Incident Response | WordPress Security | Technical Case Study

Cybersecurity Incident Response & Recovery

Server-Wide WordPress Containment — Public Technical Case Study

A client-readable and engineer-ready account of how a self-healing, multi-layer WordPress compromise was investigated, contained, verified, and placed under recurrence monitoring without exposing client credentials, private infrastructure details, or customer data.

Linux | WordPress | PHP-FPM | Incident Response | Backup-First Quarantine | Verification | Monitoring

Public technical edition — Second-wave evidence verified 26 August 2026 — Published 26 August 2026

Latest update, 16 October 2026: a new infection found and contained

On 16 October 2026 a security alert reported new malware on the same hosting account. Server logs tie the first rogue account to a successful WordPress administrator login and a plugin upload two seconds earlier. Who used that login, and how they obtained it, is not proven. The same self-repairing malware family then spread across files and databases.

What was found:

- Five live malware runner processes.
- 247 generated rogue accounts across five affected databases.
- 274 casino spam posts.
- Oversized must-use plugin loaders and hidden token APIs that the earlier detector did not classify.

What was done, evidence first:

- Preserved process, file, database and log evidence before any change.
- Stopped web execution of the malware, terminated the five runners and quarantined 79 artifact files plus six token API files.
- Cleaned two themes, removed the rogue users and 53 persistence rows, and moved the 274 spam posts to the recoverable trash.
- Rotated seven database credentials, nine legitimate administrator passwords and seven key and salt sets, then revoked every affected session and application password.

- Extended the detector to recognise the new loader and API patterns, and deployed it with drift checks and hash parity.

Verified result on 16 October 2026:

- A full scan of 114 WordPress roots and 119 option tables found zero file artifacts and zero database artifacts.
- Zero malware runners and zero active sessions across all seven databases.
- Every old database password is rejected and every new one works; all nine administrator passwords verify.
- A real HTTPS login, dashboard, plugin installer and logout passed, along with 18 public and access checks and 21 regression tests.

Lesson: cleaning files is not full recovery while a valid administrator login or a plugin-install backdoor still works. This pass rotated every affected credential, not only the infected files. A separate fresh-context review of this pass is still outstanding.

---

1. Reading Guide and Publication Boundary

This document is a public-safe technical case study for developers, delivery teams, business owners, and prospective clients. It explains the incident-response reasoning, tooling, safeguards, evidence, and lessons from a real multi-site WordPress security event. It deliberately omits client names, credentials, private contact information, internal addresses, access commands, provider identifiers, private resource IDs, exact backup locations, and other details that could weaken the environment or identify an affected customer.

The case study distinguishes four kinds of statements:

- **Observed evidence** — a result directly supported by process inspection, file classification, database queries, service checks, or HTTP verification.
- **Historical state** — a condition that was true during investigation but was later superseded by containment or repair.
- **Current verified state** — the latest measured result at the end of the response window; it is not represented as a permanent guarantee.
- **Residual risk** — uncertainty that remains even after all known indicators are removed, especially when a server has a long compromise history.

The document describes a successful containment outcome. It does not claim that cleaning a historically compromised server is equivalent to rebuilding it from a trusted image.

2. Executive Summary

A hosting estate supporting many WordPress installations was found to contain a self-healing malware chain. Early checks identified suspicious plugins, injected content, hidden loaders, rogue administrative users, and memory-resident PHP processes. The first scope estimate was

too narrow because it relied on known filenames and on-disk inspection. Process-level observation later proved that the compromise crossed hosting-account boundaries and affected a much larger estate.

The response evolved from cautious read-only triage into an explicitly authorized, backup-first containment operation. Every destructive change was limited to structurally confirmed artifacts and preceded by a recoverable quarantine copy. The team expanded the classifier as new persistence layers were discovered, recycled only affected application workers where possible, and used controlled service reloads or restarts only after risk approval.

The final persistence source was identified through writer tracing and request-triggered reproduction. Once that source and its seed copies were removed atomically, repeated full-estate scans held at zero known findings and zero exact malware runners. Residual casino content in one database was repaired from a clean revision after a validated full database backup. Public HTTP checks then confirmed that 21 known sites returned successful responses without the checked malware or casino indicators.

An alert-only recurrence monitor was installed on a ten-minute schedule. It dynamically discovers WordPress roots, runs the expanded classifier, checks exact runner processes, validates essential services, and verifies the local automation listener. A clean-baseline email test was accepted by an off-box mail provider.

A second review on the following day overturned that closeout. While a healthy site was being validated as the template for rebuilding an unrelated one, a different implant family was found inside it — a plaintext one-line loader including a hidden twin whose decoder rebuilt every string from a private alphabet, together with archive seeds carrying a large counterfeit optimization plugin. The estate-wide sweep that followed found this family on 63 of 78 hosting accounts, alongside 21 live attacker administrator accounts and a dangerous default registration role on 74 databases. None of it appeared in any earlier quarantine set, and the ten-minute monitor had reported clean throughout.

The cause was not attacker sophistication so much as detector design. The variant contains no `eval`, no base64 decoding, no compression-based unpacking and no assertion-based execution, so every signature-oriented check scored zero — and that zero was read as evidence of absence. The scanner also walked only the locations the previous family had used, which did not include a loose randomized loader at the content-directory root. A replacement scanner then reproduced the same class of mistake a third time: it reported an entirely clean estate because an unsupported argument on the host's older Python raised an exception that a broad handler swallowed, so every search and query silently returned nothing.

Remediation proceeded in three authorized backup-first passes: attacker re-entry routes first, then staged artifacts, then attacker data. Every artifact was moved to a restricted quarantine rather than deleted, each file was re-hashed against the approved manifest immediately before

being moved, and every affected site's HTTP status was recorded before and after each pass. A structural detector was then written, deployed, and wired into the existing recurrence monitor, and it refuses to report a clean result it did not earn.

Measured closeout state — first wave, 25 August 2026:

- 94 dynamically discovered WordPress roots scanned.
- Zero artifacts identified by the expanded SC/keen structural classifier.
- Zero exact `.g_<hex>.php` malware runners observed.
- 21 known public sites returned HTTP 200 after containment.
- No checked casino, loader, injector, or known source-chain indicator appeared in homepage HTML.
- Required PHP, web, database, scheduler, and automation services were active.
- The recurrence monitor was installed, scheduled, clean-baselined, and email-delivery tested.

This state was accurate for what the classifier measured and inaccurate as a description of the estate. It is retained here because the gap between those two statements is the central lesson of this case study.

Measured closeout state — second wave, 26 August 2026:

- 310 implant artifacts identified across 63 of 78 hosting accounts, and all of them removed to quarantine.
- 215 archive seeds, 44 hidden loader twins, 9 visible loaders, 36 staged counterfeit plugins, and 6 injector staging kits accounted for individually.
- 21 attacker administrator accounts removed; none owned any published content.
- Dangerous default registration role reset on 74 databases; self-registration was disabled on every one of them, so the role was never externally reachable.
- A persistence-manifest option removed from 72 database tables after all 195 file paths it referenced were checked against disk.
- Every implant pattern class re-measured at zero by an independent scan.
- All 78 checked sites returned an unchanged HTTP status across all three passes.
- Required PHP, web, database, and automation services remained active throughout.
- Approximately 145 MB of validated database dumps and 21 MB of quarantined files retained for rollback.

3. Incident Context

The initial business symptom was not a dramatic server failure. Websites continued to respond, and the automation stack had previously appeared operational. The security concern emerged from suspicious files and casino-link behavior associated with several WordPress properties. That distinction matters: application availability is not evidence of platform integrity.

Read-only inspection found multiple generations of compromise artifacts. Some were visible and conventional, such as rogue plugins, database users, and injected content. Others were designed to survive ordinary cleanup:

- Hidden PHP loaders referenced through `auto_prepend_file` directives.
- Randomized filenames and backdated modification times.
- Marked payload blocks inserted into legitimate WordPress files.
- Wholly malicious drop-ins placed where WordPress loads code automatically.
- Memory-resident PHP runners whose corresponding files disappeared from disk.
- Source copies stored in cache locations and archive seeds.
- Malicious code retained in shared PHP worker state after files were removed.

The combination created a misleading operating picture. A file could be deleted successfully and still return seconds later. A site could return HTTP 200 while malicious code remained active. A filename-based scan could find nothing while a process continued running from an already deleted file descriptor.

4. Why the First Scope Was Wrong

The earliest scope count described only the sites carrying one known toolkit. That count was later repeated as though it described the entire compromise. A deeper review showed that an older persistence layer existed on additional sites and that process-level activity extended across separate hosting accounts.

Three technical assumptions caused the undercount:

4.1 Known filenames were treated as the whole threat

The first probes searched for artifacts already observed during an earlier incident. They were useful for proving infection but incapable of proving absence. A site without those filenames could still contain a different loader, a hidden MU-plugin, injected database content, or a memory-only runner.

4.2 Disk state was treated as runtime state

The malware launched PHP processes and then removed or hid the corresponding file. Ordinary file inventory could therefore report no matching path while the process remained alive under the init process. Process inspection was required to expose the discrepancy.

4.3 An HTTP response was treated as identity evidence

Some hosts returned success-like responses for arbitrary missing paths. Without a randomized negative control in the same directory, `200 OK` or an empty body could be misclassified as

proof that a malicious file existed. Later external probes paired every candidate path with a fresh non-existent control and reported unknown rather than clean when the control behavior was ambiguous.

4.4 A scanner's zero was treated as evidence of absence

This assumption survived the first wave and caused the larger miss. A classifier reports what it can recognize in the places it looks. Its zero therefore means "nothing matched here", never "nothing exists". Three distinct failures produced the same misleading zero: a detector that matched only signature strings the variant does not contain, a detector that walked only the directories the previous family had used, and a detector whose searches silently failed and returned empty results. In all three cases the reported number was identical to a genuine all-clear, which is precisely why it was believed.

The correction is to make a scanner accountable for its own coverage. A scan now fails loudly unless it can show it enumerated a plausible amount of work, and a scan that enumerated nothing is treated as broken rather than as clean. The correction was methodological: scope became evidence-driven and multi-layered. Files, processes, PHP worker behavior, database rows, HTTP controls, account boundaries, and request-triggered regeneration were evaluated together.

5. Threat Behavior and Persistence Model

The persistence chain operated as a system rather than as one malicious file. Its components had different responsibilities:

5.1 Runtime runners

Exact command patterns identified PHP processes executing randomized hidden files beneath WordPress core directories. The runners were parented by the init system, cycled over time, and could continue after their source path disappeared.

5.2 WordPress auto-load locations

Malicious MU-plugins and WordPress drop-ins executed automatically during normal requests. Some payloads were wholly malicious; others inserted clearly delimited blocks into otherwise legitimate files. Removing only the visible plugin did not address all auto-loaded copies.

5.3 Prepend chain

Configuration directives loaded randomized PHP files before ordinary application code. The chain included a visible loader, a hidden twin, and in some cases an obfuscated cache copy capable of restoring the active layer.

5.4 Shared worker memory

Affected PHP-FPM workers retained executable state after on-disk cleanup. New requests could cause known artifacts to reappear even when no exact runner was visible. Account-scoped worker recycling reduced the risk to unrelated sites, but some shared pools required explicitly approved reloads or a full restart.

5.5 Source and seed artifacts

The final resistant account contained a fake generator-style MU-plugin, an obfuscated cache payload, and multiple identical archive seeds. The archives were inspected without extracting or executing their PHP content. Removing the seven visible symptoms without the source and seeds could not hold.

5.6 Database-layer abuse

5.7 Signature-invisible obfuscation

The second-wave family carries none of the markers that conventional scanning looks for. It contains no evaluation call, no base64 decoding, no compression-based unpacking, and no assertion-based execution. Instead it stores a table of short tokens and rebuilds every identifier and string at runtime by remapping characters through a private alphabet, assembling the result with ordinary string concatenation. The behavior is entirely conventional PHP; only the shape is unusual.

That shape is what makes it detectable. A file combining a large table of short quoted tokens, heavy single-character concatenation, and a character-remapping loop is structurally distinctive even though none of its individual elements is suspicious. Detection moved from "what strings does this file contain" to "what does this file look like".

5.8 A staging layer that is not yet running

Most of the second-wave footprint was inert. Archive seeds were planted in three locations per site — the content root, a theme directory, and a media upload path — with modification times backdated by years, which is self-evidently false where the containing directory is dated to the current year. Counterfeit plugins sat unpacked in plugin directories without appearing in any site's active-plugin list, and randomized loaders sat beside their hidden twins with nothing referencing them.

Inert is not harmless. A staging layer is a complete reinstallation kit that requires only one writable moment to become active, and it is invisible to any check that looks for running malicious code. Scope must be measured by what is present, not by what is currently executing.

5.9 The implant's own reinstall manifest

The compromise stored a persistence manifest in the database: a serialized mapping of file path to size and timestamp, recording every location the implant restores itself into. It named automatically loaded plugin files, a database drop-in, and theme function files — several of which no pattern-based sweep had been looking at.

Read as evidence rather than as a threat, this was the single most useful artifact in the entire investigation. It is the attacker's own inventory of the persistence surface, and comparing it against disk is a direct test of whether cleanup actually finished. The file compromise was accompanied by rogue administrators, a dangerous default registration role, malicious plugin activation, attacker options, casino posts and revisions, and poisoned page-builder data. A clean file scan alone could not repair or validate this layer.

6. Response Principles

The containment work followed a small set of non-negotiable rules.

6.1 Back up before mutation

Every apply pass wrote the original bytes and a manifest into a new restricted quarantine checkpoint before removing or rewriting the live artifact. The database repair began only after a full compressed dump was created and validated as readable.

6.2 Match structure, not broad suspicion

The scrubber classified files using confirmed markers, decoder structure, path relationships, archive contents, and exact directive patterns. It did not delete files merely because they were recently modified, minified, encoded, unfamiliar, or located in WordPress.

6.3 Never execute a suspected payload

The classifier read bytes, parsed archives without extraction, and syntax-checked cleaned PHP using a separate interpreter invocation. It never included or evaluated target PHP.

6.4 Make the smallest operational change

Where possible, only workers owned by affected accounts were recycled. Shared PHP services were reloaded or restarted only when isolated evidence showed that shared runtime state was replaying the malware and the broader interruption risk had been approved.

6.5 Verify after real requests

A zero-result scan before application traffic was insufficient. The response exercised public sites, reproduced request-triggered regeneration during diagnosis, and repeated scans after normal page requests and delays.

6.6 Separate containment from restored trust

The team reported successful removal of all known indicators without claiming that a historically compromised server could never contain an unknown implant. This distinction protects clients from both alarmism and false assurance.

7. Investigation Workflow

The investigation progressed through controlled stages:

1. Confirm service availability and preserve an operational baseline.
2. Inventory known WordPress roots, suspicious files, database indicators, and exact runner processes.
3. Pair external presence probes with randomized missing-file controls.
4. Capture a live runner safely through its open file descriptor for static analysis.
5. Map the runner's restoration behavior to MU-plugins, drop-ins, theme blocks, prepend directives, and loader files.
6. Build a dry-run classifier and review action counts before applying changes.
7. Quarantine confirmed artifacts and recycle only affected workers.
8. Observe regeneration, expand classification only when a new layer is positively identified, and repeat with a fresh backup checkpoint.
9. Trace the final writer using a temporary self-removing audit rule.
10. Trigger a controlled normal request to reproduce the write and correlate it to the correct application worker.
11. Remove the complete final source chain atomically while the affected worker state is controlled.
12. Run immediate, request-triggered, and delayed full-estate verification.
13. Repair database content separately with its own backup and acceptance criteria.
14. Install a recurrence monitor and prove off-box email acceptance.

This staged approach made unexpected behavior informative. Regeneration was not treated as a reason to delete more broadly; it was used to narrow the unresolved source.

8. Backup-First Containment Engineering

The scrubber operated in dry-run mode by default. A dry run discovered WordPress roots, built a candidate set, classified each candidate, summarized actions, and identified affected hosting users without changing production. Apply mode then created an isolated quarantine directory and copied each confirmed original before mutation.

For marked injections, the tool removed only the delimited malicious block. If the remaining file was empty or only an inert PHP wrapper, the file could be removed entirely. Rewritten PHP was syntax-checked before atomic replacement. If live bytes changed after planning, the item was reclassified against the current content; it was skipped if it no longer matched.

Archive seeds received additional safety treatment:

- Only the confirmed randomized archive naming form was considered.
- Archive size was capped before inspection.
- The archive had to contain the expected single PHP member.
- The member had to match the known decoder structure.
- Content was read in memory and never extracted into a web-accessible location.

This design made each apply pass recoverable, narrow, and auditable.

9. Convergence and the Final Source

The first server-wide cleanup dramatically reduced the process and file count but did not converge. A small group of accounts regenerated known artifacts. Additional analysis discovered hidden loader twins and obfuscated cache loaders that were not part of the original classifier.

After those layers were quarantined, the estate reached zero everywhere except one account using a different PHP-FPM version. A graceful reload did not clear the shared state. Even a full restart followed by symptom-only cleanup failed, proving that an on-request source still existed.

A temporary audit rule captured the actual writer: an ordinary PHP request worker for the remaining site. Controlled homepage requests reproduced the write. Static tracing then found the fake MU-plugin source, its cache payload, and three identical archive seeds.

The response added structural recognition for those items, quarantined the full 12-artifact chain in one pass, restarted the affected shared PHP runtime while the source was absent, and exercised the real homepage. The account remained at zero. Two later full-estate scans and delayed request testing also remained at zero.

The key lesson is that convergence matters more than deletion count. A cleanup is incomplete until the environment remains clean after the application performs the actions that previously triggered reinfection.

10. Database and Content Repair

Once the process and file layer held at zero, public content was rescanned. Residual casino material existed in one WordPress database even though the reinjector was gone.

The repair sequence was intentionally separate from file containment:

1. Identify the affected homepage and locate its latest clean revision.
2. Create and validate a complete compressed database backup.
3. Restore both normal page content and page-builder metadata from the clean revision.
4. Remove the current generated page cache so clean content could render.
5. Delete attacker-owned casino posts, revisions, related metadata, and term relationships using dependency-aware queries.
6. Remove rogue administrators and their metadata.
7. Remove malicious options and plugin activation state.
8. Restore the safe default user role while preserving closed registration.
9. Validate serialized plugin state and remove stale update metadata.
10. Recount database indicators and inspect the rendered public page.

The repair removed more than two thousand attacker-controlled post and revision records. Final queries found no remaining checked casino content, rogue users, malicious options, or unwanted plugin activation. The repaired site and the wider public set continued to return successful responses.

11. Verification Model

The acceptance model required independent evidence at several layers.

11.1 Static artifact verification

The expanded classifier scanned all 94 discovered WordPress roots and returned zero identified artifacts with no affected users.

11.2 Runtime verification

The exact malware runner process pattern returned zero. Required PHP-FPM versions, the web server, database, scheduler, and automation service were active.

11.3 Request-triggered verification

Normal public requests were sent to the previously affected estate, including the site that had reproduced the final reinfection. Scans were repeated after those requests.

11.4 Delayed verification

Clean results were repeated after a delay rather than accepted immediately after service recycling. This allowed cached workers and scheduled behavior an opportunity to reveal persistence.

11.5 Public content verification

Twenty-one known sites returned HTTP 200. Homepage HTML was checked for known casino strings, loader infrastructure, injector names, and final-source indicators.

11.6 Database verification

Targeted queries confirmed zero checked casino posts, zero known malicious post metadata, zero malicious options, zero rogue administrators, and safe role/plugin configuration on the repaired database.

11.7 Scanner self-accountability

A verification layer was added for the tooling itself, because three separate detectors had reported clean results they had not earned. Any scan is now required to prove that it enumerated a plausible amount of work before its findings are believed: a minimum number of discovered application roots and a minimum number of inspected configuration tables. Below either threshold the scan exits as incomplete rather than as clean, and the monitor raises an alert.

This check has already proved itself. The first deployment of the new detector failed immediately, because the host runs an older Python than the script assumed. The monitor reported an alert state rather than a clean one, which is the correct and intended outcome of a broken scan. No single check was treated as sufficient. Confidence came from agreement between file, process, database, service, and public-request evidence.

12. Recurrence Monitoring

The installed monitor is designed to detect recurrence, not to perform automatic remediation. Automatic deletion during an alert could destroy evidence, remove a false positive, or create a production outage.

Every ten minutes the monitor:

- Acquires a lock so overlapping scans cannot run concurrently.
- Dynamically discovers current WordPress roots rather than relying on a fixed site list.
- Runs the expanded structural classifier in dry-run mode with a bounded timeout.
- Treats an unparseable scan, failed scan, or zero discovered roots as an alert condition.
- Counts exact memory-runner command patterns.
- Checks required PHP, web, database, and scheduler services.
- Confirms the automation engine remains bound to its intended local listener.
- Stores clean or alert state with restricted permissions.
- Sends immediately on transition to alert and periodically reminds while the condition persists.

The monitor was subsequently extended with a second, structural gate that detects by code shape and path relationship rather than by signature. It examines loader and twin naming patterns, archive members without extracting them, injector staging directories, obfuscated automatically-loaded code and drop-ins, and the database layer — persistence-manifest options, dangerous default roles, malicious plugin activation, and administrator accounts matching attacker-generated naming. It reports a single artifact total that the monitor treats as an alert condition whenever it is non-zero or unparseable.

The gate is read-only and never remediates. It runs as a privileged process, so it invokes no shell at all, and every database and table identifier is validated against a strict allowlist before it reaches a query.

- Sends a recovery notice when a prior alert returns to clean.

The acceptance test performed a fresh full-estate scan with zero findings and zero runners. The off-box mail provider accepted the health message successfully, and no recipient entry remained deferred in the local mail queue.

The honest qualification is that this original acceptance test passed while the estate was in fact widely infected, because the classifier behind it could not see the second-wave family. The structural gate closes that specific gap; it does not prove that no third family exists. A monitor's clean state is only ever as broad as the detection behind it, and that boundary should be stated whenever the state is reported.

This on-box monitor closes the known recurrence-detection gap but not the total-outage gap. If the entire server, network, or mail path fails, the monitor may be unable to report. A separately hosted public heartbeat and an independent business-outcome alarm remain recommended.

13. Service Continuity and Change Control

Incident response took place on a shared production host. The operational objective was to remove the threat without converting a security incident into a broad outage.

Controls included:

- Dry-run classification and explicit counts before every expanded cleanup scope.
- Separate approval for server-wide artifact mutation.
- Separate approval for shared PHP reloads and the more disruptive full restart.
- Account-scoped worker recycling before shared-service action.
- Health checks for all PHP versions, web service, database, scheduler, and automation listener after each sensitive phase.
- Public HTTP checks after containment.
- Retained quarantine manifests and database backups for rollback and forensic reference.

The security monitor remains alert-only. The firewall was already operating in enforcement mode with a deny-by-default inbound policy and active login-failure defense, so no speculative firewall mutation was introduced merely to claim additional hardening.

14. What Did Not Work — and Why

14.1 Partial deletion

Removing visible files without the complete chain caused rapid regeneration. The environment required atomic source-chain containment rather than repeated symptom removal.

14.2 Known-name scanning alone

Known names found real malware but missed randomized and structurally equivalent copies. The improved classifier combined path relationships, content markers, decoder behavior, and archive structure.

14.3 Disk-only inspection

Self-deleting runners proved that absence from the filesystem did not mean absence from memory. Process inspection and open-file-descriptor recovery were essential.

14.4 Graceful reload as a universal cache reset

One PHP-FPM reload preserved enough shared runtime state for reinfection to continue. Service behavior was inspected rather than assumed, and a full restart was used only after approval.

14.5 Cleanup before proving convergence

An immediate zero result could be temporary. Real requests and delayed rescans were required before declaring the known process/file layer clean.

14.7 Signature matching against signature-free code

Every grep-oriented check scored zero on the second-wave family because the family contains none of the strings those checks look for. Signature detection is useful for confirming a known threat and structurally incapable of proving absence. Anything that reconstructs its own identifiers at runtime defeats it completely.

14.8 Scanner scope mistaken for estate scope

The classifier walked the specific locations the first family used. A randomized loader placed at the content-directory root fell outside that walk, and because the hidden twin was only ever derived from an already-classified loader, neither was reachable. The scanner was not wrong about what it examined; it was silently narrower than the thing it was trusted to measure.

14.9 Broad exception handling in security tooling

A replacement scanner reported a completely clean estate because an unsupported argument on the host's older Python raised an exception inside a helper wrapped in a catch-all handler that returned an empty result. Every search and query returned nothing, and nothing is indistinguishable from clean. In security tooling a swallowed error is not a robustness feature — it is a false negative with no warning attached. Errors must surface, and an empty result set must be justified by evidence of work performed.

14.6 Treating containment as trust restoration

Even a comprehensive zero-result scan covers only known and detectable behavior. A long-running cross-account compromise creates residual uncertainty that only a clean rebuild, scrubbed migration, and complete secret rotation can meaningfully reduce.

15. Developer Playbook

15.1 Before touching production

- Establish an authoritative asset inventory.
- Record service health, public availability, and current process state.
- Separate confirmed indicators from hypotheses.
- Define rollback for every mutation class.
- Obtain explicit approval when scope expands across accounts or shared services.

15.2 During containment

- Default every scanner and scrubber to dry-run.
- Back up original bytes before modification.
- Use structural confirmation rather than broad heuristics for deletion.
- Never execute suspicious PHP to learn what it does.
- Prefer atomic replacement and syntax validation for partially cleaned files.
- Reclassify files that change between planning and apply.
- Recycle the narrowest possible worker scope.
- Treat regeneration as evidence pointing to an unresolved source.

15.3 Before declaring success

- Scan the full discovered estate, not only the originally reported sites.
- Count exact runner processes.
- Exercise real application requests.
- Repeat the scan after a delay.
- Validate database content and administrative state separately.
- Check every required service and public site.
- Install alerting and prove actual off-box acceptance.
- Record both the verified result and the residual-risk boundary.

16. Client-Facing Guidance

Clients do not need to interpret every malware artifact, but they should receive clear answers to five questions:

1. **What was affected?** Explain the measured scope and how it changed during investigation.
2. **What was done?** Describe backups, containment, database repair, service handling, and monitoring without exposing access details.

3. **How was it verified?** Provide file, process, request, service, and content evidence rather than a single scanner screenshot.
4. **What remains uncertain?** Distinguish known-clean verification from absolute assurance.
5. **What should happen next?** Retain backups, rotate exposed secrets, maintain monitoring, and plan a clean rebuild when compromise depth warrants it.

Good security reporting is neither vague reassurance nor dramatic speculation. It is a dated statement of evidence, actions, limitations, and ownership.

17. Residual Risk and Rebuild Recommendation

The containment outcome is strong: the expanded known-indicator classifier held at zero across 94 roots, exact runners remained absent after normal requests, database indicators were removed, public sites responded successfully, and monitoring was proven.

The residual risk is also real. The environment showed long-lived persistence, cross-account activity, memory-resident execution, hidden loaders, shared-runtime replay, and database abuse. Those facts mean the host cannot regain the same assurance as a freshly provisioned system simply because known indicators are gone.

A high-assurance rebuild should:

- Start from a newly imaged server rather than an account transfer or disk clone.
- Install fresh operating-system, PHP, database, WordPress core, plugin, and theme code from trusted sources.
- Migrate scrubbed database content and media only; do not copy old executable PHP.
- Exclude executable files from uploads and block PHP execution there.
- Recreate automation services and credentials rather than copying an encrypted runtime store from the old host.
- Rotate server, database, CMS, messaging, workspace, encryption, and application secrets.
- Validate each site on a temporary hostname before DNS cutover.
- Reinstall on-box integrity checks and add an independently hosted external heartbeat.
- Retire and wipe the historical host after successful migration and evidence retention.

Containment protects current operations. Rebuilding restores trust.

18. Outcome Summary

The response succeeded because it changed its model when evidence contradicted the first explanation. A narrow plugin incident became a server-wide process investigation. A file-cleanup problem became a convergence problem. A persistent seven-file symptom became

a request-triggered source-tracing exercise. A clean file state became a multi-layer acceptance test.

The resulting controls are reusable beyond this incident:

- Dynamic asset discovery instead of hand-maintained scope.
- Randomized negative controls for external file probes.
- Process-plus-file visibility for self-deleting malware.
- Backup-first, structure-specific quarantine.
- Explicit approval gates for shared production services.
- Request-triggered and delayed verification.
- Separate database/content acceptance.
- Alert-only recurrence monitoring with tested off-box transport.
- Honest residual-risk reporting and a clean-rebuild path.
- Structural detection by code shape, for malware that carries no signature strings.
- Scanner self-accountability, so an unearned zero fails loudly instead of reassuring.
- Treatment of a staged but inactive implant as in-scope, not as a lesser finding.
- Use of the attacker's own persistence manifest as a coverage test for cleanup.
- Rebuilding rather than cloning, whenever the intended source cannot be proven clean.

This is the difference between deleting suspicious files and performing defensible incident response.

19. Second Wave — What an Unearned Zero Costs

The most instructive part of this incident is not the malware. It is that a competent, evidence-driven response produced a confident, well-documented, fully verified closeout that was wrong — and that the verification method, not the attacker, is what made it wrong.

19.1 How it surfaced

It surfaced by accident. A healthy site was being examined as the template for rebuilding an unrelated one, and the integrity check performed before trusting it as a source found artifacts inside it. Had that rebuild not been requested, the recurrence monitor would have continued reporting a clean estate indefinitely.

This is worth stating plainly for clients and engineers alike: the discovery was a byproduct of refusing to trust a source without checking it. Routine paranoia about inputs found what dedicated monitoring did not.

19.2 What the second sweep measured

The structural sweep found 310 artifacts across 63 of 78 hosting accounts: archive seeds planted in three locations per site, hidden loader twins, plaintext loaders, counterfeit optimization plugins staged in plugin directories, and complete injector kits held in randomized staging directories. The database layer carried 21 live attacker administrator accounts, a dangerous default registration role on 74 databases, a persistence manifest on 72 tables, and residual injector configuration.

Two findings materially reduced the severity. The counterfeit plugin appeared in no site's active-plugin list, so it was staged rather than running. And although the default registration role was set to administrator across most of the estate, self-registration was disabled on every one of those databases, so the role was never externally reachable. Both facts were verified rather than assumed, and both were reported alongside the alarming totals rather than after them.

19.3 What was actually dangerous

The attacker administrator accounts. Fifteen of them were created within a twenty-minute window on the day of the first cleanup, roughly an hour before that cleanup took its first backup — which means they were already present while the response was running, and were never in its scope because the response was looking at files.

Those accounts are the reason a file-only cleanup could not have held. Removing malicious code from a system that an attacker can still log into as an administrator is not remediation; it is maintenance performed on the attacker's behalf. Re-entry routes were therefore remediated first, before a single implant file was touched.

19.4 How the remediation was sequenced

Three passes, each separately authorized, each backed up before mutation, each verified afterward:

1. **Re-entry routes.** Attacker administrator accounts and their metadata removed, default registration roles reset, and automatically loaded counterfeit plugins quarantined. Database tables dumped and integrity-checked before any row changed.
2. **Staged artifacts.** Every implant file moved to quarantine, with each file re-hashed against the approved manifest immediately before the move so that anything altered between planning and execution would be skipped and reported rather than moved blind.
3. **Attacker data.** Persistence manifests, injector configuration, malicious plugin activation entries, and staging kits removed. Serialized plugin lists were rewritten by the language runtime's own serializer rather than by editing the serialized text, and validated again afterward.

Every affected site's HTTP status was captured before and after each pass. None changed.

19.5 The detection fix

A structural detector replaced the signature-based one and was wired into the existing ten-minute monitor. It recognizes decoder shape, loader and twin path relationships, archive membership, injector staging directories, and the database persistence layer.

Critically, it is accountable for its own coverage. If it discovers implausibly few application roots or configuration tables, it exits as incomplete and the monitor alerts, rather than reporting the zero findings that an empty scan would otherwise produce. That single behavior is the direct countermeasure to the failure that produced the original false all-clear.

19.6 The transferable lesson

A security scanner that has never been tested against a known positive it fails to detect is not evidence of a clean system. It is only evidence that it found nothing, and those are very different claims.

Three practices follow from this, and they generalize well beyond WordPress:

- **Test detectors against what they miss, not only against what they catch.** A detector's value is defined by its blind spots, and blind spots are only discoverable deliberately.
- **Make tools prove their coverage.** Any report of zero findings should be accompanied by evidence of how much was actually examined, and should fail when that evidence is missing.
- **Report the boundary with the result.** "Clean" is always shorthand for "clean by this method, on this date, across this scope". Stating the method and the scope alongside the verdict is what makes the verdict trustworthy, and what allows it to be revisited honestly when the method improves.

Appendix A — Detection-to-Recovery Flow

Report received → preserve evidence → verify known indicators → expand asset scope → inspect processes → recover deleted runner for static analysis → model persistence chain → build dry-run classifier → quarantine confirmed artifacts → recycle narrow worker scope → observe regeneration → discover hidden/cache layer → repeat backup-first containment → isolate final account → trace request-time writer → identify source and seeds → atomic source-chain removal → controlled runtime restart → real-request test → full-estate scan → delayed rescan → database repair → public-site validation → install monitor → prove email delivery

Appendix B — Minimum Acceptance Checklist

- ☐ Full asset discovery completes with a non-zero, plausible root count.
- ☐ Expanded classifier reports zero known artifacts.
- ☐ Exact malware runner count is zero.
- ☐ Results remain zero after normal application requests.
- ☐ Results remain zero after a defined delay.
- ☐ Required PHP, web, database, scheduler, and automation services are healthy.
- ☐ Public sites return expected status and no checked content indicators.
- ☐ Database indicators, rogue users, unsafe roles, and malicious activation state are zero.
- ☐ Every mutation has a recoverable quarantine or validated database backup.
- ☐ Monitoring runs on schedule and cannot overlap itself.
- ☐ Alert transport is proven off-box, not merely accepted into a local queue.
- ☐ Residual risk and rebuild guidance are recorded for stakeholders.

Appendix C — Glossary

- **Artifact** — a file, injected block, configuration directive, archive, database row, or process associated with the confirmed threat behavior.
- **Containment** — stopping known malicious behavior and preventing immediate recurrence while preserving recoverability and evidence.
- **Convergence** — the point at which repeated cleanup and verification remain at zero after the workloads that previously caused regeneration.
- **Drop-in** — a specially named WordPress file loaded automatically for functions such as database or cache behavior.
- **MU-plugin** — a must-use WordPress plugin loaded automatically without normal administrator activation.
- **Opcode cache** — compiled PHP code retained by the runtime for performance; during this incident it could replay malicious behavior after disk cleanup.
- **Quarantine manifest** — a record connecting each changed production path to its backed-up original and classified action.
- **Request-triggered persistence** — malicious regeneration caused by serving an ordinary application request.
- **Residual risk** — uncertainty remaining after known indicators are removed and acceptance checks pass.
- **Structural classifier** — detection based on code form, markers, relationships, and behavior rather than filenames alone.

Appendix D — Public Documentation Safety Checklist

- No passwords, access tokens, API keys, private keys, cookies, or authentication material.
- No private phone numbers, personal email addresses, or customer records.
- No server addresses, host fingerprints, login commands, or internal file paths.
- No provider account identifiers, private document IDs, workflow IDs, or backup locations.
- No copied malicious payload source capable of execution.
- Technical indicators are generalized to the minimum useful detection pattern.
- Counts and outcomes are dated evidence, not permanent guarantees.
- Rebuild and secret-rotation guidance remains explicit where trust cannot be restored in place.

Developed by Zahidul Islam

Cybersecurity incident response, WordPress containment, verification engineering, and operational resilience.