

Forum:[Hack Forums](#)

Thread: [V-Core: Advanced Automation & Recon Framework | Python 3.12 / PHP 8.3](#)

redo - April 3, 2026, 1:38 am

V A T A N - C O R E // ADVANCED AUTOMATION & RECON FRAMEWORK

??

Author: redo | Build: v1.0.0-RELEASE | Stack: Python 3.12 / PHP 8.3 / MySQL 8

??

[/HR]

[01] — EXECUTIVE SUMMARY

Let's be direct: the automation landscape is broken.

What passes for "frameworks" in most circles are glorified wrapper scripts — brittle, synchronous, unmaintained, and architecturally illiterate. They collapse under concurrency. They have no abstraction layers. Written by people who learned to script, never to engineer. You've seen them: a `Code requests.get()` loop dressed up with `argparse`, hardcoded delays as a stand-in for rate limiting.

V-Core is not that. It is not a script. It is not a tool. It is a properly engineered framework.

Designed from first principles with one mandate: zero compromise between performance, modularity, and operational control. An asynchronous, multi-threaded Python engine for data extraction, OSINT pipeline orchestration, and RESTful API lifecycle management — governed through a real-time PHP/SQL control panel with sub-second response latency.

If your stack can't handle 500 concurrent tasks without rate collisions or thread starvation — you are already behind. This release changes that baseline.

[/HR]

[02] — ADVANCED ARCHITECTURE

Layered, decoupled architecture. Every component independently testable, replaceable, and horizontally scalable. Nothing monolithic. Nothing hardwired.

» Core Engine — Python 3.12 Async-first design via `Code asyncio` + `Code uvloop` — 2–4x throughput over the default CPython event loop

Hybrid thread model: CPU-bound tasks -> `ThreadPoolExecutor` workers; I/O tasks -> non-blocking async loop. The event loop is never blocked.

Task Scheduler: Priority queue with configurable concurrency caps, exponential backoff retries, and dead-letter queuing for failed jobs pending UCP review

Session pooling: Persistent `Code aiohttp.ClientSession` management with per-domain rate limiting and connection reuse across all task workers

» Extension Layer — Lua Scripting Engine Embedded LuaJIT runtime via Code lupa — hot-loadable, sandboxed modules. Drop a new Code .lua file, reload from UCP. Zero engine downtime.

Typed bridge API: Code vcore.queue, Code vcore.http, Code vcore.store namespaces. Extension manifests declare permissions — nothing accesses beyond its defined scope.

» API Integration Layer RESTful API client factory with OpenAPI schema validation, token lifecycle management, and automatic credential injection per configured target

Middleware pipeline: request -> auth-inject -> rate-limiter -> retry-handler -> response-parser -> store

Inbound webhook listener for push-based OSINT triggers — pipeline reacts in real-time, not on a polling interval

» Backend — PHP 8.3 / MySQL 8 PDO + prepared statements throughout — not in most places. Everywhere. Zero raw query construction in this codebase.

JWT auth with RS256 asymmetric signing, short-lived access tokens, and refresh token rotation on every use

Encrypted credential vault: AES-256-GCM at rest, key derived via PBKDF2 from server secret + user passphrase

PHP<->Python bridge over UNIX domain socket — no TCP port exposure, no loopback attack surface

[/HR]

[03] — THE UCP (USER CONTROL PANEL)

The UCP is not a dashboard bolted onto a Python script as an afterthought. It is the command surface. Every operational parameter — task queues, credentials, rate limits, module configs, live logs — from a single low-latency interface, zero cloud dependency.

» Design Dark-themed, zero-bloat UI — Tailwind CSS only. No React, no Vue, no 4MB JS bundles. Fully rendered under 150ms on LAN.

Real-time telemetry via Server-Sent Events — no polling, no WebSocket overhead for read-only live streams

Live log console with ANSI-to-HTML rendering, filterable by module, severity, and task ID

» Modules Task Manager: Create, schedule, pause, kill, re-queue. Live status. Full history with retry counts and last-error display.

Module Registry: Install, toggle, and configure Lua extensions without touching the engine process

API Vault: Encrypted key storage with per-module access scoping — a module only sees credentials explicitly granted to it

Output Explorer: Browse, filter, export, and tag extraction results in JSON, CSV, and raw formats

Audit Log: Immutable, append-only. Every UCP operation recorded with timestamp, operator ID, and affected resource.

[/HR]

[04] — CODE TEASER

Excerpt — vcore/engine/executor.py — Async/Threaded Hybrid Dispatcher

```
Code from __future__ import annotations
import asyncio, logging
```

```
from collections.abc import Awaitable, Callable
from concurrent.futures import ThreadPoolExecutor
from dataclasses import dataclass, field
from enum import Enum, auto
from typing import Any
import uvloop
```

```
logger = logging.getLogger("vcore.executor")
```

```
class TaskState(Enum):
    PENDING = auto(); RUNNING = auto()
    COMPLETED = auto(); FAILED = auto(); DEAD = auto()
```

```
@dataclass(slots=True)
```

```
class VTask:
```

```
    task_id: str
    coro_fn: Callable[..., Awaitable[Any]]
    args: tuple[Any, ...] = field(default_factory=tuple)
    kwargs: dict[str, Any] = field(default_factory=dict)
    priority: int = 5
    max_retries: int = 3
    _retries: int = field(default=0, init=False, repr=False)
    state: TaskState = field(default=TaskState.PENDING, init=False)
```

```
    def __lt__(self, other: "VTask") -> bool:
        return self.priority < other.priority
```

```
class VCoreExecutor:
```

```
    def __init__(self, max_concurrent: int = 200, thread_workers: int = 16) -> None:
        self._semaphore = asyncio.Semaphore(max_concurrent)
        self._thread_pool = ThreadPoolExecutor(max_workers=thread_workers,
                                              thread_name_prefix="vcore-worker")
        self._queue: asyncio.PriorityQueue[tuple[int, VTask]] = asyncio.PriorityQueue()
        self._dead_letter: list[VTask] = []
```

```
    async def submit(self, task: VTask) -> None:
        await self._queue.put((task.priority, task))
        logger.debug("[QUEUE] %s enqueued (priority=%d)", task.task_id, task.priority)
```

```
    async def run_until_empty(self) -> None:
        workers = [asyncio.create_task(self._worker(i)) for i in range(64)]
        await self._queue.join()
        for w in workers: w.cancel()
        if self._dead_letter:
            logger.warning("[DLQ] %d task(s) exhausted all retries.", len(self._dead_letter))
```

```
    async def _worker(self, wid: int) -> None:
        while True:
            _, task = await self._queue.get()
            async with self._semaphore:
                await self._dispatch(task, wid)
            self._queue.task_done()
```


