ChatGPT

Kratos Deep Research Report on the WordPress Plugin Malware Detection Pipeline

Background and threat landscape

The heart of this project is a very practical observation: if a huge chunk of the web runs on a plugin-driven CMS, then "trusting plugins" becomes a security-critical decision—often made by stressed-out humans on a deadline. WordPress [1] is routinely measured as one of the most deployed CMS platforms on the public web; W3Techs [2] reports (as of Feb 17, 2026) that WordPress is used by **59.9% of websites with a known CMS**, and **42.6% of all websites** overall. [3]

That widespread usage matters because the plugin ecosystem is enormous and constantly evolving. WordPress.com's own market-share explainer notes **59,000+ free plugins** in the WordPress.org directory (and acknowledges the broader, harder-to-count commercial ecosystem). [4] Another recent ecosystem retrospective (from the WordPress Plugins Team) reports **12,713 plugin reviews in 2025**, emphasizing continued growth in submissions and review workload. [5]

Against that backdrop, your paper frames a specific supply-chain-shaped problem: "nulled" (pirated) paid plugins are distributed via multiple channels, and attackers can monetize the demand for discounted/free premium plugins by inserting malware into these redistributed packages. The paper's core claim is not that *all* nulled plugins are malicious, but that the ecosystem creates strong incentives for widespread malware seeding—and that this malware often hides "in plain sight," rather than relying on sophisticated obfuscation.

Research study overview and empirical findings

Your accompanying paper ("Hidden in Plain Sight: An Empirical Study of Malware in Nulled WordPress Plugins") studies three distribution channels in depth—web-based marketplaces, Telegram [6] channels, and torrents—and uses this exploration to motivate and validate Kratos as an automated analysis framework.

A few empirical results (as the paper presents them) define the stakes:

The study analyzes **4,271 plugins** total and flags **1,851 as malicious**, implying an overall malicious rate of roughly **46%** in the sampled nulled-plugin ecosystem.

The distribution channels differ sharply in both *volume* and *dominant malware behavior*. In the paper's dataset, web marketplaces contribute the most samples and the most malicious plugins; Telegram and torrents show lower malicious rates, but different (often more directly security-relevant) behaviors.

Within the web-marketplace subset, the paper highlights that “MPlugin” (ad-injection + self-hiding behavior) dominates detections for some marketplaces, suggesting a strong coupling between a distribution venue and a particular malware family/actor pattern.

A key methodological observation is that this is not the typical “run it in a sandbox and watch what it does” malware study. Instead, the paper argues that CMS/plugin ecosystems make full dynamic analysis costly and brittle (complex deployments, configuration needs, hosting constraints), and that static, signature-driven analysis can be more operationally realistic for this threat model—especially for technically novice site owners or those without server-level access.

One early “sanity check” result in the paper is also important for why Kratos exists at all: in the initial manual investigation, scans using VirusTotal [7] did not reliably flag the malicious behaviors the authors observed by reading the PHP code (with only occasional low-confidence detections like ad injection). That gap motivates a “purpose-built” detector for this malware niche.

Kratos pipeline design

Your paper’s architecture description and the repository’s implementation align on a concrete design philosophy: **focus on PHP malware in plugins/themes**, extract meaningful behaviors via a mix of **regex signatures** and **AST-derived semantic signatures**, and emit a **human-readable, “friendly” report** rather than just a binary verdict.

At a conceptual level, the pipeline has three phases (as described in the paper):

First, extraction/normalization: Kratos aims to avoid scanning irrelevant assets and concentrates on **PHP** as the primary malware carrier in this ecosystem.

Second, detection + classification: Kratos uses signatures derived from manual malware analysis; it combines syntactic indicators (regex/string patterns, sensitive APIs) with semantic ones (behavioral patterns derived from AST structure).

Third, result compilation: Kratos produces an output report describing suspicious behaviors and where they appear.

A design detail worth calling out—because it’s critical in plugin ecosystems—is metadata extraction. WordPress plugin loading depends on a standardized header block that declares fields like “Plugin Name,” “Version,” “Author,” etc. The WordPress Plugin Handbook explicitly documents this header convention (including the fact that “Plugin Name” is required). [8] Your paper leverages that fact operationally: if an attacker “evades” by removing the header entirely, the plugin won’t load normally—so Kratos can safely treat a missing header as a strong signal that the sample isn’t a standard WordPress plugin package.

The containerization choice is also foundational. The paper explains Kratos was bundled into a Docker image to create a stable baseline environment, isolate analyses, and make runs reproducible. The repository reinforces this goal through a dedicated Docker [9] build/run workflow and a “bridge directory” output strategy for container runs. [10]

Repository architecture and execution flow

The repository you linked (AtharavRH/kratos) describes itself as "a malware detection pipeline tailored for CMS plugin files" and explicitly positions itself as building on top of Jedi (a fork of YODA from the Cyber Forensics Innovation Lab [11] lineage). [12] The upstream "Mistrust Plugins You Must" work—published via USENIX Association [13] —describes YODA as a framework for detecting malicious plugins at scale and tracking origin economics, providing the conceptual ancestry for the Kratos-style pipeline approach. [14]

In this repo, the main entry point is `framework.py`, and its orchestration logic is straightforward but worth understanding deeply because most "how does Kratos behave?" questions reduce to this file:

It defines a global `mal_file_analysis_rules` list containing the active detection passes: API abuse, blackhat SEO, downloader, function construction, gated plugin logic, MPlugin regex detector, and spam injection. [15]

For each candidate file, Kratos reads the file content, generates an AST by invoking `ast_utils/generateAST.php`, and then runs each analysis rule's `reprocessFile()` method on the file object (passing both AST and raw text as inputs depending on the rule). [16]

To scale across plugin files, it uses `multiprocessing.Pool(cpu_count())` to run per-file detection in parallel. [15]

To identify which files to analyze, it walks the plugin directory and uses the `magic` library to get MIME types (rather than trusting extensions), then filters to PHP-ish MIME matches (and `style.css` for themes). [17] This aligns conceptually with the paper's "don't trust extensions; use file-type detection" rationale.

Reports are written only when malware is detected. If `total_mal_files_count > 0`, Kratos builds a plugin-level output structure via `process_outputs()` and writes a JSON report under `results/`. [17] This means "clean" plugins typically produce no output artifact unless the code is modified. [15]

The repository also supports a specific container mode interface: `BASE_PATH` is mandatory in container mode, and `BRIDGE_DIR` enables writing results into a host-mounted directory (hardcoded in the Dockerfile and described in the README). [10]

A secondary but practically important behavior is how the framework infers dataset provenance. `get_plugin()` infers `download_platform` and `download_source` from directory structure by walking up parent directories, which is consistent with a dataset layout like `<platform>/<source>/<plugin>/...`. [18] This matters because the output filenames embed `download_platform` and `download_source`, making offline analysis easier when running Kratos across large scraped corpora. [17]

Detection rules and signatures deep dive

This section ties the paper's "signature list" conceptually to the concrete implementation. The paper describes seven major signature families (API abuse, blackhat SEO, downloader, function construction, input

gating, MPlugin, spam injection). The repository implements these as a mix of Python "analysis rules" plus PHP AST parser scripts.

API and WordPress function abuse is implemented by `Analysis_API_Abuse`, which shells out to `analysis_rules/ast_parsers/api_abuse_parser.php` and then maps parser output into tags such as `DISABLE_ALL_PLUGINS`, `USER_ENUM`, `POST_INSERT`, `SPAM_DOWN`, `USER_INSERT`, `CHECK_FOR_GET`, `FAKE_FUNCTIONS`, and `USER_INFO_BASED_BACKDOOR`. [19] The Python side applies additional heuristics to reduce false positives, such as requiring ≥6 distinct post-insert-related function hits for `POST_INSERT`, ≥11 distinct function hits for `SPAM_DOWN`, and excluding a small allowlist of plugin names from `USER_INSERT` tagging. [20]

Blackhat SEO is implemented by `Analysis_Blackhat_SEO` + `blackhat_seo_parser.php`. The PHP parser looks for suspicious URLs, bot user-agent strings, and the presence of `json_decode`, and only reports "detected" when all relevant ingredients appear together. [21] The Python wrapper then normalizes extracted URLs/domains and checks them either against a built-in malicious URL list or by querying VirusTotal; depending on whether the VT response is conclusive, it emits either `SEO` or `MAYBE_SEO`. [22]

Downloader detection is one of the most behaviorally rich passes. On the PHP side (`downloader_parser.php`), the parser follows multiple patterns: GET-like calls feeding into `file_put_contents`, as well as `download_url`, `wp_remote_get`, `file_get_contents`, and `curl_setopt` flows, attempting to extract the URL argument even when it's assigned indirectly and partially constructed. [23] The Python wrapper (`Analysis_Downloader`) then vets extracted domains against a built-in list (`m_urls`), a "clean" list (`c_urls`), and VirusTotal; it also includes additional heuristics to suppress likely false positives (e.g., URL strings containing local placeholders or certain variable-construction artifacts) and emits `DOWNLOADER` vs `MAYBE_DOWNLOADER` depending on whether the URL reputation check is decisive. [24]

Function construction focuses on dynamic function-name behaviors (e.g., variable function calls) combined with taint-style reasoning. The Python side (`Analysis_Function_Construction`) calls `fc_parser.php` and then checks whether the parser reports (a) any constructed function names and (b) taint-analysis hits (via ProgPilot results) before tagging `FUNCTION_CONSTRUCTION`. [25] The dependency chain here is explicit in the repository's PHP tooling: `ast_utils/composer.json` requires both `nikic/php-parser` and ProgPilot [26] (`designsecurity/progpilot`). [27] The repository also includes a ProgPilot configuration file enabling PHP + multiple CMS/framework "modes," including WordPress. [28]

Gated-plugin detection looks for "conditional execution gates" that are inconsistent with benign plugin patterns—e.g., comparisons involving hashed strings or suspicious superglobal array keys. Concretely, `Analysis_Gated_Plugin` consumes `gated_plugin_parser.php` output and emits `GATED_PLUGIN` when parser-reported "plugin gates" are present. [29]

MPlugin detection is the most purely regex-driven signature in the codebase. Its Python analyzer looks for specific identifying strings (plugin name markers, suspicious settings registration, custom function names, and suspicious domain patterns) and emits tags like `MPLUGIN_HEADER`, `MPLUGIN_FUNCTIONS`, and `MPLUGIN_DOMAINS`. [30] This aligns with the paper's observation that some plugin malware families are "brazen" and stable enough that robust regex indicators emerge from manual analysis.

Spam injection detection (`Analysis_Spam` + `spam_parser.php`) focuses on content-fetch-and-inject patterns. The parser in particular follows "GET request return value ⇒ decoder (e.g., JSON decode)" chains, extracts URLs and query parameters where possible, and emits an output array that the Python wrapper converts into a `SPAM_INJECTION` tag when non-empty. [31]

A subtle but critical cross-cutting mechanism is URL reputation checking. Multiple analyzers in this repo call into `vt.py` (VirusTotal scanning), and the repository's logic implicitly assumes the constraints of the VirusTotal Public API (e.g., very low per-minute rate limits). The VirusTotal docs explicitly describe the Public API's constraints (including 4 requests/minute) and restrictions on multiple accounts to evade quotas. [32]

Output format and operational considerations

Kratos' "friendly report" is concretely expressed as a JSON structure assembled in `process_outputs()`. At the top level, the output includes plugin metadata fields (name, version, author/URIs), the base path, the inferred download platform/source, whether the artifact is a theme, file counts, and an overall analysis runtime. [18]

The most important forensic payload is `mal_file_info`, which maps each suspicious filepath to (a) `suspicious_tags` and (b) tag-specific `extracted_results` (for example: extracted URLs and VT verdicts, or the concrete code locations/functions that triggered an API-abuse heuristic). [33]

A minimal schema sketch (based directly on the code) looks like this:

```
{
  "plugin_name": "...",
  "base_path": "...",
  "download_platform": "...",
  "download_source": "...",
  "plugin_version": "...",
  "plugin_author": "...",
  "plugin_author_uri": "...",
  "plugin_plugin_uri": "...",
  "is_theme": false,
  "theme_name": "...",
  "num_files": 123,
  "tot_mal_files": 4,
  "mal_file_info": {
    "/abs/path/file.php": {
      "suspicious_tags": ["DOWNLOADER", "USER_ENUM"],
      "extracted_results": { "DOWNLOADER": {...}, "USER_ENUM": [...] }
    }
  },
  "time": 2.345
}
```

[18]

Operationally, there are a few behaviors and constraints that strongly shape how Kratos is used in practice:

It is Linux-first. The README explicitly warns that Windows support involves “quirks” around PHP runtime behavior and phar execution, and it notes known-good testing on Python 3.8.10. [34]

Container mode is the intended “batch research” path: the README documents `BASE_PATH` and `BRIDGE_DIR`, and the Dockerfile bakes in a bridge directory default (`/usr/src/bridge/`). [10]

The repo includes convenience scripts for dataset hygiene and batch runs: recursive ZIP extraction (`extractor.py`), recursive RAR extraction (`extractor_rar.py`), and a PowerShell “run all” driver that runs the Kratos container repeatedly over a dataset folder, logging outputs per run. [35]

The PHP tooling dependencies are deliberately pinned by Composer. In particular, `ast_utils/composer.json` declares `nikic/php-parser` and ProgPilot, implying that AST parsing and taint-style reasoning are meant to be reproducible from a container build. [36]

Finally, there is a security hygiene point that matters if you plan to keep this repository public: the `vt.py` implementation contains embedded VirusTotal API keys in source, which is generally unsafe even if the keys are rate-limited. [37]

Public blog draft suitable for Medium or LinkedIn

Title: Kratos vs. “Free Premium Plugins”: A Tale of WordPress, Malware, and Questionable Life Choices [38]

If you’ve ever built a WordPress site, you know the feeling: you need exactly one plugin to finish the job. Then you install three “just in case.” Then you add an SEO plugin because a blog post from 2014 told you to. Then you add a caching plugin because performance is love. Performance is life. [39]

And then—because you are a perfectly rational adult who makes excellent choices—you see it:
“Premium plugin, nulled, FREE download.”
Free. The most expensive word in cybersecurity.

Why we built Kratos

Here’s the uncomfortable reality: WordPress is *everywhere*. As of Feb 2026, W3Techs reports WordPress is used by about **42.6% of all websites** (and about **59.9%** of sites where the CMS is known). That’s not a niche. That’s part of the web’s load-bearing structure. [3]

WordPress is powerful largely because plugins are powerful. But plugin power comes with plugin privilege: plugins run code on your server. So when plugin distribution gets sketchy—especially the “nulled plugin” ecosystem—attackers don’t need Hollywood-level tricks. They just need someone to install the thing.

In our research, we looked at three major “how people actually get nulled plugins” channels: web-based marketplaces, Telegram channels, and torrents. We then asked a simple question: **How often are these plugins malicious, and what do they do?**

Spoiler (the kind you don’t want): in our dataset of **4,271 plugins**, we flagged **1,851** as malicious. That’s roughly **46%**. A coin flip, except the “tails” side can steal your admin account and install more malware.

The twist: antivirus wasn’t enough

When we manually inspected early samples, we also ran them through VirusTotal. The results were… underwhelming. Many plugins that looked clearly malicious in code review weren’t strongly flagged by generic scanners. This isn’t a dunk on antivirus—it’s a sign that the CMS/plugin malware niche needs detectors that understand *what these scripts are trying to do*, not just what they “look like” in byte patterns.

So we built **Kratos**: an automated malware detection pipeline tailored specifically for CMS plugin files—especially PHP-heavy WordPress plugins/themes. [40]

Kratos, in plain English

Kratos is opinionated:

It mostly targets **PHP**, because that’s where the plugin malware action is.

It doesn’t trust file extensions. It uses MIME type identification (`magic`) to decide what’s actually PHP-like content. [41]

It pulls plugin/theme metadata from the standard headers (because WordPress requires them). That’s how it produces reports that say something more useful than “file.php is suspicious.” [42]

It uses two kinds of signatures: * **Regex signatures** (fast, good for “malware families that can’t help but introduce themselves”) * **AST-based behavioral signatures** (more semantic: “this code downloads a payload and writes it to disk,” not just “this code contains the string ‘download’”).

And finally, it emits a JSON report: which files were suspicious, which rules triggered, and what evidence was extracted (URLs, function calls, suspicious gate conditions, etc.). [18]

What Kratos catches

Kratos was built around the malware behaviors we kept seeing in the wild. Here are a few examples (edited for readability, not for drama):

API abuse and admin shenanigans
We look for WordPress/PHP API patterns that show up in malicious plugins: disabling other plugins, enumerating admin users, injecting posts, creating privileged accounts, and other “nothing to see here” behavior. [43]

Blackhat SEO
Some malware behaves differently depending on who's visiting. If you're a normal user? Maybe you see a normal site. If you're a search-engine crawler? Surprise—spam links and redirects. Kratos detects this by looking for bot user agents + suspicious URL patterns at the AST level, and then it can reputation-check domains. [44]

Downloaders
A classic: pull something from a remote URL and drop it onto the server. The downloader rules follow multiple flows (GET→write-to-disk, `download_url`, cURL sessions). If a URL looks suspect, Kratos can query VirusTotal (within public API limits) or compare against a known-malicious list. [45]

"Gated" backdoors
Sometimes malicious code sits idle unless a magic condition is met—like a hardcoded password in `$_GET` compared against a hash-looking string. Kratos detects these suspicious equality checks in the AST. [46]

Spam injection
Fetch content from elsewhere, decode it, inject it into output—often with telltale parameter patterns. Kratos tracks these GET→decode flows in the AST and extracts the URLs/params where possible. [47]

Why Docker was non-negotiable

We bundled Kratos as a container because large-scale malware analysis should not feel like speedrunning "dependency hell." Containerization gives us a consistent baseline, isolates runs, and makes the pipeline reproducible. Also, if a plugin package contains something truly nasty, we want that blast radius as small as possible.

What we learned (beyond "don't install shady plugins")

The big takeaway from the study is that the ecosystem isn't uniform. Different distribution channels tend to correlate with different malware "styles" and motivations—ranging from ad-injection-heavy markets to more directly dangerous downloader behavior elsewhere. So it's not just "is it malicious?" but also "what kind of malicious are we talking about?"

And yes, there's still a lot to do: broader datasets, better actor attribution, deeper data-flow analysis for exfiltration patterns, and understanding how compromised servers get reused downstream.

---

[1] [31] https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_spam.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_spam.py

[2] [15] [16] [17] [41] [42] https://raw.githubusercontent.com/AtharavRH/kratos/master/framework.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/framework.py

[3] https://w3techs.com/technologies/comparison/cm-wordpress?utm_source=chatgpt.com
https://w3techs.com/technologies/comparison/cm-wordpress?utm_source=chatgpt.com

[4] [11] [39] https://wordpress.com/blog/2025/04/17/wordpress-market-share/
https://wordpress.com/blog/2025/04/17/wordpress-market-share/

5 9 https://make.wordpress.org/plugins/2026/01/07/a-year-in-the-plugins-team-2025/
https://make.wordpress.org/plugins/2026/01/07/a-year-in-the-plugins-team-2025/

6 35 https://raw.githubusercontent.com/AtharavRH/kratos/master/scripts/extractor.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/scripts/extractor.py

7 12 26 40 https://github.com/AtharavRH/kratos
https://github.com/AtharavRH/kratos

8 https://developer.wordpress.org/plugins/plugin-basics/header-requirements/
https://developer.wordpress.org/plugins/plugin-basics/header-requirements/

10 34 https://raw.githubusercontent.com/AtharavRH/kratos/master/README.md
https://raw.githubusercontent.com/AtharavRH/kratos/master/README.md

13 18 33 https://raw.githubusercontent.com/AtharavRH/kratos/master/jedi_utils.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/jedi_utils.py

14 https://www.usenix.org/conference/usenixsecurity22/presentation/kasturi
https://www.usenix.org/conference/usenixsecurity22/presentation/kasturi

19 20 43 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_api_abuse.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_api_abuse.py

21 22 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_blackhat_seo.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_blackhat_seo.py

23 45 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/downloader_parser.php
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/downloader_parser.php

24 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_downloader.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_downloader.py

25 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_function_construction.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_function_construction.py

27 36 raw.githubusercontent.com
https://raw.githubusercontent.com/AtharavRH/kratos/master/ast_utils/composer.json

28 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/progpilot_config/config.yml
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/progpilot_config/config.yml

29 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_gated_plugin.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_gated_plugin.py

30 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_mplugin.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/analysis_mplugin.py

32 https://docs.virustotal.com/reference/public-vs-premium-api
https://docs.virustotal.com/reference/public-vs-premium-api

37 https://raw.githubusercontent.com/AtharavRH/kratos/master/vt.py
https://raw.githubusercontent.com/AtharavRH/kratos/master/vt.py

38 https://w3techs.com/technologies/details/cm-wordpress
https://w3techs.com/technologies/details/cm-wordpress

44 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/blackhat_seo_parser.php
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/blackhat_seo_parser.php

46 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/gated_plugin_parser.php
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/gated_plugin_parser.php

47 https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/spam_parser.php
https://raw.githubusercontent.com/AtharavRH/kratos/master/analysis_rules/ast_parsers/spam_parser.php