From Payload to Plugin: Web-Scale Ecosystem Attribution of JavaScript Injection Campaigns

Ravindu De Silva
ldesilva@ucsb.edu
University of California Santa Barbara, Santa Barbara, USA

Nicholas Shao
nicholasshao@ucsb.edu
University of California Santa Barbara, Santa Barbara, USA

Yigitcan Kaya
yigitcan@ucsb.edu
University of California Santa Barbara, Santa Barbara, USA

Mingxuan Yao
mingxuanyao@gatech.edu
Georgia Institute of Technology, Atlanta, USA

Yanick Fratantonio
yanickf@google.com
Google, Zurich, Switzerland

Luca Invernizzi
invernizzi@google.com
Google, Zurich , Switzerland

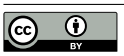
Christopher Kruegel
chris@ucsb.edu
University of California Santa Barbara, Santa Barbara, USA

Giovanni Vigna
vigna@ucsb.edu
University of California Santa Barbara, Santa Barbara, USA

Abstract

External security analysts monitoring content management system (CMS) websites for compromise often lack backend access. From the frontend alone, they can detect malicious JavaScript injections and fingerprint installed plugins, but cannot link the two to identify the *root cause* of compromise—the vulnerable plugin that enabled the breach. As a result, site maintainers often remove visible payloads while leaving the underlying vulnerability unpatched, creating a risk of rapid reinfection. Prior work has studied malicious JavaScript detection and CMS ecosystem security in isolation, without a systematic way to bridge this gap at scale.

We present a large-scale measurement study that connects frontend-observable injections to their likely root causes in the WordPress ecosystem by comparing plugin prevalence across victim and non-victim website populations. Over 84 days, we monitor 850,000 WordPress domains, use cross-snapshot JavaScript diffs to identify candidate injections, cluster behavioral traces into campaigns, and apply *enrichment* analysis[1] to recover likely exploited components at the plugin:version level. This process identifies 20 campaigns spanning 13 behavioral families and affecting 1,160 sites, of which 18 campaigns remain entirely undetected by VirusTotal during the study period. Guided by the components identified through our analysis, we uncover three previously unreported vulnerabilities in popular WordPress plugins, develop proof-of-vulnerability exploits, and report them to Wordfence under coordinated disclosure. We additionally release all extracted Indicators of Compromise via AlienVault OTX to support further research and defense.



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846686

CCS Concepts

• **Security and privacy → Web application security**; *Vulnerability management*; *Malware and its mitigation.*

Keywords

JavaScript injection, WordPress, behavioral clustering, ecosystem attribution, enrichment analysis

ACM Reference Format:
Ravindu De Silva, Nicholas Shao, Yigitcan Kaya, Mingxuan Yao, Yanick Fratantonio, Luca Invernizzi, Christopher Kruegel, and Giovanni Vigna. 2026. From Payload to Plugin: Web-Scale Ecosystem Attribution of JavaScript Injection Campaigns. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846686

1 Introduction

Injected malicious JavaScript remains a practical post-compromise mechanisms for abusing legitimate websites. Once a site is compromised, adversaries leverage their access to insert loader snippets, external script dependencies, or obfuscated inline JavaScript [9, 20] into pages that are served to unsuspecting visitors. These injected scripts enable traffic redirection, digital skimming, and credential theft. To gain access, attackers exploit vulnerabilities such as Stored Cross-Site Scripting (XSS), file inclusion, or SQL injection.

A frequent target for attackers are Content Management Systems (CMSes) and their third-party plugins and themes [8]. A single vulnerability in a popular plugin allows the same malicious injection to be replayed across every site running it. Among CMSes, the WordPress platform is a particularly popular victim, as it powers over 40% of the web [42] and depends on a large ecosystem of third-party plugins and themes distributed across multiple marketplaces [17]. These components are often plagued by exploitable vulnerabilities due to inadequate vetting and poor security practices [17].

Security analysts who monitor the web for compromised sites typically operate without backend access to the domains they observe [14, 24, 28, 39]. Just looking at web pages and their content,

one can detect injected JavaScript, yet one cannot determine which plugin's vulnerability was actually exploited to inject the payload (if, indeed, the attackers obtained access through such a vulnerability; alternative infection vectors are, of course, possible). This makes it more difficult to determine (and remediate) the root cause of infections. In turn, even if a specific injection or malicious script is cleaned up, the site remains vulnerable to reinfection [18].

Prior research has advanced related lines of defense: First, malicious JavaScript detection, which has evolved from syntactic classifiers [3, 7, 31] to cloaking-aware and deobfuscation-oriented systems [9, 20] to runtime behavioral signatures derived from browser API call sequences [16, 34]. These systems identify infected pages, but do not link the compromise to any specific ecosystem component. Second, campaign attribution tools such as JSgraph [22] track which scripts modify a page and what they do, but not which plugin or theme enabled their injection. Third, CMS ecosystem studies such as YODA [17] and TARDIS [18] characterize vulnerable Wordpress plugins and demonstrate that root-cause identification prevents reinfection. However, these studies rely on backend or marketplace-level access that is unavailable to an external observer. No prior work connects these threads end-to-end: from detecting a JavaScript injection campaign, to attributing it to specific vulnerable plugins or themes that may have enabled the breach. Without this link, site owners may remove the visible payload while leaving the entry point intact, leading to rapid reinfection [18].

In this paper, we ask a simple question to bridge this gap: what vectors do attackers use to compromise WordPress sites and inject malicious JavaScript? Answering it without backend access requires moving beyond detection (*"this page is infected"*) toward actionable ecosystem intelligence (*"this campaign co-occurs with Plugin X at Version Y"*). Our central intuition is that a vulnerable component behind an injection campaign is rarely identifiable from a single compromised site alone. We therefore frame attribution as a population-level problem: if a certain JavaScript injection campaign appears across many sites, the responsible component should be overrepresented among victims relative to its baseline prevalence in the broader WordPress ecosystem. Although such statistical evidence cannot prove causality, it can highlight which components are most likely to serve as attackers' entry points.

Operationalizing our intuition requires tackling several challenges. First, we must identify compromised sites (and malicious JavaScript) at scale, despite payloads that have evolved for years to evade detection. Second, we must identify which malicious scripts belong to the same campaign and are therefore more likely to share an underlying entry point. This is difficult because modern JavaScript campaigns routinely rewrite payloads for each victim site, for example by re-encrypting the same malicious payload with a per-site encryption key or regenerating variable names so that each infected page produces a unique content hash [20, 34]. As a result, grouping scripts by code similarity or payload hashes is often unreliable. Third, even after identifying a campaign's victim sites, each site typically runs dozens of plugins, making it difficult to infer the likely entry point without backend access.

We address these challenges with an external-observer analysis pipeline. Starting from a large corpus of WordPress domains, we periodically crawl each site and compare consecutive crawls. For each site, we diff its JavaScript to isolate newly added or modified scripts. When a JavaScript diff cannot be attributed to a benign source, e.g., a plugin version update, we load the page in an instrumented browser and record the runtime execution trace of each suspicious script. We then cluster these traces by behavioral similarity to identify candidate campaigns, each associated with a set of potential victim websites. Because clustering relies on behavior rather than code content, this approach is robust to the per-site payload polymorphism and randomized obfuscation that campaigns commonly use to evade static detection signatures [20, 34].

After clustering, we prioritize the most suspicious clusters for deeper review using a heuristic scoring stage, reducing downstream workload by more than 95%. For each confirmed campaign, we fingerprint the victim sites to identify the plugins and themes installed on them. We then perform *enrichment analysis*[1], comparing each component's prevalence among victim sites against its prevalence in the full crawled population. Components that are statistically overrepresented among victims are flagged as potential entry points. When an overrepresented component has published CVEs, the association provides strong evidence of a likely entry point. When no matching public CVE exists, the component becomes a zero-day candidate warranting source code audit.

We deployed our pipeline and ran it on about 850,000 WordPress domains over 84 days (from December 28, 2025 through March 22, 2026). During this time, we identified 20 campaigns spanning 13 behavioral families (Section 4.2) across 1,160 compromised sites. For each campaign, we match the enriched plugin and theme components against published CVE records at the exact version level, recovering at least one known vulnerability per campaign across all 20 campaigns. We further discover three previously undisclosed vulnerabilities in popular plugins for which we developed working proof-of-vulnerability exploits, (CVE-2026-6287 [43]). We additionally find that 18 of the 20 campaigns have zero detections across all VirusTotal engines for every payload sample observed throughout the study period, and release all extracted Indicators of Compromise via AlienVault OTX [29] to support broader defensive efforts.

In this paper, we make the following contributions.

(1) **A hybrid detection pipeline** that monitors JavaScript deltas at web scale and groups injection campaigns through static code analysis, JavaScript execution tracing, and clustering, capturing polymorphic campaigns that evade simple content-based similarity identification. Heuristic-based cluster prioritization reduces the downstream workload by over 95% (Section 3).
(2) **An ecosystem attribution methodology** that links campaigns to the plugins, themes, and versions observed on victim sites, recovers matching CVEs, and applies our enrichment analysis to separate genuine entry-vector candidates from background noise (Sections 3.5 and 4.5).
(3) **An 84-day longitudinal measurement** (December 28, 2025 to March 22, 2026) across 850,000 WordPress domains identifying 20 campaigns spanning 13 behavioral families. The measurement recovers known CVE correlations for each campaign, uncovers attacker-created fake plugins used for post-compromise persistence, and discovers three previously undisclosed vulnerabilities in popular plugins for which we developed working

[1]Inspired by gene-set enrichment analysis [38], which identifies meaningful associations by testing over-representation relative to a background distribution.

proof-of-vulnerability exploits, reported to Wordfence (the CVE Numbering Authority for the WordPress ecosystem) under coordinated disclosure.

2 Problem Scope

We consider an external observer with no backend access to monitored domains. An attacker compromises one or more WordPress sites by exploiting a vulnerable plugin, theme, or core component, then injects persistent JavaScript that appears in the page source on each load. The injected code may be obfuscated, polymorphic, or gated behind runtime conditions such as geolocation, user-agent, or cookie state. Our goal is to detect injected malicious JavaScript and to correlate each campaign with the frontend-observable ecosystem state (active plugins, themes, and versions) of its victim sites, producing candidate entry-vector attributions. We achieve this through longitudinal delta monitoring, behavioral trace clustering, and population-level enrichment analysis.

Operating from an external vantage point exposes three challenges that shape our pipeline design.

The Attribution Gap. Existing tools that can identify that a page has been infected [7], but they rarely reveal which vulnerable component allowed the compromise from an external vantage point. Without this information, site owners resort to superficial remediation where they remove the injected code but leave the entry point intact, resulting in rapid reinfection [18].

Visibility Constraints. Backend vulnerabilities such as SQL injection or file inclusion are invisible from an external perspective [8, 21]. However, the frontend of a CMS site often exposes fingerprintable configurations, including active themes, plugins, and version strings [21]. We focus on correlating these observable properties with newly detected injection campaigns.

Evasion and Temporal Dynamics. Adversaries use runtime generated payloads and environment-specific cloaking to evade static analysis [20]. Campaigns span a spectrum from byte-identical payloads to per-site polymorphic encodings [39].

Terminology. We use three levels of granularity throughout the paper. An *observation* is a distinct JavaScript payload identified by its SHA-256 content hash. The same hash may appear on multiple sites or recur across multiple crawl windows. A *campaign* groups observations that produce behaviorally equivalent execution traces into one logical entity via behavioral trace clustering. Two observations belong to the same campaign if and only if their dynamic behavioral traces are similar, regardless of when or where they were seen. A *behavioral family* groups campaigns that share the same overarching attack technique and infrastructure pattern but differ in operational parameters such as obfuscation keys or target domains, capturing the broader malware lineage across distinct deployment waves. For instance, one campaign delivers a ClickFix payload by fetching attacker instructions from a Polygon smart contract [35] (PolyClickFix, Section 4.2), while a second delivers a distinct ClickFix payload via a Binance Smart Chain contract [12]. Both resolve different blockchain RPC endpoints and produce distinct execution traces, making them separate campaigns. Yet both belong to the same ClickFix behavioral family because they share the overarching technique of tricking site visitors into running attacker-supplied PowerShell script.

3 Methodology

Our pipeline executes five stages in an iterative cycle, repeated after each crawl window (approximately every 2.5 days): *longitudinal JavaScript harvesting* (§3.1), *delta detection and triage* (§3.2), *dynamic behavioral tracing* (§3.3), *behavioral clustering for campaign discovery* (§3.4), and *frontend fingerprinting and ecosystem attribution* (§3.5). Campaigns accumulate across windows as new observations are discovered and linked to prior ones. Figure 1 provides an end-to-end overview of a single iteration.

The central design decision is combining static content-hash analysis with behavioral tracing, motivated by the diversity of obfuscation strategies observed across campaigns (described in detail in Section 4.2). Campaigns exist on a spectrum of code diversity. ① Some campaigns deploy byte-identical payloads across all victim sites, which static hash comparison can efficiently detect. ② Others campaigns generate a unique encoding for every deployment wave, making each payload appear distinct at the byte level and rendering static deduplication ineffective. ③ A third group of campaigns falls between these extremes, reusing the same code structure but varying parameters such as command-and-control domains, encryption keys, or obfuscator seeds across batches. A purely static pipeline fails to group the second and third classes into campaigns, while a purely dynamic pipeline is too expensive to apply at web scale without a filtering stage. Our hybrid design resolves this tension. Static differencing and Subresource Integrity (SRI) allowlist filtering (§3.2) serve as a cheap first pass that reduces the candidate set by orders of magnitude, while behavioral tracing and clustering operate on remaining deltas to unify campaigns whose runtime behavior remains consistent despite divergent code representations.

3.1 Longitudinal JavaScript Harvesting

Watchlist construction. We derive our target population from the Common Crawl [6], extracting all domains whose landing page includes at least one WordPress-specific URL path marker (e.g., `/wp-content/plugins/`, `/wp-includes/`). After deduplication and DNS-reachability filtering, this yields a watchlist of about 850,000 unique WordPress domains that we monitored throughout the study period.

Crawl architecture. Each crawl iteration visits every domain and persists a complete snapshot of the JavaScript code served to visitors. The harvester uses asynchronous HTTP/2 requests with connection pooling, maintaining a bounded number of in-flight workers via a shared work queue. For HTML parsing, we employ a lightweight event-driven parser that processes only `<script>` elements (both `src`-referenced and inline), avoiding full Document Object Model (DOM) construction to minimize per-page overhead. An adaptive concurrency controller monitors the ratio of connection timeouts to completed requests within a sliding window, adjusting the number of concurrent workers to saturate bandwidth without destabilizing the infrastructure. The watchlist is partitioned across 20 parallel crawler instances, with each partition retrying (3 retries) failed domains with exponential backoff.

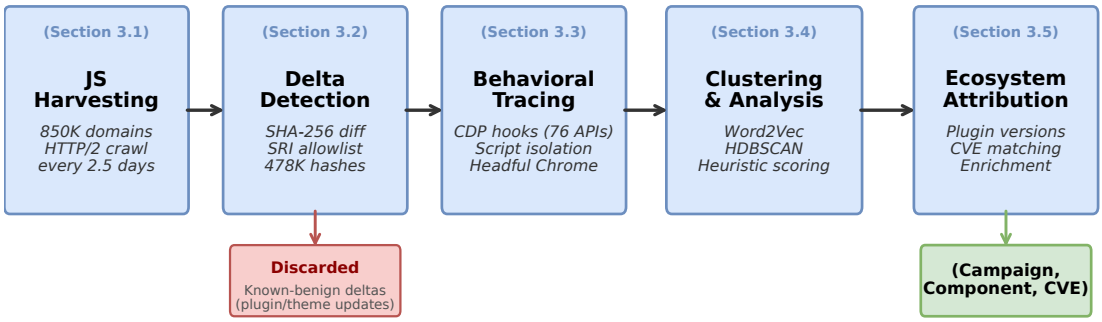


Figure 1: End-to-end methodology pipeline. Blue boxes represent processing stages; the green box is the final attribution output (campaign, component, CVE). Every JavaScript delta that survives the SRI allowlist filtering (discarded deltas shown beneath the Delta Detection stage) is traced in an instrumented browser and clustered by behavioral similarity.

Script storage. For each domain, the crawler downloads every external script and captures every inline script block, recording the DOM position index of each element. External scripts are streamed to disk with content-type sniffing to reject non-JavaScript resources. A per-domain metadata record logs the resolved URL, HTTP status codes, total script count, and DNS errors.

Cadence. We recrawl the full watchlist approximately every 2.5 days. Each crawl produces a timestamped snapshot directory, yielding a longitudinal time series of JavaScript code from which temporal deltas are computed.

3.2 Delta Detection and Triage

Between consecutive snapshots $T-1$ and T, we compute a *JavaScript delta* for each domain by comparing SHA-256 content hashes. A delta is generated whenever a script is entirely new (no matching hash in $T-1$) or when an existing script's hash has changed. This byte-exact comparison ensures that even a single injected character produces a distinct hash, capturing subtle injection techniques such as appending a payload to a legitimate library.

SRI allowlist filtering. A large fraction of deltas correspond to benign activity such as plugin updates, theme updates, and CDN version rotations. We maintain a Subresource Integrity (SRI) allowlist of 478K SHA-256 hashes, constructed from two sources and updated biweekly. First, we enumerate every publicly listed plugin and theme on WordPress.org via its Subversion repository, export up to seven of the most recent stable release tags for each component, and compute SHA-256 hashes of all JavaScript assets shipped in those releases. Second, we retrieve up to seven recent versions of each library indexed by CDNJS [4] and hash their distributed JavaScript files. The union of both hash sets forms the allowlist. Any delta matching the allowlist is classified as known-benign and excluded. Every remaining delta is forwarded to the dynamic behavioral tracing stage.

3.3 Dynamic Behavioral Tracing

For each remaining delta, we re-visit the affected domain in a headful Chromium instance controlled via the Chrome DevTools Protocol (CDP). We chose CDP over higher-level frameworks (e.g., Puppeteer) for its direct access to low-level APIs (e.g., `Debugger.scriptParsed` for script resolution and `Runtime.consoleAPICalled` for capturing structured instrumentation output).

Instrumentation. Before page navigation, we inject a monitoring script via `Page.addScriptToEvaluateOnNewDocument` that interposes on 76 browser APIs spanning 14 categories (Appendix E). Each interposed API emits a structured JSON event containing the event type and type-specific parameters, captured via `Runtime.consoleAPICalled` with full call stacks. A re-entrancy guard prevents the monitoring script from triggering on its own activity.

Synthetic interactions. To trigger JavaScript that activates only on user engagement (scroll-triggered redirectors, click-jacking overlays, lazy-loaded skimmers), the tracer follows a deterministic interaction sequence after a page load: (1) a smooth scroll to the vertical midpoint of the viewport (50% of `document.body.scrollHeight`) at 500 px/s; (2) a scroll back to the top at the same rate; (3) dispatched `MouseEvent` click events at six normalized viewport coordinates: center (50%, 50%), upper-left (10%, 10%), upper-right (90%, 10%), lower-center (50%, 90%), lower-left (10%, 90%), and lower-right (90%, 90%) Each click is preceded by a 200 ms delay to allow event-handler registration. A configurable idle-capture window (default 10 s) is used to collect delayed callbacks and timer-based payloads.

Target script isolation. Each tracing job is parameterized with the SHA-256 hash of the script that executes the corresponding code and, when available, its URL. After page load, the tracer resolves the target script's CDP-assigned `scriptId`. For external scripts, the tracer normalizes the URL by stripping the scheme (`http://` vs. `https://`) and lowercasing the hostname, then compares the result against each `Debugger.scriptParsed` event and verifies the content hash via `Debugger.getScriptSource`. For inline scripts (which lack a resolvable URL), the tracer iterates over all parsed inline scripts and matches by content hash. The full event stream is

then filtered to retain only events whose `stackTrace.callFrames` reference the target `scriptId`, isolating the behavioral footprint of the changed script from unrelated page activity.

Deployment. The system runs as a containerized worker pool in which a Redis-backed task queue distributes jobs across 25 scan worker replicas, each managing isolated Chromium browser contexts. A circuit breaker (50,000 event limit) and per-job timeout (300 s) prevent runaway scripts from blocking the pipeline.

3.4 Behavioral Clustering for Campaign Discovery

After each crawl window, remaining deltas are traced and the resulting execution traces are clustered to surface new campaigns. The goal is to group execution traces from the current window with similar runtime behavior into clusters, where each cluster corresponds to a candidate campaign.

Trace tokenization. Each raw trace is converted to a sequence of behavioral tokens. For example, an `XMLHttpRequest` (XHR) call to `evil.com/api/callback?sid=abc123` becomes `XHR_Request-|host=evil.com|path=/api/callback`, and a cookie write of `waf_sc=5889647726` becomes `Cookie_Write|name=waf_sc`. The central challenge in tokenization is choosing a level of abstraction that groups traces from the same campaign together while separating traces from different campaigns. If tokens retain too much detail (e.g., full URLs with query parameters), every victim produces a unique token sequence and traces from the same campaign never match. If tokens are too coarse (e.g., retaining only the event type), distinct campaigns produce identical sequences and become indistinguishable. We address this by applying per-field normalization rules that strip specific variations while preserving the structural details that distinguish one campaign from another. For example, URL fields retain only the hostname and a short path prefix, discarding query parameters and fragments that change per victim, cookie events retain only the cookie name, not its value. The general principle is that each field is reduced to the coarsest representation that still separates the campaigns we observe empirically.

Embedding. To compare behavioral tokens numerically, we learn a vector representation for each token. A Word2Vec skip-gram model [23] trains on the full token corpus from each crawl window, mapping each token to a 64-dimensional vector such that tokens appearing in similar behavioral contexts are placed close together. Embeddings are used solely for within-window distance computation. Cross-window campaign linkage relies on behavioral trace equivalence and shared command-and-control infrastructure rather than embedding proximity, so per-window retraining does not affect campaign continuity across windows.

Distance computation. We compute pairwise trace distances using Dynamic Time Warping (DTW) [33]. DTW takes two token sequences and finds the best alignment between them, allowing tokens to be matched even when the sequences differ in length or ordering. The cost of matching any two tokens is their cosine distance in the Word2Vec embedding space: $c(t_i, t_j) = 1 - \frac{\vec{v}_i \cdot \vec{v}_j}{\|\vec{v}_i\| \, \|\vec{v}_j\|}$. Tokens that represent semantically similar behaviors (e.g., two `fetch` calls to different domains) are mapped to nearby vectors and contribute low alignment cost, even when their surface forms differ. Conversely, sequences where the same API calls appear in a fundamentally different order receive higher distances, reflecting different campaign behavior.

Clustering. We cluster traces using HDBSCAN [2] (`min_cluster_size=2`), chosen because the algorithm does not require specifying the number of clusters *a priori*, handles clusters of varying density (campaigns range from hundreds of victims to single-digit infections), and assigns noise labels to outlier traces that do not belong to any meaningful cluster, naturally separating one-off injections from cohesive campaigns.

Heuristic-based cluster prioritization. HDBSCAN produces thousands of clusters, most of which correspond to benign plugin activity, making manual inspection of every cluster infeasible at this scale. We therefore apply two complementary sets of heuristic rules in parallel with clustering to score every data point in the corpus, then project those per-trace scores onto each cluster to prioritize manual review (the full rule inventories in Appendix D).

Static heuristics (37 weighted rules) operate on the raw JavaScript source code. Rules span seven categories, including obfuscation signatures (e.g., `_0x` hex naming, JSFiretruck encoding, XOR decryption loops), suspicious API calls (`eval`, `createElement('script')`, `document.write`), evasion patterns (WordPress cookie checks, bot user-agent detection, session gating), DOM manipulation (CSS-fade overlays, hidden iframes), network and C2 patterns (multiple base64-encoded URLs, fingerprint exfiltration), WordPress-specific indicators (rogue user creation, REST API abuse), and entropy/size anomalies (single-line packing, high Shannon entropy).

Behavioral heuristics (33 weighted rules) operate on the CDP execution traces produced in the preceding stage. Rules cover five categories, including network communication (XHR/fetch to external domains, beacon exfiltration, blockchain RPC calls, known malicious domains), code injection (`document.write` with iframes or scripts, `eval`/`Function` constructor invocation, layered `atob` deobfuscation), browser fingerprinting (heavy DOM property reads, hardware enumeration, WebGL context creation), event hooking and persistence (excessive listeners, click/keyboard hijacking, ad-network keywords, `defineProperty` abuse, visibility-change persistence), and data exfiltration (cookie-gated execution, Post message communication). Behavioral rules account for the fact that the monitoring harness itself generates events during initialization (e.g., property hooks, synthetic scroll interactions). These harness-generated events are excluded before scoring so that only events triggered by the target script contribute to the heuristic score.

Out of our 70 total rules, 26 (16 static and 10 behavioral) target indicators that are well established in the malicious JavaScript detection literature. Obfuscation signatures and suspicious API calls such as `eval` and `createElement` are similar to features used by Zozzle [7] and JaSt [9], entropy and packing anomalies resemble the page-level filters of Prophiler [3] and Cujo [31], and runtime network and exfiltration patterns draw on the behavioral signatures of WeBehavior [34]. Those systems train classifiers to label individual scripts, whereas our rules score traces to *rank clusters* for manual review. Among the 26 "established" features, fifteen (12 static, 3 behavioral) draw specifically on the obfuscation taxonomy by Zhou et al. [45]. The remaining 44 rules are domain-specific,

seeded from public threat intelligence reports on active WordPress campaigns [10, 30, 39] and from manual source-code analysis of payloads encountered during the study. The full rule inventories are summarized in Appendix D.

For each cluster, we aggregate the per-trace heuristic scores of its members and rank clusters by a composite priority score that combines cluster size, mean heuristic score, and payload diversity (whether member payloads differ by content hash). This ranking allows analysts to focus on the highest-priority clusters first. Section 4.1 quantifies the resulting filtering funnel, reporting the number of clusters produced, flagged, and confirmed at each stage. The heuristic rules are a prioritization mechanism rather than a contribution of this paper, and Appendix G evaluates both rule sets against 5,000 human-verified benign JavaScript files to confirm that the assigned weights reflect empirical specificity.

3.5 Frontend Fingerprinting and Ecosystem Attribution

For every domain assigned to a campaign cluster, we extract a *frontend ecosystem fingerprint* consisting of active WordPress plugins, themes, and version strings. WordPress assigns every plugin and theme a *slug*, a short lowercase unique identifier that serves as the component's canonical name throughout the ecosystem (e.g., `yoast-seo` or `astra`). Extension assets are served from standardized paths (`/wp-content/plugins/{slug}/` for plugins and `/wp-content/themes/{slug}/` for themes), so the first path segment after the marker yields the component slug [21]. This identification is performed passively during crawling without additional requests. Version resolution proceeds through two strategies. The primary method parses the query string of each asset URL for the `ver`, `version`, or `v` parameters that WordPress appends when enqueuing scripts (e.g., `style.css?ver=5.2.1`). When no query parameter is present, the extractor falls back to filename pattern matching, searching for a numeric version segment preceded by a hyphen or underscore (e.g., `plugin-name-1.2.3.js`). In both cases, candidate strings are validated against a strict numeric format to reject non-version identifiers such as random strings or timestamps that some sites append to asset URLs to bypass browser caching. As a secondary method, we compute SHA-256 hashes of all assets (CSS, JavaScript, HTML fragments) loaded under each plugin's URL path during runtime and match them against a pre-built index of asset hashes from the WordPress plugin repositories across all publicly available releases. When a site serves a plugin asset whose hash matches a specific release version in the repository, we infer the installed version even when the `ver` parameter is absent or unreliable. Despite both strategies, **13.1**% of components still lack a parseable version. However, even without a resolved version, the plugin slug itself remains useful. The enrichment analysis (Section 4.5) operates on component identity, so a versionless component that is disproportionately concentrated on victim sites is still flagged as a candidate entry vector. Version information, when available, strengthens the signal by narrowing the match to specific CVE-affected ranges, but its absence does not prevent a component from contributing to the attribution.

For each campaign, we aggregate the fingerprinted components across all domains whose traces were assigned to that campaign's cluster. We then cross-reference these components against public vulnerability databases (National Vulnerability Database (NVD), WPScan, Wordfence) to identify CVEs whose affected version ranges overlap with the observed versions. The resulting (campaign, component, CVE) triples form our attribution output. This attribution is correlational: co-occurrence does not prove that a component was the entry vector, but version-aware filtering narrows the candidate set to actionable remediation targets. Critically, enrichment also surfaces components that are disproportionately common among victim sites yet lack a public CVE. These components become zero-day candidates for source-code audit, because their concentration on compromised sites suggests an undisclosed vulnerability that attackers may already be exploiting. Section 4.6 reports several such discoveries confirmed through manual audit. We describe the enrichment methodology and its full results in Section 4.5.

4 Results

We ran our pipeline on about 850,000 (846,067) WordPress domains from December 28, 2025 through March 22, 2026, completing **24 crawl windows** over an 84-day study period with a mean cadence of 2.5 days. Across all windows, the crawler achieved an 83–86% success rate (domains returning parseable HTML with at least one script), collecting over 605 million JavaScript artifacts (377 million external scripts and 228 million inline blocks). The main failure modes were connection timeouts (7.5%), HTTP 403 responses (3.1%), and unreachable hosts (2.0%), consistent with the natural churn expected when monitoring a population of this scale. Each full crawl completed in 4–5 hours. Our pipeline surfaced **61 observations** (61 distinct malicious payload hashes collectively detected across **1,160** compromised sites) grouped into **20 campaigns** spanning 13 behavioral families. This section presents the campaign landscape, validates our hybrid detection approach through similarity analysis, reports the VirusTotal detection gap, the ecosystem correlations recovered for each campaign, and the resulting operational impact.

4.1 Heuristic Filtering Funnel

Table 1 reports the per-window filtering funnel. Each crawl-window diff examines ∼17.7 million scripts common to the current and previous snapshot, of which ∼2.67 million exhibit a byte-level change (raw deltas). SRI allowlist filtering then removes assets whose SHA-256 matches our precomputed integrity-hash corpus, leaving ∼1.10 million deltas. These resolve to ∼694K unique content hashes spanning ∼335K distinct JavaScript file paths. We see more unique hashes than file paths because the same path served on different WordPress sites is often polymorphic, as sites run different plugin versions, embed site-specific inline configuration, or apply developer customizations, each producing a distinct SHA-256. Behavioral tracing of these unique scripts and subsequent HDBSCAN clustering yields ∼10,000–11,000 clusters per window. Heuristic scoring and composite ranking flags on average 400–500 candidate clusters per window (∼**95%** reduction), which a team of two analysts reviewed before the next window completed.

Over the full 24-window study period, this process surfaced 46 confirmed malicious clusters. We then linked confirmed clusters from different crawl windows into campaigns by matching clusters

Table 1: Per-window filtering funnel and study-wide confirmation totals. Funnel counts are representative per-window values drawn from the latest window.

Per-Window Stage	Count
JS script positions diffed	~17.7 M
Raw deltas (common-diff ≥ 1)	~2.67 M
Deltas after SRI filtering	~1.10 M
Unique delta hashes	~694 K
Distinct JS file paths	~335 K
HDBSCAN clusters formed	~10,000
Heuristic-flagged clusters (↓ ~95%)	~400–500
Study-Wide Total (24 windows)	**Count**
Confirmed malicious clusters	46
Canonical campaigns	20
Analyst effort (total)	~96 h

that produce equivalent behavioral traces or share command-and-control infrastructure, yielding the 20 campaigns reported below. The rejected clusters were predominantly benign plugin update waves with incidentally elevated scores. The majority of flagged clusters were dismissed within seconds during a rapid triage pass (e.g., recognizable plugin updates with incidentally elevated heuristic scores), while the remaining candidates required deeper investigation, averaging 15 to 30 minutes each. Much of this investigation time is spent triaging legitimate advertising networks and commercial bot detection frameworks such as reCAPTCHA, whose payloads routinely rely on packed string array lookups, base64 and XOR decoding routines, runtime Function constructor invocation, and canvas or WebGL fingerprinting, all of which our static and behavioral heuristics are designed to flag, and an analyst can only dismiss such a cluster after decoding the payload, inspecting its outbound network activity, and verifying that the observed endpoints belong to a known benign vendor rather than an attacker-controlled endpoint. This two-tier review process required approximately **96 analyst-hours** over the entire study period.

4.2 Campaign Discovery and Classification

Table 2 summarizes the 20 campaigns. The 61 observations collectively affect 1,160 distinct sites.

We name campaigns after salient behavior or infrastructure markers, such as *HexArray* for a hex-encoded string array loader, *PolyClickFix* for a Polygon-anchored ClickFix family, and *WooBackdoor* for a WooCommerce REST-API backdoor. When a campaign matches a family already documented in public threat intelligence, we retain the community-assigned name and cite the originating report in Appendix F, which expands each entry with its fingerprintable indicators and full behavior description.

WooBackdoor: Creates a hidden administrator account on WooCommerce stores and uses the WooCommerce REST API to exfiltrate customer records, orders, and payment metadata.

PolyClickFix and MultiC2: Weaponize compromised sites as ClickFix delivery beachheads that trick visitors into running attacker-supplied PowerShell, installing an infostealer or remote-access trojan on the visitor's own machine [10, 12, 30, 35, 40].

MalvertKit: Fingerprints visitors through canvas, WebGL, and font-probing channels and opens pop-under windows to attacker-controlled advertising that stages fake-giveaway scams and browser-notification permission harvesting.

JSFiretruck, SessHijack-A/B/C, FadeRedirect, TDSRedirect, and YWXILoader: Hijack visitor navigation and route it into a server-side traffic distribution service that delivers phishing portals, fake software updates, affiliate-fraud forms, and push-notification permission prompts [11, 26, 27].

TagInject-A/B and CSSInject: Plant external-payload channels that the attacker can repoint at any time, giving them a persistent foothold for redirects, fingerprinting, or further malware delivery.

HexArray-A/B/C and GDPRInject: Conduct advertising fraud by displacing the site's legitimate ad inventory with fraudulent impressions the attacker is paid for.

CookieLoader: is a cookie-gated redirect loader whose second stage activates only after a marker cookie is set on the visitor's browser.

OverlayInject: injects a DOM overlay that covers portions of the victim page with attacker-controlled content.

PolyClickFix, *MultiC2*, and *YWXILoader* in Table 2 each demonstrate distinct polymorphism strategies that evade static detection while preserving identical runtime behavior. Three payloads with no byte-level similarity, generated using different `obfuscator.io` keys, produced identical execution traces and clustered with a Word2Vec cosine similarity of 1.0, whereas signature-based approaches would treat them as unrelated. In *MultiC2*, the three observations differ significantly at both the byte (0.11) and AST (0.18) levels, where each victim site wraps the injected code inside different jQuery event handlers and plugin-specific callback structures, introducing substantial syntactic variability. Nevertheless, their runtime behavior remains equivalent and is correctly unified through behavioral clustering. In *YWXILoader*, all 33 victims yield distinct hashes despite identical injected code, as variations in the surrounding WordPress context perturb the diff output. Here, behavioral clustering isolates a consistent `createElement→insertBefore` pattern from contextual noise, recovering the underlying campaign identity. Collectively, these cases capture the three polymorphism strategies discussed in Section 4.3.

4.3 Static vs. Behavioral Similarity

We computed pairwise within-campaign similarity at three levels: *byte* (raw binary Jaccard similarity over content bytes), *Abstract Syntax Tree (AST)* (normalized Jaccard similarity over the ordered token sequence from the Esprima JavaScript parser), and *behavioral* (DTW-based behavioral similarity as defined in Section 3.4).

Table 3 reveals that behavioral similarity remains at 1.0 for every campaign regardless of polymorphism strategy, while byte and AST metrics collapse for polymorphic families.

Table 2: Summary of the 20 campaigns (Dec. 28, 2025–Mar. 22, 2026). *Obs*: distinct injected-payload hashes; *Vuln. Comp.* and *CVEs* count components with in-range CVEs. *StoXSS*, *SQLi*, and *Auth* denote Stored XSS, SQL Injection, and Authentication/Privilege weaknesses. ✓ indicates at least one matching CVE.

Campaign	Family	Obs.	Sites	Vuln. Comp.	CVEs	StoXSS	SQLi	Auth
HexArray-A	Ad Injection	2	52	5	11	✓	✓	–
HexArray-B	Ad Injection	2	53	2	4	✓	✓	–
HexArray-C	Ad Injection	1	20	2	4	✓	✓	–
GDPRInject	Ad Injection	6	87	4	13	✓	✓	–
SessHijack-A	Session Redirect	1	20	14	59	✓	✓	✓
SessHijack-B	Session Redirect	1	24	26	97	✓	✓	✓
SessHijack-C	Session Redirect	1	23	7	18	✓	–	–
FadeRedirect	Session Redirect	4	41	17	61	✓	–	✓
MultiC2	Multi-C2 Inject	3	62	26	76	✓	–	✓
JSFiretruck	JSFiretruck Redirect	27	565	33	109	✓	✓	✓
MalvertKit	Malvertising	1	22	33	103	✓	✓	✓
TagInject-A	Tag Injection	1	20	10	33	✓	✓	✓
TagInject-B	Tag Injection	1	20	9	32	✓	✓	✓
CSSInject	CSS Injection	1	28	21	78	✓	–	✓
PolyClickFix	Blockchain ClickFix	4	70	43	149	✓	✓	✓
CookieLoader	Cookie Loader	1	2	1	5	✓	–	–
TDSRedirect	TDS Redirect	1	7	16	69	✓	✓	✓
WooBackdoor	WooCommerce Backdoor	1	9	16	60	✓	✓	✓
OverlayInject	Overlay Injector	1	2	7	30	✓	–	✓
YWXILoader	CDN Loader	1	33	30	102	✓	–	✓
Aggregate †		**61**	**1,160**	**15**	**60**	**20/20**	**13/20**	**14/20**

†Obs. and Sites are totals; Vuln. Comp. and CVEs are medians across all 20 campaigns (not weighted by site count);
StoXSS/SQLi/Auth are campaign counts. Vuln. Comp. counts only components whose observed frontend version falls within a known CVE's affected range.

Table 3: Within-campaign pairwise similarity at three abstraction levels. *Byte*: raw binary. *AST*: structural token similarity. *Behav.*: DTW behavioral similarity (Word2Vec token cost) used by HDBSCAN.

Campaign	Byte	AST	Behav.
Polymorphic campaigns			
PolyClickFix	0.00	0.93	1.00
JSFiretruck	0.08	1.00	1.00
YWXILoader	0.41	0.51	1.00
FadeRedirect	0.26	0.49	1.00
MultiC2	0.11	0.18	1.00
Near-identical payloads			
GDPRInject	0.81	0.90	1.00
14 uniform campaigns (all metrics 1.00)			

The following strategies, used by five campaigns, are aimed at defeating different static analysis layers:

Per-build obfuscation (PolyClickFix). Three payloads processed through `obfuscator.io` with different encryption keys produce zero byte similarity, yet all three execute the same 14-token behavioral trace (Polygon RPC queries, smart-contract resolution, C2 callback), yielding perfect behavioral similarity. While 14 tokens is short, trace lengths across the 20 campaigns range from 3 tokens (CookieLoader, a minimal script loader) to 847 tokens (MalvertKit, a complex advertising toolkit), with a median of 42 tokens. PolyClickFix's brevity reflects the payload's narrow behavioral scope (smart-contract query followed by a single C2 callback) rather than aggressive normalization by our analysis. AST similarity remains high (0.93) because `obfuscator.io` alters variable names and string encodings without changing the underlying control flow or statement structure. A hash-based or signature-based system would treat these as three unrelated injections.

Per-campaign encoding (JSFiretruck). Each of the 27 observations has a unique JSFiretruck encoding (byte similarity 0.08), but the underlying program structure is identical (AST 1.00) because the encoding permutes character representations without changing the syntax tree. Behavioral similarity remains 1.0 because the Word2Vec embeddings map per-wave token variants to identical vectors.

Injection-point polymorphism (YWXILoader). Each of the 33 victim sites has a unique delta content hash because an identical four-line injection appears within different per-site WordPress contexts (surrounding scripts, theme code, plugin ordering). This context noise lowers both byte (0.41) and AST (0.51) similarity, yet behavioral similarity remains 1.0 because the embeddings isolate the shared `createElement`→`insertBefore` pattern from noise.

Combined polymorphism (FadeRedirect, MultiC2). FadeRedirect's four variants exhibit low byte similarity (0.26) and moderate AST similarity (0.49), with each variant loading its second-stage script from a different compromised host. MultiC2 has the lowest byte (0.11) and AST (0.18) similarity of any campaign, because its three observations appear in different WordPress plugin contexts whose

site-specific jQuery hooks produce highly variable raw token sets. In both cases, learned embeddings map parameterized token variants to near-identical vectors, keeping behavioral similarity at 1.0.

GDPRInject shows minor payload variation (byte 0.81, AST 0.90) with perfect behavioral agreement. The remaining 14 campaigns exhibit completely uniform payloads with all three similarity metrics at 1.00, encompassing HexArray-A/B/C, MalvertKit, SessHijack-A/C, TagInject-A/B, SessHijack-B, CSSInject, CookieLoader, TDSRedirect, WooBackdoor, and OverlayInject, each traced and clustered successfully during live deployment.

Together, our results demonstrate that static similarity metrics are insufficient. Byte hashing fails for PolyClickFix, AST analysis fails for YWXILoader and FadeRedirect, yet the Word2Vec cosine similarity used by HDBSCAN remains at 1.0 for every campaign as the Word2Vec embedding space absorbs per-victim and per-wave variation that would fragment raw token matching.

4.4 VirusTotal Detection Gap

We compared our pipeline's first detection of each campaign with VirusTotal's [41] coverage over the same period. For every distinct payload observed during the study, we uploaded the JavaScript file directly to VirusTotal each week and recorded the maximum number of engines that ever flagged the payload as malicious. Figure 3 in Appendix H plots the per-campaign timeline.

Eighteen of the 20 campaigns were never flagged by a single VirusTotal engine throughout the 84-day study window, across the initial upload and every subsequent weekly rescan. These non-detections (false negatives) occurred both with low-sophistication scripts (HexArray, under 1 KB) and highly complex payloads (171 KB MalvertKit, blockchain-based PolyClickFix). We treat VirusTotal as a black box and make no claim about why any individual engine did or did not flag a specific payload, because the engines are proprietary, their signature sets evolve continuously, and we have no visibility into their internal decisions. The MultiC2 campaign further illustrates the resulting detection gap. Our pipeline identified MultiC2 on December 30, 2025, while VT first flagged the payload on January 15, 2026 (16/61 engines), ramping to 23/61 by January 20 before plateauing at this level (37%), meaning a majority of the engines never flag the payload even after some initial detections. All JSFiretruck variants maintain a consistent 1/62 detection rate throughout the study, operationally equivalent to undetected for security workflows relying on consensus thresholds [36].

4.5 Ecosystem Attribution

We first report the scale and coverage of the fingerprinted plugin and theme data before cross-referencing component versions against published CVE databases. During our most recent crawl window, the WordPress fingerprinting pipeline recorded 2.63 million observations spanning 184,912 distinct WordPress.org plugins/themes slugs (e.g., yoast-seo, astra). Version strings are recoverable for **86.9%** of the observations, establishing that analyzable version metadata is often available even without administrative access. When we cross-reference these fingerprints against the Wordfence vulnerability database, only **3.4%** of the 184,912 distinct plugin and theme slugs have any recorded CVE. However, because the most widely installed plugins are also the most scrutinized, that small percentage accounts for **71.9%** of all component occurrences across the crawled population, meaning that vulnerability data covers the vast majority of what is actually deployed.

Our fingerprinting recovers a median of **15 vulnerable components** and **60 CVEs** per campaign across all fingerprinted plugins and themes. Enrichment analysis described below filters this broad pool to only components whose prevalence on compromised sites significantly exceeds their baseline rate in the general WordPress population, producing the smaller candidate lists in Table 4.

Prevalence-ratio ranking. Raw component prevalence alone cannot distinguish plugins that are campaign-specific from those that appear on compromised sites simply because they are popular across the WordPress install base. To separate signal from background, we adapt the *prevalence ratio*, a standard measure in epidemiology for quantifying the association between an exposure and an outcome [32], to the WordPress ecosystem setting. For a component c and campaign k, we define the *enrichment ratio*

$$E(c,k) \;=\; \frac{p_k(c)}{p_{\mathrm{bg}}(c)}$$

where $p_k(c)$ is the fraction of campaign k's compromised sites running component c, and $p_{\mathrm{bg}}(c)$ is the fraction of all sites in the background population running c. An enrichment ratio of $E \gg 1$ indicates that c is disproportionately concentrated among the campaign's victims relative to the general install base. A value of $E \approx 1$ indicates that the component's presence merely reflects its market share. We compute the background distribution directly over the full host-level population of our latest crawl window, covering **715,141** unique WordPress hosts after removing the 300 compromised hosts that appear in any campaign. Using the full crawled population as the baseline, rather than a smaller set derived from known-malicious signatures, provides a stable estimate of $p_{\mathrm{bg}}(c)$ for each observed slug and slug:version pair, and isolating the background from the compromised set prevents attacker-planted packages from inflating their own denominator.

Table 4 presents the enrichment analysis results for each campaign: the plugin and theme components whose victim-side prevalence most exceeds their background rate. We report enrichment as $\log_{10}(E)$ so that a value of 1.0 corresponds to a 10× elevation, 2.0 to 100×, and 3.0 to 1,000×. Rows are ranked within each campaign by $\log_{10}(E)$, and the *Cov.* column reports a per-campaign lower bound on the fraction of victims running at least one listed component. We report Cov. as the largest single-component prevalence in the block, which is a conservative lower bound because the fraction of victims running at least one listed component is always at least as large as the fraction running any single component alone. The rightmost column states the independent evidence of vulnerability, which is either a disclosed CVE that matches the exact slug:version pair, a finding we confirmed through source audit (†), or *attacker-planted* for packages absent from every public plugin or theme repository.

Three observations emerge from Table 4. First, every listed component exceeds the background population by at least 8.9× ($\log_{10}(E) \geq 0.95$), and the majority exceed it by four orders of magnitude or more, so the selection is not an artifact of plugin popularity. Second, most of these components map onto disclosed CVEs at the exact slug:version pair that we observe on victims,

Table 4: Unusually prominent plugin and theme components in each campaign, ranked by prevalence-ratio enrichment $\log_{10}(E)$ relative to the baseline of 715,141 non-compromised WordPress hosts. *Vic.%* is the component prevalence among campaign victims, and *Cov.* is a lower bound on the fraction of victims containing at least one listed component, estimated by the largest *Vic.%* in each campaign. Evidence includes an exact slug:version CVE match, a source-audited 0-day (†), or an *attacker-planted* package absent from public repositories. Campaigns with ≤ 2 crawled victims may show inflated percentages. *SessHijack-C* and *YWXILoader* are omitted because no component met the enrichment threshold.

Campaign	Plugin:Version	Vic.%	$\log_{10}$(E)	Cov.	Evidence
HexArray-A	theme:enjoymini	40	5.5	≥98%	*unrecognized theme*
	ali-post-editor:6.9	98	5.2		*attacker-planted plugin*
	theme:visualblogger	35	4.9		*unrecognized theme*
	url-shortify:1.8.6	63	4.9		Sto. XSS: CVE-2025-32134
	simple-tags:3.28.1	35	4.0		SQLi: CVE-2025-11972, Sto. XSS: CVE-2025-0627
HexArray-B	ali-post-editor:6.9	96	5.2	≥96%	*attacker-planted plugin*
	url-shortify:1.8.6	57	4.8		Sto. XSS: CVE-2025-32134
	simple-tags:3.28.1	30	3.9		SQLi: CVE-2025-11972, Sto. XSS: CVE-2025-0627
HexArray-C	ali-post-editor:6.9	85	5.2	≥85%	*attacker-planted plugin*
	url-shortify:1.8.6	60	4.9		Sto. XSS: CVE-2025-32134
	simple-tags:3.28.1	30	3.9		SQLi: CVE-2025-11972, Sto. XSS: CVE-2025-0627
GDPRInject	theme:enjoymini	41	5.5	≥95%	*unrecognized theme*
	ali-post-editor:6.9	95	5.2		*attacker-planted plugin*
	url-shortify:1.8.6	61	4.9		Sto. XSS: CVE-2025-32134
	simple-tags:3.28.1	39	4.0		SQLi: CVE-2025-11972, Sto. XSS: CVE-2025-0627
SessHijack-A	theme:oceanwp:3.5.8	10	3.3	≥10%	Auth: CVE-2025-8944, Sto. XSS: CVE-2024-5647, CVE-2025-5524
SessHijack-B	sfwd-lms:4.18.1	8	3.8	≥8%	Auth: CVE-2025-24662
FadeRedirect	salient-core:1.0	7	1.2	≥7%	File: CVE-2024-3812, Refl. XSS: CVE-2023-48748, Sto. XSS: CVE-2023-48749
MultiC2	theme:diza:1.0.6	3	∞	≥3%	File: CVE-2025-52729, CVE-2025-49261
JSFiretruck	fusion-builder:1	5	1.2	≥5%	RCE: CVE-2024-13345, Sto. XSS: CVE-2024-12477, CVE-2025-1665, CVE-2025-6747, CVE-2025-49940
MalvertKit	eventON:3.1	5	3.4	≥5%	Auth: CVE-2025-47565, CVE-2025-47564, Sto. XSS: CVE-2025-3527, Refl. XSS: CVE-2023-7200
TagInject-A	theme:puca:2.1.4	10	4.6	≥10%	*premium, not auditable*
	add-to-any:1.1†	10	0.95		Host hdr CSRF to XSS, §4.6
TagInject-B	theme:puca:2.1.4	10	4.6	≥10%	*premium, not auditable*
CSSInject	theme:astra:4.12.1	18	2.3	≥18%	Sto. XSS: CVE-2024-2347, CVE-2024-29768
PolyClickFix	woolentor-addons:3.3.1†	6	3.6	≥6%	CVE-2026-6287, §4.6
	go_pricing:3.3.19	1	2.1		Auth: CVE-2023-2494, CVE-2023-2496, Sto. XSS: CVE-2023-2498
CookieLoader	tablepress:1.9	100	∞	100%	Sto. XSS: CVE-2024-9595, CVE-2025-2685, CVE-2025-5096, CVE-2025-9500, CVE-2025-12324
TDSRedirect	download-monitor:5.1.8†	14	2.9	≥14%	Sto. XSS and CSRF, §4.6
	goodlayers-core:1.3.9	14	2.0		Sto. XSS: CVE-2024-12163, CVE-2024-11357, Refl. XSS: CVE-2024-11200
WooBackdoor	pixelyoursite-pro:2.1.3	11	2.1	≥11%	Sto. XSS: CVE-2023-2584
OverlayInject	facebook-pagelike-widget:1.0	50	3.2	100%	Sto. XSS: CVE-2024-0973, CVE-2024-13207
	revslider:6.0	100	2.5		Auth: CVE-2025-10249, Sto. XSS: CVE-2024-8107, CVE-2024-37449, CVE-2024-4581

† Source-audited 0-day confirmed with a proof-of-vulnerability (Section 4.6).

which turns the prevalence-ratio ranking into a near-automatic vulnerability-attribution pipeline covering Stored and Reflected XSS, SQL injection, authentication and privilege weaknesses, remote code execution, and file inclusion. Third, three candidates that the enrichment step flagged turned out to have no matching public CVE at all, and we audited their source code to characterize them, finding 0-days in `add-to-any:1.1` (a host-header-controlled CSRF-to-XSS chain used by TagInject-A, Section 4.6), in `woolentor-addons:3.3.1` (a REST-API authenticated arbitrary function call used by PolyClickFix, now published as CVE-2026-6287 [43]), and in `download-monitor:5.1.8` (a Stored XSS via unescaped shortcode output together with a CSRF bypass where nonce verification is called but its return value is never checked, used by TDSRedirect).

The *Cov.* column translates directly into remediation guidance. A high Cov. value (e.g., CookieLoader at 100% or HexArray-A at ≥98%) means the campaign's victims are concentrated on a single fingerprintable component, so patching that one plugin or theme plausibly protects most of the compromised population. A low Cov. value (e.g., MultiC2 at ≥3%, JSFiretruck and MalvertKit at ≥5%) indicates a more diverse victim install base, so external security analysts need a longer remediation shortlist to cover the same fraction of victims. Alternatively, low coverage may indicate that a campaign spread through a vector other than a vulnerable plugin, such as credential theft or a hosting-level compromise. Notably, Cov. values follow a bimodal pattern across campaigns: either a single component accounts for most victims (85–100%) or no component dominates (3–18%), suggesting a categorical distinction between plugin-exploiting and non-plugin-exploiting campaigns.

The HexArray and GDPRInject families present a distinct pattern. Their top-ranked candidates, `ali-post-editor:6.9`, do not appear in the WordPress.org directory, CodeCanyon, or any publicly

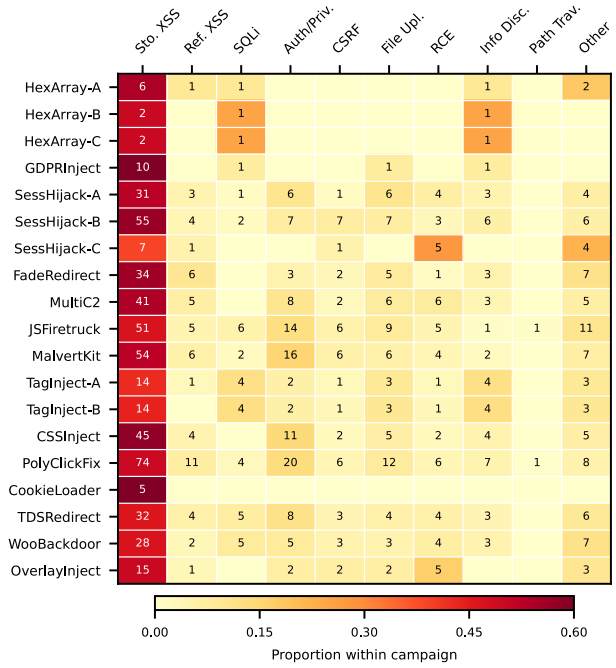


Figure 2: CVE type distribution across campaigns; cells show raw counts, with darker shading indicating higher within-campaign proportions. YWXILoader is omitted because no version-matched plugin CVEs remained after enrichment filtering.(*Sto. XSS* = Stored XSS, *Refl. XSS* = Reflected XSS, *SQLi* = SQL Injection, *Auth/Priv.* = Authentication/Privilege, *CSRF* = Cross-Site Request Forgery, *File U/I* = File Upload/Inclusion, *RCE* = Remote Code Execution, *Info Disc.* = Information Disclosure, *Path Trav.* = Path Traversal, *Other* = Any vulnerability)

indexed mirror we searched. We classify these as attacker-planted packages used for post-compromise persistence rather than as entry vectors, and their enrichment signal is carried almost entirely by the compromised population, so their presence on any new site is a strong indicator of compromise rather than of exposure.

Figure 2 shows the per-campaign CVE distribution across the 19 enrichment-eligible campaigns, and components with Stored XSS CVEs co-occur with every displayed campaign (100%), providing the direct mechanism for injecting persistent JavaScript. Components with SQL injection CVEs co-occur with **13 of the 20 campaigns**. While SQLi does not directly inject JavaScript, SQL injection enables database-level manipulation that produces the same observable outcome. Components with authentication and privilege weakness CVEs co-occur with **14 of the 20 campaigns**, enabling attackers to gain the authenticated context required to exploit lower-severity XSS vulnerabilities or to modify site content via the WordPress admin API. This co-occurrence pattern is consistent with a multi-stage attack model in which privilege weaknesses provide the access needed to exploit injection paths. Campaigns lacking these CVE types tend to have minimal plugin footprints (e.g., HexArray with two to five vulnerable components), suggesting either a different initial access vector or fingerprinting limitations.

4.6 Validating Enrichment via Source Audit

Across 20 campaigns, our pipeline identifies more than 330 distinct plugins and themes as enriched components. For most, the enrichment signal is adequately explained by known CVEs present in vulnerability databases. A smaller set of components, however, shows high enrichment without a known vulnerability explaining their concentration on compromised sites, making them candidates for deeper investigation. Three of the four components in this pool have public source code on WordPress.org; the fourth (`theme:puca:2.1.4`) is a commercial premium theme with no publicly accessible source. We audited the public ones and produced working proof-of-vulnerability exploits for each ([†] in Table 4).

add-to-any:1.1. AddToAny has two known CVEs (CVE-2021-24616 and CVE-2021-24568 for authenticated Stored XSS), but both require authenticated or network-level access. Our audit of v1.1 revealed an unauthenticated CSRF-to-XSS chain that enables complete site takeover without prior authentication. The vulnerability was silently patched in v1.8.17, but no CVE was assigned. This case illustrates how silent patches can leave older installations exposed without any public signal to prompt updates.

woolentor-addons:3.3.1. No known CVE existed for this plugin, distributed as ShopLentor (WooLentor), at the time of our study. Our audit found that the plugin's REST API endpoint accepts arbitrary PHP function names from user input. We also found a Stored XSS vulnerability in the same plugin through which a Contributor-level user can steal an admin session and then exploit the REST endpoint, creating a two-step Contributor-to-RCE attack chain. The vendor patched the plugin after our report [13], and Wordfence published the finding as CVE-2026-6287 [43].

download-monitor:5.1.8. Our audit found a Stored XSS via the `[download_data]` shortcode, which returns download titles and filenames without HTML escaping, and a CSRF bypass where nonce verification is called but the return value is never checked. The developers reviewed our report and classified the issue as a configuration issues. Site owners can close the exposure through configuration alone. They should grant download editing rights only to fully trusted users.

Together, these three audits show that enrichment-flagged components without a matching public CVE correspond to real, exploitable vulnerabilities.

5 Discussion

The enrichment ratios in Table 4 raise a complementary question: why do campaigns compromise only a fraction of the sites running a given vulnerable plugin? The vulnerability pre-conditions observed in our source audits (Section 4.6) offer one explanation. Most attributed CVEs require authenticated access. For example, the `woolentor-addons` chain requires Contributor credentials, a crafted post, and administrator interaction before the stored payload escalates to RCE. Each pre-condition acts as a probabilistic gate that narrows the effective attack surface below the total vulnerable population. By contrast, the CSRF-to-XSS in `add-to-any:1.1`

requires no attacker credentials: the attacker only needs to lure an administrator to a crafted URL, which silently forges an authenticated request and injects the payload without any further action. This removes the authentication gate present in multi-step chains.

Table 4 also shows that individual campaigns concentrate on multiple plugins with independent vulnerability classes. HexArray targets both `url-shortify:1.8.6` (Stored XSS) and `simple-tags:3.28.1` (SQLi), two codebases with no shared code. The attacker exploits whichever vector succeeds on a given target, which explains why several plugins appear enriched for the same campaign.

This multi-plugin targeting pattern has a direct defensive counterpart in the *Cov.* column of Table 4. A high Cov. value indicates that one component dominates the compromise surface, so a single-plugin patch likely remediates the root cause for most victims even when several components are enriched. A low Cov. value signals that no single plugin covers the population, so site maintainers must sequence patches across several components or deploy compensating controls such as WAF rules to achieve comparable coverage.

6 Limitations

Correlational attribution. All attribution signals are correlational. Component co-occurrence does not establish that a specific vulnerability was the actual entry vector, and enrichment candidates are best understood as prioritized remediation hypotheses rather than confirmed exploitation paths. **Crawl coverage:** Our pipeline observes only JavaScript present on unauthenticated landing pages, so scripts gated behind logins, served through client-side routing (e.g., single page SPA), or embedded on interior pages remain invisible. The 2.5-day mean recrawl cadence does not capture injections that are deployed and removed within a single window, and the Common Crawl-derived watchlist may over-represent certain site categories relative to the broader WordPress population. **Detection boundaries:** The SRI allowlist is not exhaustive. Newly released assets may generate false-positive deltas, and a trojanized resource republished at a matching hash would pass the allowlist check undetected. Our headful Chromium instance neutralizes headless-detection cloaking, but campaigns that gate activation on geolocation, IP reputation, or interaction sequences beyond our synthetic scroll-and-click pattern may not surface during behavioral tracing. Single-observation campaigns fall into the HDBSCAN noise class by construction and are surfaced only through manual review. **Version fingerprinting and enrichment:** Version inference relies on `?ver=` query parameters and asset-hash matching against the WordPress.org repository. Some assets expose module-internal build identifiers rather than release versions, introducing noise into CVE range checks. Ubiquitous plugins co-occur with many campaigns as a base-rate artifact of market share rather than as an attribution signal, a bias the prevalence-ratio enrichment corrects for, though low-prevalence components on small victim sets can still produce inflated enrichment scores. **Scope:** The study is scoped to WordPress. The underlying methodology is platform-agnostic, but applying it to Joomla, Drupal, or Shopify would require rebuilding the fingerprinting conventions, SRI allowlists, and plugin repository integrations for each target ecosystem.

7 Related Work

Our work bridges three areas of security research, namely malicious JavaScript detection, campaign attribution, and CMS ecosystem security. Large-scale external crawling has been the dominant approach to web malware detection since Moshchuk et al. [24] measured spyware prevalence through automated browsing and Provos et al. [28] built Google's infrastructure for identifying drive-by download pages. EvilSeed [14] later introduced search-guided discovery to scale this external vantage point efficiently. All three systems operate without backend access to monitored domains, a constraint our pipeline shares. Within this tradition, the detection techniques for malicious JavaScript have matured over several generations. Early systems such as Zozzle [7] classified scripts based on Abstract Syntax Tree (AST) features, while Prophiler [3] and Cujo [31] combined page-level features with lightweight dynamic analysis. Later work explicitly addressed evasion techniques. Rozzle [20] explored multiple execution paths to defeat environment-dependent cloaking, and JaSt [9] detected obfuscated JavaScript through purely syntactic analysis without requiring execution. More recent work has shifted to dynamic behavioral analysis. VisibleV8 [16] enabled large-scale in-browser API monitoring, WeBehavior [34] showed that browser API call sequences form stable behavioral signatures even when code representations diverge, Zafar et al. [44] characterized platform-dependent execution divergence, and Chen et al. [5] focused on event-loop granularity for evasion detection.

On the attribution side, AdGraph [15] and JSgraph [22] trace which scripts are responsible for observed page modifications, enabling reconstruction of attack sequences at the script level. Our work extends this attribution one level higher, linking campaigns not to the scripts that executed the attack but to the WordPress plugins and themes that enabled attackers to inject those scripts, producing concrete patch targets rather than a forensic audit trail.

Within CMS ecosystem security, YODA [17] performed a large-scale longitudinal study of the WordPress plugin marketplace through static analysis of marketplace code, while TARDIS [18] addressed compromise remediation through server-side rollback and demonstrated that root-cause identification of the vulnerable component is essential to prevent reinfection. Our enrichment-normalized attribution pursues the same remediation goal, but operates from a purely external vantage point, enabling any analyst to monitor sites at scale without site-owner cooperation or backend access. Kondracki and Nikiforakis [21] characterized the accuracy of web application fingerprinting, directly relevant to the version extraction limitations we discuss in Section 3.5, and other studies have characterized supply chain risks [25] and vulnerable dependency notification [37]. Our pipeline connects these threads by detecting injections through longitudinal JavaScript delta monitoring and behavioral clustering, then linking each confirmed campaign to the frontend-fingerprintable ecosystem state of its victim sites, producing the attribution context that prior work lacks.

8 Conclusion

We presented a measurement pipeline that bridges JavaScript injection detection and CMS ecosystem attribution across about 850,000 WordPress domains over an 84-day period. By combining behavioral clustering with frontend fingerprinting and enrichment-normalized

analysis, an external observer with no backend access can attribute injection campaigns to specific vulnerable plugin:version pairs, surface both known CVE-bearing entry vectors and previously undisclosed vulnerabilities, and isolate attacker-planted artifacts from legitimate components. External security analysts can therefore identify the likely root cause of the compromise, enabling site maintainers to remediate the underlying vulnerable component rather than only removing the visible payload. Together, these results shift web malware monitoring from simple detection to actionable ecosystem intelligence, providing specific components to report, CVEs to prioritize, and undisclosed vulnerability candidates to investigate. As future work, extending the approach to other CMS platforms and performing deeper validation of attribution through backend access in collaboration with hosting providers are promising directions.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM.

Y.K. was supported by an appointment to the Intelligence Community Postdoctoral Research Fellowship Program at the National Institute of Standards and Technology administered by Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy and the Office of the Director of National Intelligence (ODNI).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies, sponsoring organizations, or other entities that supported this work.

References

[1] Anthropic. 2026. Claude. https://www.anthropic.com/claude.

[2] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. 2013. Density-Based Clustering Based on Hierarchical Density Estimates. *Advances in Knowledge Discovery and Data Mining (PAKDD 2013)* (2013).

[3] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. 2011. Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*.

[4] cdnjs. 2026. cdnjs – The #1 free and open source CDN. https://cdnjs.com.

[5] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *2021 IEEE Symposium on Security and Privacy (SP)*.

[6] Common Crawl Foundation. 2026. Common Crawl. https://commoncrawl.org/.

[7] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security 11)*.

[8] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2014)*.

[9] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2018)*.

[10] GoDaddy. 2024. Threat Actors Push ClickFix Fake Browser Updates Using Stolen Credentials. https://www.godaddy.com/resources/news/threat-actors-push-clickfix-fake-browser-updates-using-stolen-credentials.

[11] GoDaddy. 2025. DollyWay World Domination: Eight Years of Evolving Website Malware Campaigns. https://www.godaddy.com/resources/news/dollyway-world-domination.

[12] Guardio. 2023. EtherHiding – Hiding Web2 Malicious Code in Web3 Smart Contracts. https://guard.io/labs/etherhiding-hiding-web2-malicious-code-in-web3-smart-contracts.

[13] HasThemes. 2025. ShopLentor (WooLentor) – WooCommerce Builder for Elementor. https://wordpress.org/plugins/woolentor-addons/.

[14] Luca Invernizzi, Paolo Milani Comparetti, Stefano Benvenuti, Christopher Kruegel, M. Cova, and Giovanni Vigna. 2012. Evilseed: A guided approach to finding malicious web pages. In *Proceedings of the 33rd Symposium on Security and Privacy*.

[15] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[16] Jordan Jueckstock and Alexandros Kapravelos. 2019. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the 2019 Internet Measurement Conference (IMC '19)*.

[17] Ranjita Pai Kasturi, Jonathan Fuller, Yiting Sun, Omar Chabklo, Andres Rodriguez, Jeman Park, and Brendan Saltaformaggio. 2022. Mistrust Plugins You Must: A Large-Scale Study of Malicious Plugins in WordPress Marketplaces. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*.

[18] Ranjita Pai Kasturi, Yiting Sun, Ruian Duan, Omar Alrawi, Ehsan Asdar, Victor Zhu, Yonghwi Kwon, and Brendan Saltaformaggio. 2020. TARDIS: Rolling Back The Clock on CMS-Targeting Cyber Attacks. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[19] Martin Kleppe. 2012. JSFuck – Write any JavaScript with 6 Characters: []()!+. http://www.jsfuck.com/.

[20] Clemens Kolbitsch, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *2012 IEEE Symposium on Security and Privacy (SP)*.

[21] Brian Kondracki and Nick Nikiforakis. 2024. Smudged Fingerprints: Characterizing and Improving the Performance of Web Application Fingerprinting. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*.

[22] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2018. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2018)*.

[23] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations (ICLR 2013), Workshop Track*.

[24] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. 2006. A Crawler-based Study of Spyware in the Web. In *Proceedings of the 13th Network and Distributed System Security Symposium*.

[25] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2020)*.

[26] Palo Alto Networks Unit 42. 2024. Parrot TDS: A Persistent and Evolving Malware Campaign. https://unit42.paloaltonetworks.com/parrot-tds-javascript-evolution-analysis/.

[27] Palo Alto Networks Unit 42. 2025. JSFireTruck: Exploring Malicious JavaScript Using JS_uck as an Obfuscation Technique. https://unit42.paloaltonetworks.com/malicious-javascript-using-jsfiretruck-as-obfuscation/.

[28] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. 2008. All Your iFRAMEs Point to Us. In *Proceedings of the 17th USENIX Security Symposium*.

[29] R De Silva. 2026. alienvault IOC pulses. https://otx.alienvault.com/user/ammoniaMe96/pulses/.

[30] Rapid7. 2026. When Trusted Websites Turn Malicious: WordPress Compromises Advance Global Stealer Operation. https://www.rapid7.com/blog/post/tr-malicious-websites-wordpress-compromise-advances-global-stealer-operation/.

[31] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*.

[32] Kenneth J. Rothman, Sander Greenland, and Timothy L. Lash. 2008. *Modern Epidemiology* (3rd ed.). Lippincott Williams & Wilkins.

[33] Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* (1978).

[34] Shaown Sarker, William Melicher, Oleksii Starov, Anupam Das, and Alexandros Kapravelos. 2024. Automated Generation of Behavioral Signatures for Malicious Web Campaigns. In *Information Security – 27th International Conference (ISC 2024)*.

[35] Sekoia. 2025. ClearFake's New Widespread Variant: Increased Web3 Exploitation for Malware Delivery. https://blog.sekoia.io/clearfakes-new-widespread-variant-increased-web3-exploitation-for-malware-delivery/.

[36] Ravindu De Silva, Mohamed Nabeel, Charith Elvitigala, Issa Khalil, Ting Yu, and Chamath Keppitiyagama. 2021. Compromised or Attacker-Owned: A Large Scale Classification and Study of Hosting Domains of Malicious URLs. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*.

[37] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. 2018. Didn't You Hear Me? – Towards More Successful Web Vulnerability Notifications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2018)*.

[38] Aravind Subramanian, Pablo Tamayo, Vamsi K. Mootha, Sayan Mukherjee, Benjamin L. Ebert, Michael A. Gillette, Amanda Paulovich, Scott L. Pomeroy, Todd R. Golub, Eric S. Lander, and Jill P. Mesirov. 2005. Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences* (2005).
[39] Sucuri. 2025. SiteCheck Malware Trends Report 2024. https://sucuri.net/reports/sitecheck-malware-trends-report-2024/.
[40] Trend Micro. 2026. Through the Lens of MDR: Analysis of KongTuke's ClickFix Abuse of Compromised WordPress Sites. https://www.trendmicro.com/en_us/research/26/c/kongtuke-clickfix-abuse-of-compromised-wordpress-sites.html.
[41] VirusTotal. 2026. VirusTotal. https://www.virustotal.com.
[42] W3Techs. 2025. Usage Statistics and Market Share of Content Management Systems. https://w3techs.com/technologies/overview/content_management.
[43] Wordfence. 2026. CVE-2026-6287: ShopLentor ≤ 3.3.8 – Authenticated (Contributor+) Stored Cross-Site Scripting via Product Grid `blockUniqid` Block Attribute. https://www.wordfence.com/threat-intel/vulnerabilities/wordpress-plugins/woolentor-addons/shoplentor-woocommerce-builder-for-elementor-gutenberg-338-authenticated-contributor-stored-cross-site-scripting-via-product-grid-blockuniqid-block-attribute.
[44] Ahsan Zafar, Junhua Su, Sohom Datta, Alexandros Kapravelos, and Anupam Das. 2025. Same Script, Different Behavior: Characterizing Divergent JavaScript Execution Across Different Device Platforms. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25).*
[45] Dongchao Zhou, Lingyun Ying, Huajun Chai, and Dongbin Wang. 2026. From Obfuscated to Obvious: A Comprehensive JavaScript Deobfuscation Tool for Security Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

A Open Science

In accordance with the CCS 2026 open science policy, we commit to releasing the artifacts that underpin the measurements and claims in this paper so that independent researchers can reproduce, inspect, and extend our results. We make the paper artifacts available through the repository at :
https://zenodo.org/records/22165590 and *github.com/ucsb-seclab/From-Payload-to-Plugin* and commit to keeping this repository accessible for the entire duration of the review process.

Released artifacts. Upon publication we will release (i) the crawler and behavioral instrumentation harness used to collect the dataset, (ii) the clustering and enrichment analysis pipeline that produces the campaign attribution results in Sections 3 and 4, (iii) the heuristic rule inventories in Appendix D together with the machine-readable definitions consumed by the pipeline, (iv) the campaign catalog metadata in Appendix F, and (v) the scripts that regenerate every figure and table in the paper from the raw data.

Data release and redaction. We deliberately exclude three categories of sensitive material from the public artifact upon publication. First, the malicious JavaScript payloads themselves are withheld because redistributing live exploit code could enable reuse against unpatched sites. Second, victim domain lists are withheld to avoid directing attention to sites that may still be compromised. Third, the full crawl corpus (>100 GB) is withheld because it contains live URLs that presently serve attacker-controlled content and could expose end users to harm.

B Ethical Considerations

Our crawling and behavioral tracing uses a standard browser that performs synthetic interactions (scrolling and clicking at predetermined coordinates) to trigger engagement-gated payloads. While not passive in the strictest sense, these interactions are equivalent to those of a normal user browsing a page and do not probe, exploit, or modify any backend state. We sent no authentication attempts, vulnerability scans, or exploit payloads. The scanning infrastructure maintained a PTR record with a contact email for opt-out. We acknowledge this is a limited mechanism and received no such requests during the study period.

Published IOCs (C2 domains, exfiltration endpoints) were already publicly observable on compromised sites at the time of collection. We recognize that detailed campaign descriptions (e.g., specific payload techniques, rogue usernames) could theoretically aid attackers. We include these details because they are necessary for external security analysts to identify active infections and for site maintainers to remediate them.

We also considered notifying the 1,160 affected domains directly and found it infeasible at this scale. Most of these domains hide their registration records behind WHOIS privacy services, so the operators are not identifiable from public data. The remaining domains publish only generic role addresses such as `support@` or `info@`, which are typically read by non-technical staff, and sending compromise or vulnerability details to such an address risks exposing the information to unauthorized parties. We therefore notified at the ecosystem level and released every extracted Indicator of Compromise as public AlienVault OTX pulses [29], following common practice for web compromise disclosure at scale. Hosting providers, security vendors, and site operators can match these indicators against their own inventories.

For the three previously undisclosed vulnerabilities uncovered through the source-audit procedure of Section 4.6 (`add-to-any:1.1`, `woolentor-addons:3.3.1`, and `download-monitor:5.1.8`), we followed coordinated disclosure. On April 2, 2026 we reported the `woolentor-addons` and `download-monitor` findings to Wordfence, the CVE Numbering Authority for the WordPress plugin and theme ecosystem, each with a reproducible proof-of-vulnerability. The `add-to-any` flaw needed no report because the vendor had already fixed it silently in v1.8.17. Wordfence published the `woolentor-addons` chain as CVE-2026-6287 on May 27, 2026, and the vendor patched the plugin. The `download-monitor` developers classified their finding as a configuration matter and will not patch it, so Section 4.6 gives site owners hardening guidance rather than exploit primitives. We keep the full exploit primitives out of the public version of this paper because affected installations remain reachable in the wild.

C Generative AI Usage

We used Claude [1] for editorial improvements to enhance the clarity and readability of the manuscript. All AI-assisted suggestions were reviewed, verified for accuracy, and edited by the authors.

D Heuristic Rule Inventories

This appendix covers the 70 heuristic rules (37 static and 33 behavioral) used to prioritize clusters for manual review as described in Section 3.4. The complete rule tables, giving each rule identifier, weight, matching condition, and campaign coverage, are available in our repository at *github.com/ucsb-seclab/From-Payload-to-Plugin*. These rules are not deployed as a standalone classifier. Instead, they serve as an empirical prioritization aid that ranks clusters by suspiciousness so that analysts can focus their effort on the most promising candidates first. Many rules target indicators that are

well established in the malicious JavaScript detection literature, including AST-level obfuscation signatures and suspicious API calls identified by Zozzle [7] and JaSt [9], page-level entropy and packing anomalies used by Prophiler [3] and Cujo [31], and runtime API-call patterns studied by WeBehavior [34]. Unlike those systems, which train classifiers to label individual scripts, our rules rank clusters for manual review. Beyond these broadly shared indicators, 15 rules (12 static and 3 behavioral) draw specifically on the obfuscation taxonomy of Zhou et al. [45]. Domain-specific rules were seeded from public threat intelligence reports on WordPress injection campaigns [10, 30, 39], and the remaining rules emerged from manual source-code analysis over the 24 crawl windows.

E CDP Instrumentation Hooks

The behavioral monitoring script injected via `Page.addScriptToEvaluateOnNewDocument` (Section 3.3) instruments 76 browser APIs grouped into 14 categories, covering network I/O, DOM mutation, script injection, navigation, storage, timers, deobfuscation, sensitive data access, workers and messaging, observers, evasion, UI manipulation, canvas and push, and error and lifecycle events. Each interposed API emits a structured JSON event containing the event type and a type-specific subset of parameters. The complete table of hooked APIs is available in our repository at *github.com/ucsb-seclab/From-Payload-to-Plugin*.

We compiled this list from three sources. First, we adopted the instrumentation surfaces of prior in-browser monitoring and drive-by detection systems, namely VisibleV8 [16], WeBehavior [34], and Cujo [31]. Second, we added the APIs that public WordPress threat intelligence reporting names in live injection campaigns, drawing on Sucuri [39], Rapid7 [30], and Palo Alto Networks Unit 42 [26, 27]. Third, we manually analyzed the payloads captured during the first three crawl windows and added the APIs those samples exercised that neither of the first two sources covered. The first two sources give coverage of behaviors already known to the literature and to industry, and the third closes the gap toward the payloads that our own measurement observed in the wild.

F Campaign Catalog

The full catalog of the 20 canonical campaigns introduced in Section 4.2 is available in our repository at *github.com/ucsb-seclab/From-Payload-to-Plugin*. For each campaign it lists the fingerprintable indicators extracted by the pipeline (payload markers, C2 and exfiltration endpoints, injected file paths, and obfuscation signatures) and the behavior the campaign produces on victims and site operators, together with the public threat intelligence reports that first named the family, namely PolyClickFix [10, 12, 35, 40], MultiC2 [30], TDSRedirect [11, 26], and JSFiretruck [19, 27]. Campaign variants whose injection, C2 infrastructure, and behavior are otherwise identical are merged into a single entry (for example, HexArray-A/B/C and GDPRInject).

G Heuristic Rule Evaluation

The heuristic rules serve as a cluster prioritization aid rather than a standalone classifier (Section 3.4). To characterize their individual specificity, we evaluated all 70 rules against a human-verified ground truth corpus of 5,000 unique benign JavaScript files constructed during the study. Each file was drawn from WordPress sites whose crawl snapshots contained no injected deltas across consecutive windows and whose page content was manually confirmed as benign, then deduplicated by SHA-256 so that widely deployed libraries appear only once. We evaluated both the static source code and the corresponding execution trace of each file, so the static and behavioral columns in the released heuristic-evaluation table reflect the same population of scripts observed under both analysis modalities. The full evaluation table, covering this benign corpus and the 61 confirmed malicious campaign payloads, is available in our repository at *github.com/ucsb-seclab/From-Payload-to-Plugin*.

Static rules exhibit a higher false positive rate (22.7%) because legitimate WordPress JavaScript routinely triggers low-weight indicators such as minified single-line code (48% of benign files), `insertBefore` calls (26%), and `createElement('script')` (20%). Because these patterns appear in nearly half of all benign files, they receive low individual weights (1.0 to 2.0) so that no single pattern alone can accumulate sufficient score to cross the detection threshold. Behavioral rules achieve a false positive rate of 0.06% (3 out of 5,000) because benign scripts in isolation rarely produce the specific action combinations (external script injection, cookie gating, beacon exfiltration) that the behavioral rules target.

The two rule sets are complementary. Static rules provide broad recall (93.6%) at the cost of higher per-file false positives, while behavioral rules provide precise confirmation (0.06% FPR) with lower standalone recall (64.1%). The pipeline combines both scores at the cluster level, where static scores flag candidates broadly and behavioral scores confirm them precisely.

Setting every rule to equal weight, as reported in the released evaluation table, reduces static recall from 93.6% to 43.6% and collapses behavioral recall from 64.1% to 7.7%, because high-specificity indicators (e.g., `net_suspicious_domain` at 0% FP and 39.7% TP) are drowned out by frequently firing low-specificity rules. This confirms that the weight calibration reflects empirical specificity rather than arbitrary assignment.

H VirusTotal Detection Timeline

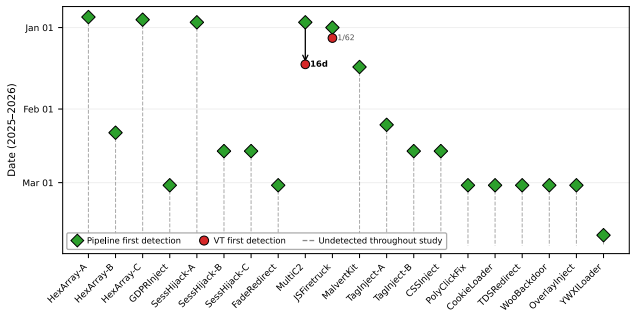


Figure 3: VirusTotal detection gap. Diamonds (◇): pipeline first detection; circles (○): VT first non-zero detection. Dashed lines: undetected at study end. Only MultiC2 received meaningful coverage (23/61 engines, 16-day delay).