-adjacent configuration a shared-hosting customer is allowed to touch. So the stubborn persistence of `.htaccess` hardening is not really evidence that it is good security. It is evidence that the hosting model has locked users out of the layer where security actually lives, and left them a per-directory file in the app tree as a consolation prize. That is a real trade-off, and it is why a "secure by default" `.htaccess` is such a tempting idea for a project like Joomla: for a huge share of its users, that file is the only surface the project can reach at all.

But recognising the trade-off is not the same as pretending the file is something it is not. If shared hosting is your reality, use the `.htaccess` for what it is worth, and keep your expectations of it low. When you *can* choose, choose the layer where security actually lives: hosting that gives you real per-site isolation, configuration kept in the environment rather than in files in the tree, and application code that defends itself so it stops mattering which folder an attacker reaches. A bigger, cleverer file in the folder tree is not the destination. It is what you settle for when you have been locked out of the room where the real controls are.

So should you delete your Joomla .htaccess?

No. This is not an argument to throw the file away, and I want to be completely clear about that, because "your `.htaccess` is not security" is easy to misread as "your `.htaccess` is worthless". It is not worthless. It is just not what you think it is.

Keep your `.htaccess`, and keep it doing the jobs it is good at.

What a .htaccess is good for

Forcing HTTPS and HSTS. Blocking direct browser access to sensitive files like configuration.php, .git and backup archives. Handling redirects and canonical URLs. Setting sensible security response headers. A sane hardening baseline from a tool like Admin Tools, as one thin layer of defence in depth. These are all worth doing. None of them is the same thing as stopping a vulnerability.

The distinction that matters is between using the `.htaccess` for those narrow, real jobs, and mistaking it for the thing that stands between an attacker and a bug. It is a safety net for a specific, limited set of problems. It is not the perimeter, it is not a firewall, and it is never the thing that fixes the vulnerable code itself.

If you want the order of priority that actually keeps Joomla sites standing, it looks like this, and notice where hardening the `.htaccess` sits:

1. **Run current, well-maintained code.** The overwhelming majority of real-world compromises exploit a known, already-patched bug in an outdated extension. Patching promptly is worth more than every rewrite rule you could ever write, combined. This is the single highest-value thing you can do.
2. **Prefer secure-by-design extensions** that validate their own input, cast their own integers, and check their own permissions, rather than extensions that assume the server will catch what they let through. When you evaluate a plugin, this is what you are really evaluating.
3. **Watch for change.** When an attack does route through `index.php`, and it will, the tell is a file appearing or changing that should not have. Real-time file-modification alerting catches the shell the moment it is written, and a suspect-content scan surfaces the ones already sitting there from before you were watching.
4. **Then, and only then, harden the .htaccess** as a thin extra layer, with a clear-eyed understanding of exactly how little of your actual risk it addresses.

Do all four and the `.htaccess` is a useful part of a real defence. Do only the fourth, and you have a chained gate on an open house.

How mySites.guru watches regardless of your .htaccess

Because the real attacks come through approved endpoints, the defence that matters is not another wall at a door the attacker was never going to try. It is knowing, immediately, when something on the other side of that door has changed. That is a monitoring problem, not a rewrite-rule problem, and it is the problem mySites.guru is built to solve.

mySites.guru indexes every extension on every connected site and cross-references their versions against vulnerability data twice a day, so you find out that a site is running a component with a known flaw before an attacker's scanner does, with no `.htaccess` rule required and nothing to hand-configure. It watches your files for modification and alerts you when one changes unexpectedly, which is how you catch a web shell that rode in through `index.php` past every rewrite rule you own, in the minutes after it appears rather than the weeks after your traffic collapses. And when the worst does happen, the connector keeps reporting even if Joomla itself is flat on its back, because it was built from the start to be independent of the site it is monitoring.

If you are not sure what state your sites are actually in, run a free audit on any Joomla site and see what it turns up. If you manage a portfolio rather than a single site, mySites.guru lets you watch and update dozens or hundreds of Joomla sites from one place, spotting the vulnerable version across all of them and pushing the fix in one operation. And if one of them is already compromised, the Joomla hacked guide walks you through cleaning it up properly, without destroying the evidence you will want later.

Harden your `.htaccess` if you like. Just do not go on holiday with the gate chained, a padlock on the fridge, and every window in the house wide open.

Further reading

- Joomla htaccess examples (security) - the official documentation, including the antispam and Easter-egg rules dissected above
- When (not) to use .htaccess files - Apache's own guidance on the per-request performance cost of `.htaccess`

- Nginx does not support per-directory config files - Nginx's own guide on migrating Apache `.htaccess` rules into central configuration
- Caddy will not support .htaccess (issue #4091) - the maintainers on why it is neither feasible nor in keeping with a modern server
- Joomla pull request #48076 - the open proposal to whitelist allowed file types in the media directories
- PHP expose_php directive - the one-line server-layer setting that replaced the Easter-egg rewrite rule
- The Twelve-Factor App: Config - why configuration belongs in the environment, not in files sitting inside the app's folder tree
- Akeeba Admin Tools - the checkbox-driven `.htaccess`, Nginx and web.config makers, the gold standard for doing hardening from inside Joomla
- CVE-2026-23898 on the Joomla Security Centre - the arbitrary file deletion flaw in com_joomlaupdate

Frequently Asked Questions

Does a .htaccess file make my Joomla site secure?

Not in the way most people think. A .htaccess can add useful hardening, but the attacks that actually take Joomla sites down (SQL injection, unauthenticated file uploads) travel through index.php, the one endpoint your .htaccess is designed to allow. For those requests the file is never the thing standing in the way.

Can an attacker delete or overwrite my .htaccess?

Yes. An arbitrary-file-deletion or upload-anywhere vulnerability can remove or replace it in a single request. CVE-2026-23898, an arbitrary file deletion flaw in Joomla's own com_joomlaupdate component, is a real example of a bug that could wipe files the web server can reach, including your .htaccess.

Can an attacker beat my .htaccess PHP block by uploading their own .htaccess?

Yes, and it is easier than deleting yours. Apache merges .htaccess files from the document root down to the requested file and the deepest one wins, so an attacker who can write into your uploads folder drops their own .htaccess there that re-enables PHP execution, overriding your root rule without ever touching it. The only block they cannot reverse is one set at the server layer with php_admin_flag engine off or AllowOverride None on that path, which a .htaccess cannot countermand.

Should I delete my .htaccess then?

No. Keep it for what it is genuinely good at: blocking direct access to sensitive files, redirects, forcing HTTPS. Just do not treat it as your security perimeter. A secure-by-design core and extension stack, plus file-change monitoring, protects you far more than another RewriteCond.

Why does Joomla ship rules that block words like viagra?

The antispam example in Joomla's own htaccess documentation blocks pharmaceutical terms in the query string. It is a copy-pasted mid-2000s spam blocklist that returns a 403 to legitimate medical, pharmacy and health sites while stopping no real attack. Nobody voted for the Joomla project to decide which words your visitors are allowed to search for.

Is the PHP Easter egg .htaccess rule still worth having?

No. PHP removed the Easter eggs in version 5.5, released in 2013, so no PHP version Joomla runs on even has them. If you ever needed to hide the PHP version, the php.ini

setting expose_php = Off did it at the server layer with no per-request cost, and it also removes the X-Powered-By header.

Why are user-agent and blocklist rules weak security?

Because the attacker controls the request. A blocklist can only ever list the bad strings you already know about. The moment an attacker uses a user-agent, query word or referrer you did not list, which costs them nothing, they walk straight through, while your blocklist keeps blocking real visitors who trip a stale entry.

Will Joomla's planned stricter default .htaccess fix the problem?

It reduces one narrow risk, a stray file executing in an upload folder, but it does not fix the vulnerable code behind the entry points that stay open, it only works on Apache, and the community's own discussion flagged that it can break legitimate tools such as Akeeba's backup and restore scripts. It is a useful thin layer, not a substitute for patching and secure-by-design extensions.

Does mySites.guru need my .htaccess relaxed to work?

The connector needs direct access to its own endpoint so it keeps working even when Joomla itself is down or compromised. That independence is the whole point: an external watcher that does not depend on the thing it is watching.

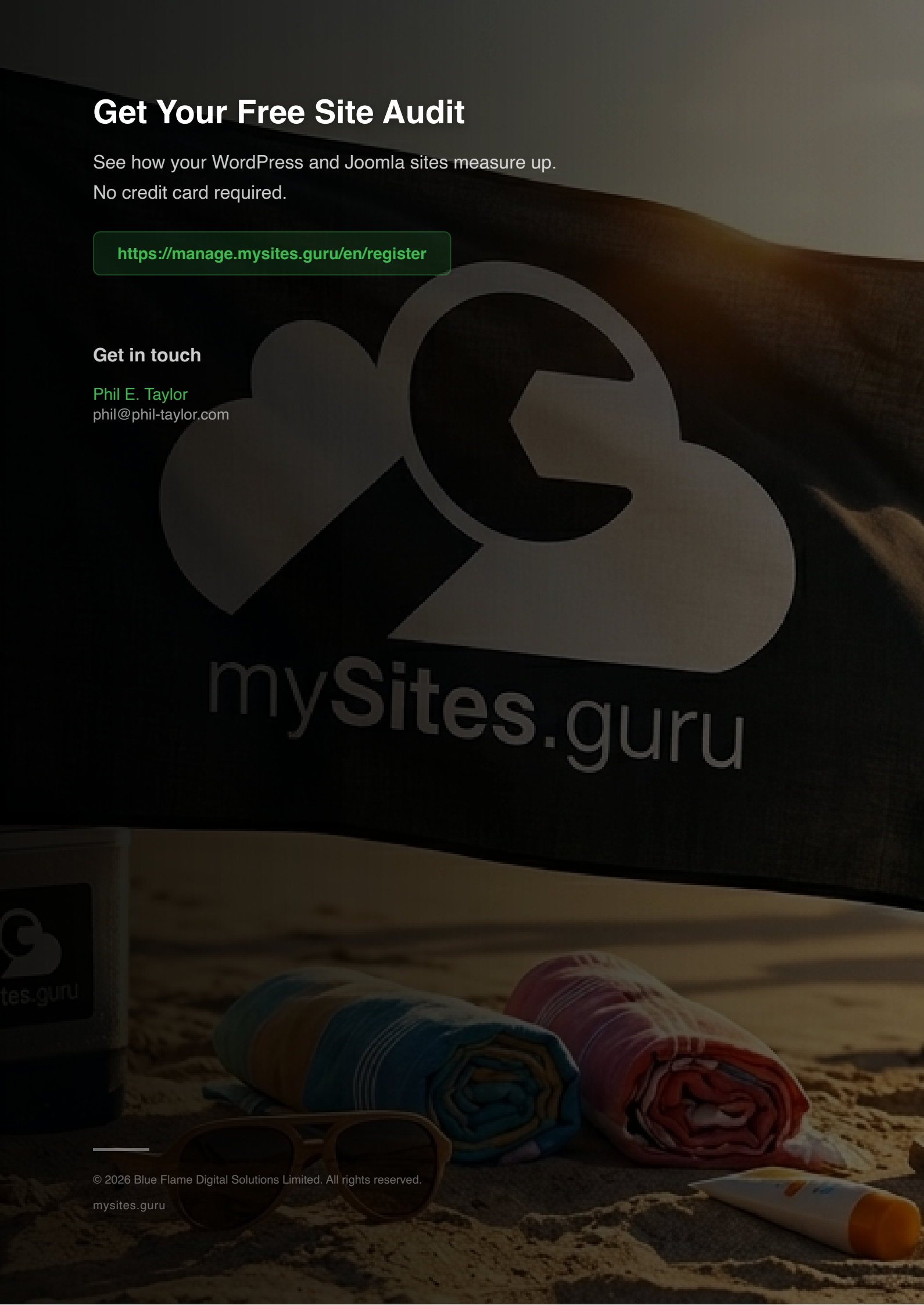
Written and edited by a human, with AI assistance. Our approach to AI

Get Your Free Site Audit

See how your WordPress and Joomla sites measure up.

No credit card required.

https://manage.mysites.guru/en/register



Get in touch

Phil E. Taylor
phil@phil-taylor.com

© 2026 Blue Flame Digital Solutions Limited. All rights reserved.