

A Product-Grade Threat for the AI Service Era

✕ blog.xlab.qianxin.com/nadmesh-botnet-analysis-a-product-grade-threat-for-the-ai-service-era-en

Xlab

July 17, 2026



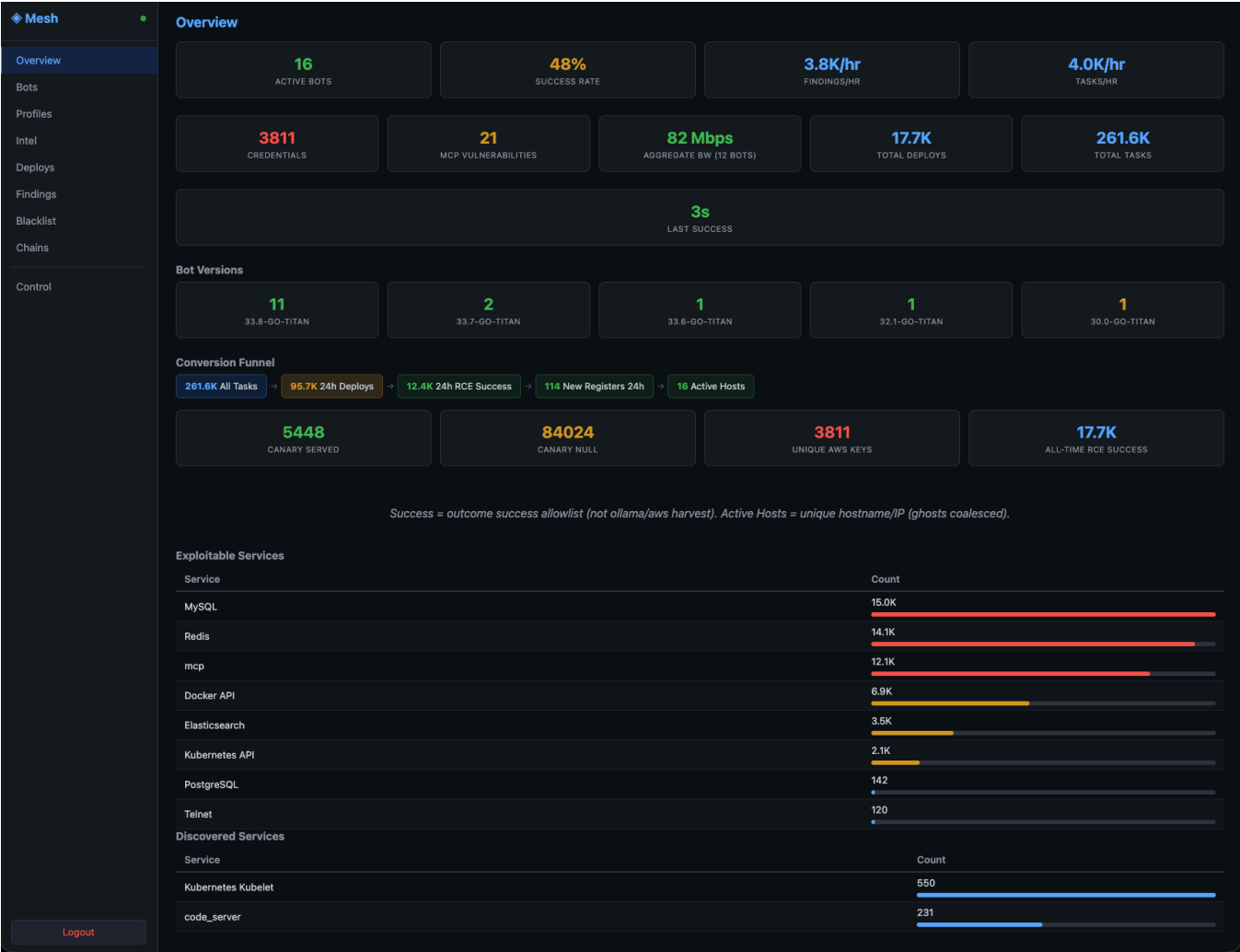
Overview

In early July 2026 we observed a Go-based botnet pushing bot samples onto the internet at scale. It folds scanning, exploitation, and credential/AI-service intelligence harvesting into a single autonomous platform. Because its controller calls itself `n4d mesh controller` in the source, we named it **NadMesh**.

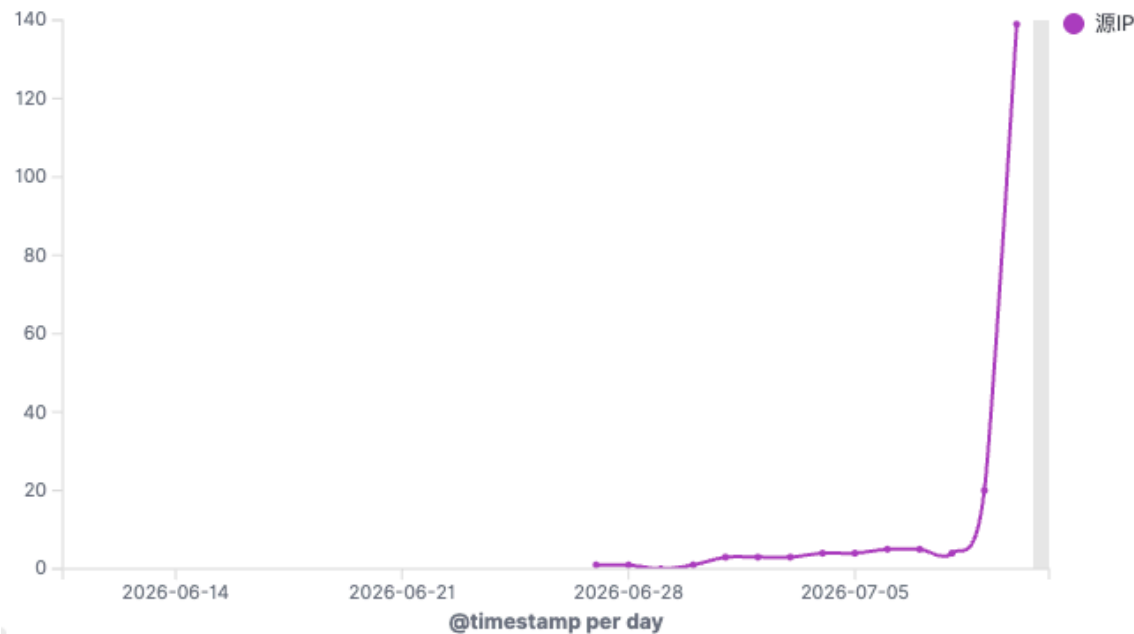
NadMesh is not a one-off worm outbreak. It is a continuously iterated, autonomous botnet aimed squarely at AI infrastructure and the MCP ecosystem. What sets it apart from traditional worms:

- **Autonomous scanning engine:** 90+ built-in cloud provider address ranges, spreading unattended
- **20+ exploitation vectors:** RCE across Redis, Docker, MCP, Kubernetes, and more
- **Targeted AI service harvesting:** collects ComfyUI, Ollama, and similar AI service IPs via Shodan and grants them the highest scan priority
- **Productized operations:** built-in web panel, conversion funnel statistics, and canary updates make the whole operation observable
- **Redundant persistence:** SSH public-key backdoor + agent process + cron watchdog make single-point removal ineffective
- **Polymorphic builds:** Garble obfuscation + UPX packing + random padding give every agent a unique hash

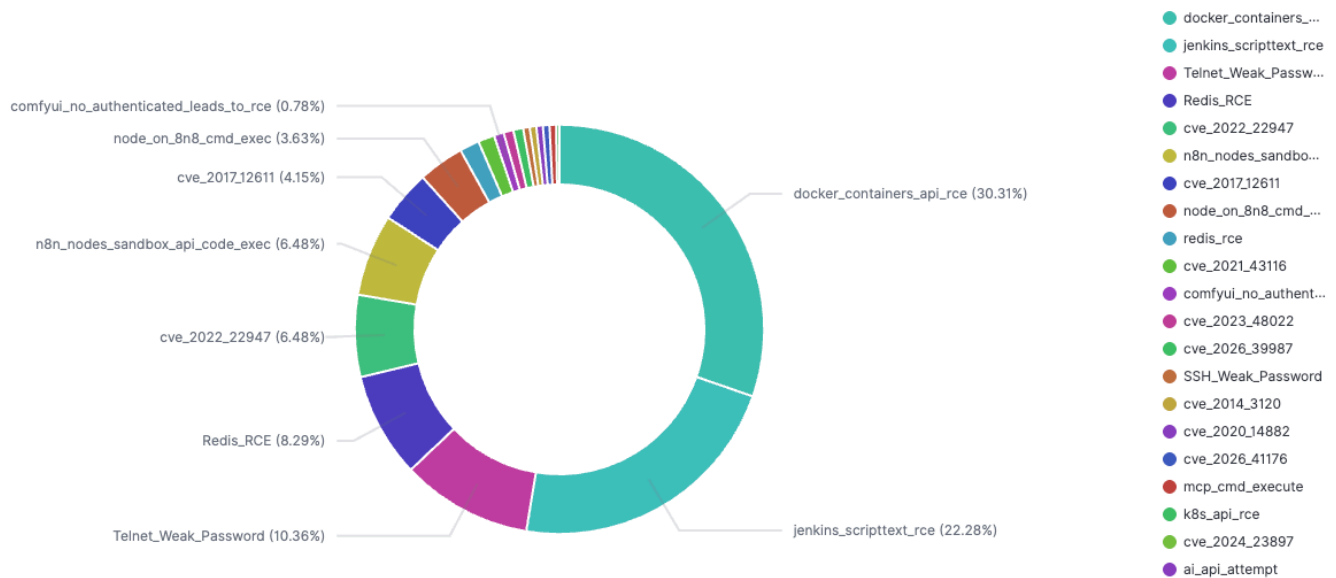
Taken together, NadMesh is not a script kiddie's mashup. It is industrial-grade malware with clear commercial intent and an eye on return on investment — a "long-term evolving" botnet that looks nothing like a classic worm.



NadMesh is still in an early stage. Infection trend:



Distribution of the NadMesh exploits we observed:



The Full NadMesh Kill Chain

Detailed analysis of the samples shows NadMesh to be a closed-loop, autonomous attack platform: the attacker's VPS is the hub, and the target network is the execution surface. The operation is heavily productized. The kill chain breaks into five cooperating stages: **intel** → **control** → **supply** → **build** → **delivery**.

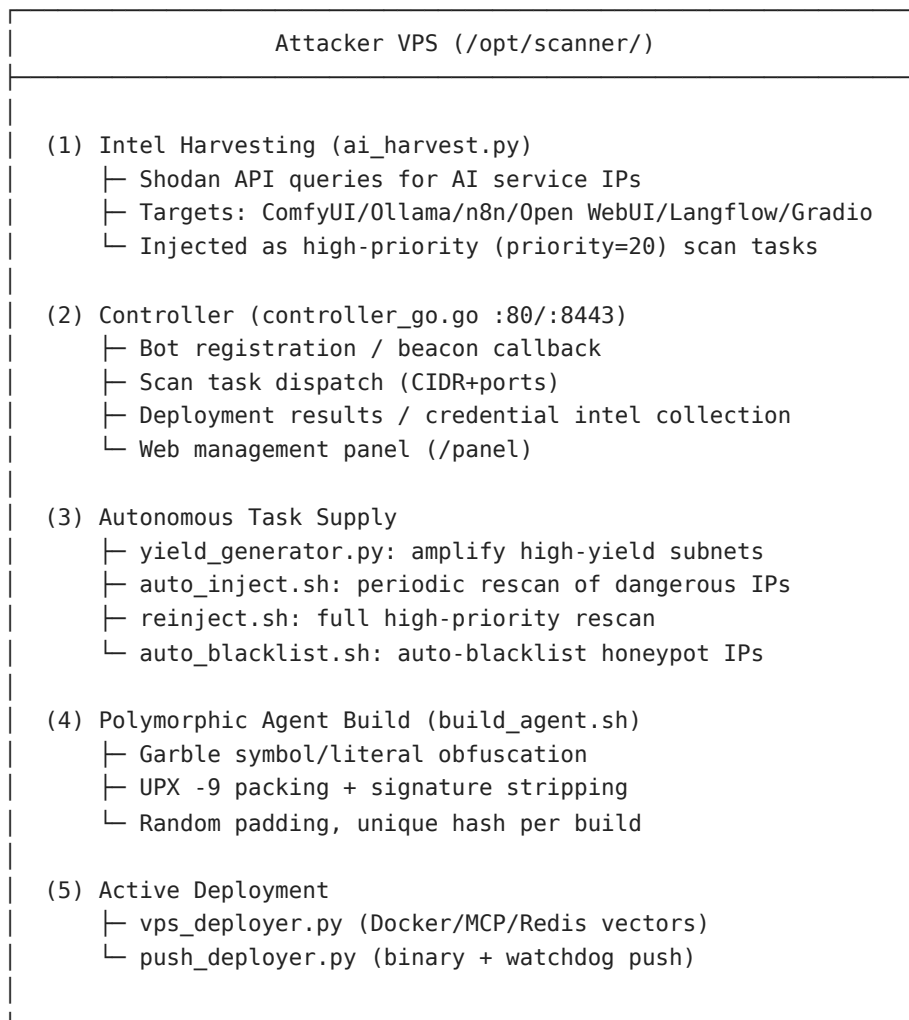
Intel is handled by `ai_harvest.py`, which uses the Shodan API to hunt for ComfyUI, Ollama, n8n, Open WebUI, Langflow, and Gradio assets, then injects them into the scan queue at the highest priority (`priority=20`). This is the most direct evidence that the campaign deliberately targets AI infrastructure.

Control is `controller.go`, listening on ports 80 and 8443. It handles bot registration and beacon callbacks, dispatches CIDR + port scan tasks, collects deployment results and credential intelligence, and exposes the `/panel` dashboard for observable management of the whole botnet.

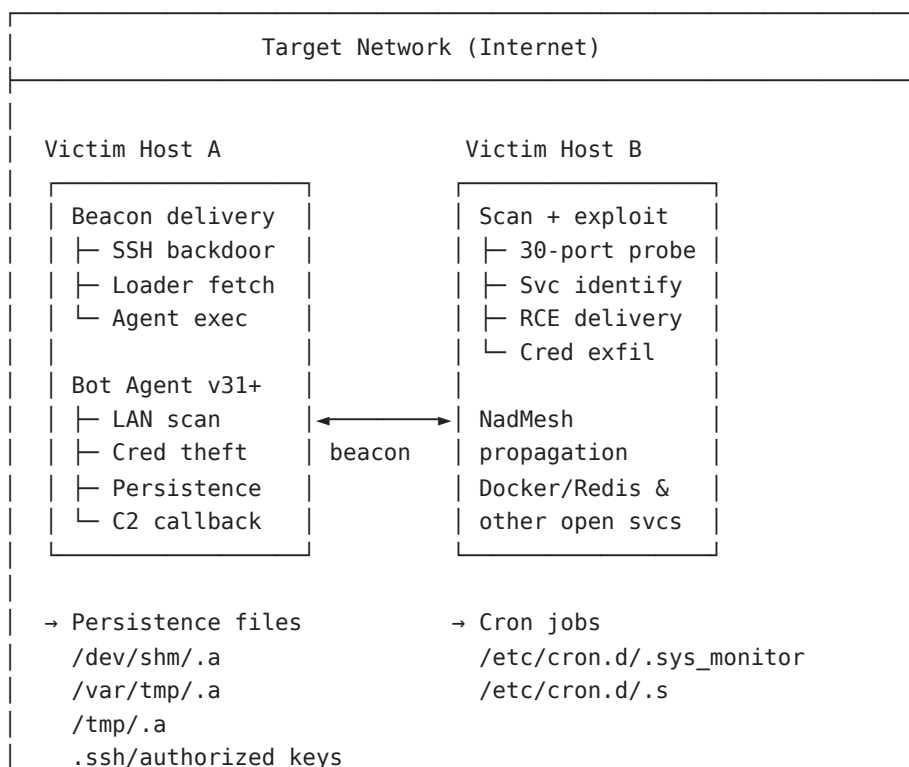
Supply comes from four scripts forming an autonomous task-feed loop: `yield_generator.py` amplifies high-yield subnets, `auto_inject.sh` periodically rescans dangerous IPs, `reinject.sh` performs full high-priority rescans, and `auto_blacklist.sh` steers clear of honeypot IPs. The loop lets "scan → exploit → spread" amplify itself with no human in the loop.

Build and delivery: `build_agent.sh` uses a polymorphic strategy — Garble symbol/literal obfuscation, UPX-9 packing, and random padding — so every agent carries a distinct hash and defeats signature-based detection. Delivery is then handled by `vps_deployer.py` (Docker/MCP/Redis vectors) and `push_deployer.py` (binary + watchdog push).

On the victim, the bot agent establishes persistence along three independent paths: an SSH public-key backdoor (`.ssh/authorized_keys`), persistence files in multiple locations (`/dev/shm/.a`, `/var/tmp/.a`, `/tmp/.a`), and hidden cron watchdogs (`/etc/cron.d/.sys_monitor`, `/etc/cron.d/.s`). Remove any one and the others bring it back. The agent simultaneously performs 30-port probing, service identification, 20+ vector RCE delivery.



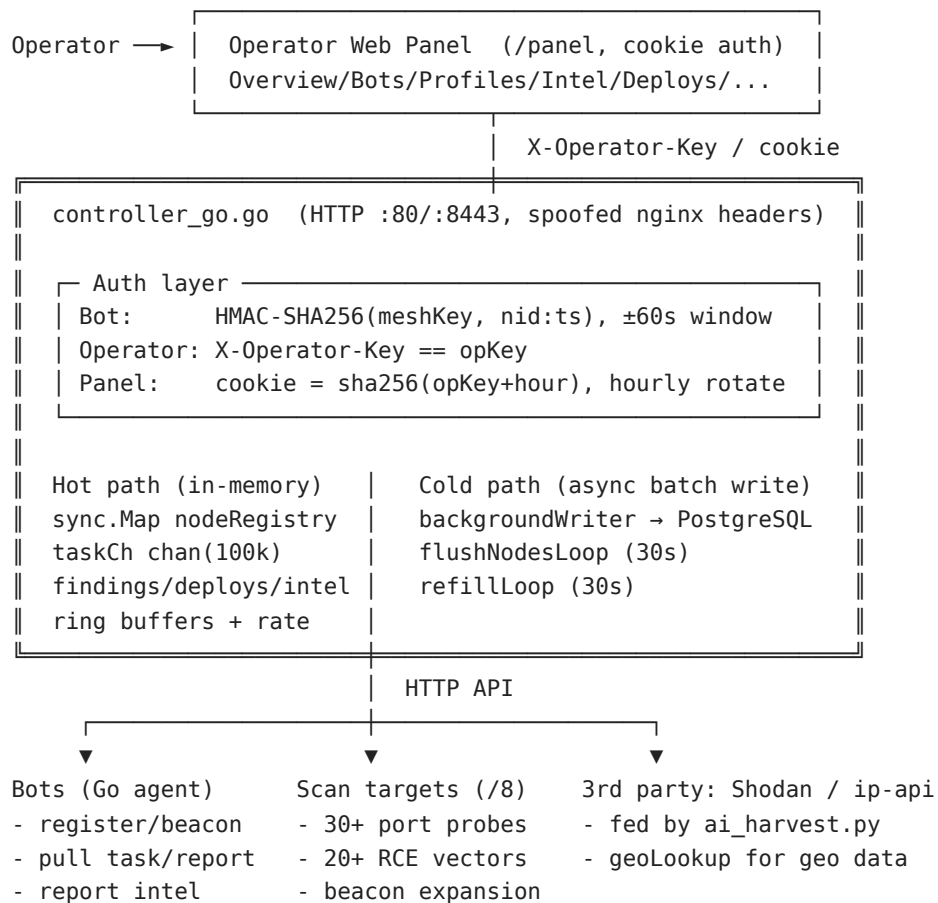
▼ HTTP /api/*



NadMesh Controller Analysis

The NadMesh controller ships in two versions, Go and Python, with essentially identical functionality. This article analyzes the Go version, `controller_go.go`, in which the author refers to the software as "n4d mesh controller".

Architecture



The author states their design philosophy in a comment:

In-memory hot path: `sync.Map` + channels + ring buffers. DB only for background persistence.

In other words: reads never touch the database, and writes are batched asynchronously to storage. That trade-off is what lets a single machine absorb high-concurrency bot traffic.

Authentication

The controller implements three independent authentication schemes.

Bot authentication (`verifyBotAuth`) uses HMAC: the header `X-Mesh-Auth: <node_id>:<hex_hmac>` carries `HMAC-SHA256(meshKey, "<node_id>:<unix_ts>")`. The server iterates 121 times over a ± 60

second window around the current time and compares with `hmac.Equal` in constant time to resist timing side channels.

Operator authentication (`verifyOperatorAuth`) is the crudest of the three: it merely checks the `X-Operator-Key` header against `opKey` in plaintext.

Panel cookie authentication is comparatively complete. The login password is `opKey`, compared with `subtle.ConstantTimeCompare`. A successful login issues the cookie `n4d_panel = sha256(opKey + "YYYY-MM-DD-HH")`, which rotates hourly; validation accepts both the current and previous hour window to avoid boundary failures. The cookie carries `HttpOnly` and `SameSite=Lax`, adds `Secure` over HTTPS, and has `MaxAge 86400`. Login is rate limited to 5 attempts per IP per 10 minutes, returning 429 beyond that.

Host profiles reported by bots are decrypted with `xorDecrypt` — base64 decode followed by a repeating XOR against `meshKey`. The strength is barely better than nothing.

Authentication is enforced through layered middleware:

- `openEndpoints`: registration, beacon, binary download, and update checks require no auth (beacon callbacks use a `?k=` token, or nothing at all)
- `operatorEndpoints`: GET on `intel/chains/stats/conversion` requires operator auth
- All other `/api/` paths require bot HMAC auth
- `/api/result/` is deliberately open so bots can submit scan results

HTTP Endpoint Inventory

Path	Method/Auth	Function
<code>/api/status</code>	open	Heartbeat; returns <code>binaryToken</code> (version fingerprint)
<code>/api/register</code>	open	Bot registration/check-in, node upsert, async geolocation
<code>/api/beacon</code>	open	Serves the initial infection shell script (SSH backdoor + loader download)
<code>/api/agent/binary(.gz)</code>	open	Serves the agent binary (per architecture)
<code>/api/agent/loader</code>	open	Serves the 722KB lightweight loader (which pulls full in background)
<code>/api/agent/full / /minimal</code>	open	Serves the complete agent
<code>/api/agent/download</code>	token(<code>?k=</code>)	Token-validated download
<code>/cdn/<binaryToken>/assets/img.bin</code>	open	Binary distribution disguised as a static asset
<code>/api/bot_update.py</code>	open	Serves the Python update script

Path	Method/Auth	Function
/api/update/check	token	Check for stable release updates
/api/update/canary	token/bot	Canary update (A/B rollout)
/api/update/health /promote	—	Update health check/promotion (stub)
/api/client/update	open	Returns latest_update.json directly
/api/endpoints	open	Returns the C2 endpoint list (incl. CDN fallback)
/api/peers	bot	Returns the peer list
/api/task/<nid>	bot	Bot pulls a scan task (CIDR+ports)
/api/result/<nid>	open	Bot reports scan findings
/api/deployed	bot	Bot reports exploitation/deployment results
/api/intel	GET operator / POST bot or operator	Intel submission and query (credentials/AI/MCP)
/api/chains /chains/refresh /chains/result/	bot/operator	Multi-step exploitation chains
/api/conversion	operator	24h deployment conversion funnel
/api/tasks/add	bot (operator use in practice)	Manual scan task injection
/api/seeds	bot	Bulk ip:port seed injection
/api/health	open	Runtime metrics (goroutines/memory/DB)
/panel/login /logout	—	Panel login/logout
/panel/api/*	cookie	Panel backend API (overview/bots/profiles/intel/deloys/findings/blacklist)
/panel/	cookie	Panel SPA

Autonomous Scanning and Task Scheduling

Scan Port Set (30 ports)

80,443,3000,5000,8000,8080,8443,8888,9000,9999,6443,10250,9200,22,23,
8088,2718,8090,10000,2379,8848,8265,8188,5678,11434,7860,5432,3306,2375,2376,6379

Services covered:

- Web: 80/443/3000/5000/8000/8080
- Kubernetes: 6443 (API) / 10250 (kubelet) / 2379 (etcd)
- Databases: 5432 (PostgreSQL) / 3306 (MySQL) / 6379 (Redis)
- Containers: 2375/2376 (Docker API)
- Monitoring: 9200 (Elasticsearch)
- **AI services: 8188 (ComfyUI) / 11434 (Ollama) / 5678 (n8n) / 7860 (Gradio) ← priority targets**
- Management: 10000 (Webmin) / 8090, etc.

Task Supply (Adaptive Feedback)

yield_generator.py — amplifying high-yield subnets

```
# Runs every 5 minutes:
# 1. Query results from the last 24h where dangerous != '[]'
# 2. Aggregate by /16 prefix, take the top 50 highest-yield subnets
# 3. Randomly generate 2000 /24 tasks from those subnets
# 4. Inject at priority=10 (above random, below dangerous IPs)
```

The effect is straightforward: the more results a subnet yields, the more densely it gets rescanned — a **positive feedback loop**.

auto_inject.sh — periodic rescan of dangerous IPs

```
# Runs every 15 minutes:
# 1. IPs from the last 24h with dangerous != '[]' are rescanned as /32 at high priority (priority=20)
# 2. AI service ports first: 8188/11434/7860/5678
# 3. Also generates 500 random /24s from 130 curated cloud provider /16 prefixes (priority=5)
```

reinject.sh — full high-priority rescan

```
INSERT INTO tasks (cidr, ports, priority=50)
SELECT DISTINCT ip||'/32' FROM results
WHERE dangerous != '[]'
  AND created_at > now()-604800    -- 7 days
  AND ip NOT IN honeypot_blacklist;
```

Stacked together, the three scripts form a clear priority ladder:

Priority	Meaning
50 (highest)	Full rescan of confirmed dangerous IPs
20 (high)	Dangerous findings within 24h; AI services harvested via Shodan
10 (medium)	High-yield subnet amplification
5 (low)	Newly added random subnets
0 (lowest)	/24 generated on the fly when a bot finds no task to pull

Automatic Honeypot Blacklisting

```
-- auto_blacklist.sh runs hourly
INSERT INTO honeypot_blacklist (ip, reason='auto: 10+ deploys')
SELECT target_ip FROM deploys
WHERE status NOT LIKE 'fail_%'
GROUP BY target_ip HAVING count(*) >= 10;
```

The logic: if 10 or more deployments against the same IP still produce no result, treat it as a honeypot and blacklist it automatically. This tells us the author is aware of security researchers and has actively built in evasion.

Agent Capabilities and Exploitation Vectors

Scanning and Identification (Bot Side)

The bot agent's scan workflow:

```
Bot —GET /api/task/<nid>—> Controller
      ↑
      |
  Receive task: CIDR + ports
  └─ If no task available, generate a random /24 on the spot (fallback)
```

```
Bot —scan CIDR/ports—> discover targets
  └─ Open port check
  └─ Service identification (banner grab)
  └─ Vulnerability detection (version fingerprint)
  └─ Danger flagging (dangerous=true)
```

```
Bot —POST /api/result—> Controller (report findings)
```

The fallback is worth noting: even if the controller's task queue runs dry, bots generate random /24s and keep scanning rather than sitting idle.

The 20+ RCE Vectors

Exploitation methods supported by the controller, in priority order:

Service	Attack vector	Risk	Notes
MCP	JSON-RPC tools/call → execute_command	High	Core threat of the AI era
Kubernetes	API pod creation + hostPath mount	High	Cluster escape
Docker API	/containers/create + host network + always-restart	High	Container escape
Redis	CONFIG SET dir + dbfilename + SAVE	High	Commonly unauthenticated
Elasticsearch	Script RCE (_search)	High	Widely exposed
SSH/Telnet	Weak credential brute force	Medium	Deploys backdoor
Jenkins	Script Console RCE	Medium	Weak admin passwords
Spring Cloud Gateway	Expression injection	Medium	Version-dependent
code-server	Command execution via API	Medium	Exposed without auth
Airflow/Superset/XXL-Job	DAG/Admin API RCE	Medium	Weak auth / default creds
WebLogic	Deserialization RCE	Medium	CVE-2016-0638 and others

Deployment Result Classification

The status reported by agents is bucketed into four classes by the controller. This classification is what feeds the conversion funnel on the panel:

```
status: "docker_push"    → class: "success"    (Docker RCE succeeded)
status: "mcp_sql_rce"    → class: "success"    (MCP SQL execution)
status: "redis_save"     → class: "success"    (Redis cron persistence)
status: "ssh_deploy"     → class: "success"    (SSH backdoor deployed)
status: "beacon_start"   → class: "exploit_sent" (Beacon delivered)
status: "timeout"        → class: "failed"      (No response)
status: "hw_suspect"     → class: "intel"      (Suspected honeypot, not stored)
status: "os_profile"     → class: "intel"      (Profile intelligence)
```

Credential and AI Intelligence Harvesting

Intel Fields (Reported by Agent)

Intelligence the agent collects immediately after a successful compromise:

```
{
  "node_id": "bot_id_hash",
  "metadata": {
    "profile": {
      "hostname": "prod-server-01",
      "os": "linux",
      "arch": "x86_64",
      "kernel": "5.10.0",
      "env_creds": ["AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE"],
      "services": ["redis", "docker", "k8s"],
      "internal_ranges": ["10.0.0.0/8", "172.16.0.0/12"],
      "k8s_sa": "ServiceAccount token with cluster-admin...",
      "docker_host": true,
      "in_container": false
    },
    "aws_creds": ["AKIA..."],
    "bedrock_creds": [...],
    "ssh_sessions": [...]
  },
  "files": {
    "aws_config": "~/.aws/config",
    "env_file": ".env",
    "docker_config": "~/.docker/config.json"
  },
  "docker": true,
  "ai_models": [
    "ollama://llama2",
    "ollama://mistral",
    "openai://gpt-4"
  ],
  "mcp_services": [
    { "name": "mcp-sql", "tool": "execute_sql", "exploitable": true },
    { "name": "mcp-shell", "tool": "execute_shell", "exploitable": true }
  ]
}
```

This structure reveals what the attacker actually cares about: not the host itself, but the **cloud credentials**, **Kubernetes cluster privileges**, **AI model access**, and **exploitable MCP tools** that live on it.

Targeted AI Service Harvesting via Shodan (ai_harvest.py)

```
# Shodan API target query list
```

```
TARGETS = {
  'comfyui': 'port:8188',
  'ollama': 'port:11434',
  'n8n': 'port:5678',
  'openwebui': 'http.title:"Open WebUI"',
  'langflow': 'http.title:"Langflow"',
  'sdwebui': 'port:7860',
  'gradio': 'port:7861'
}
```

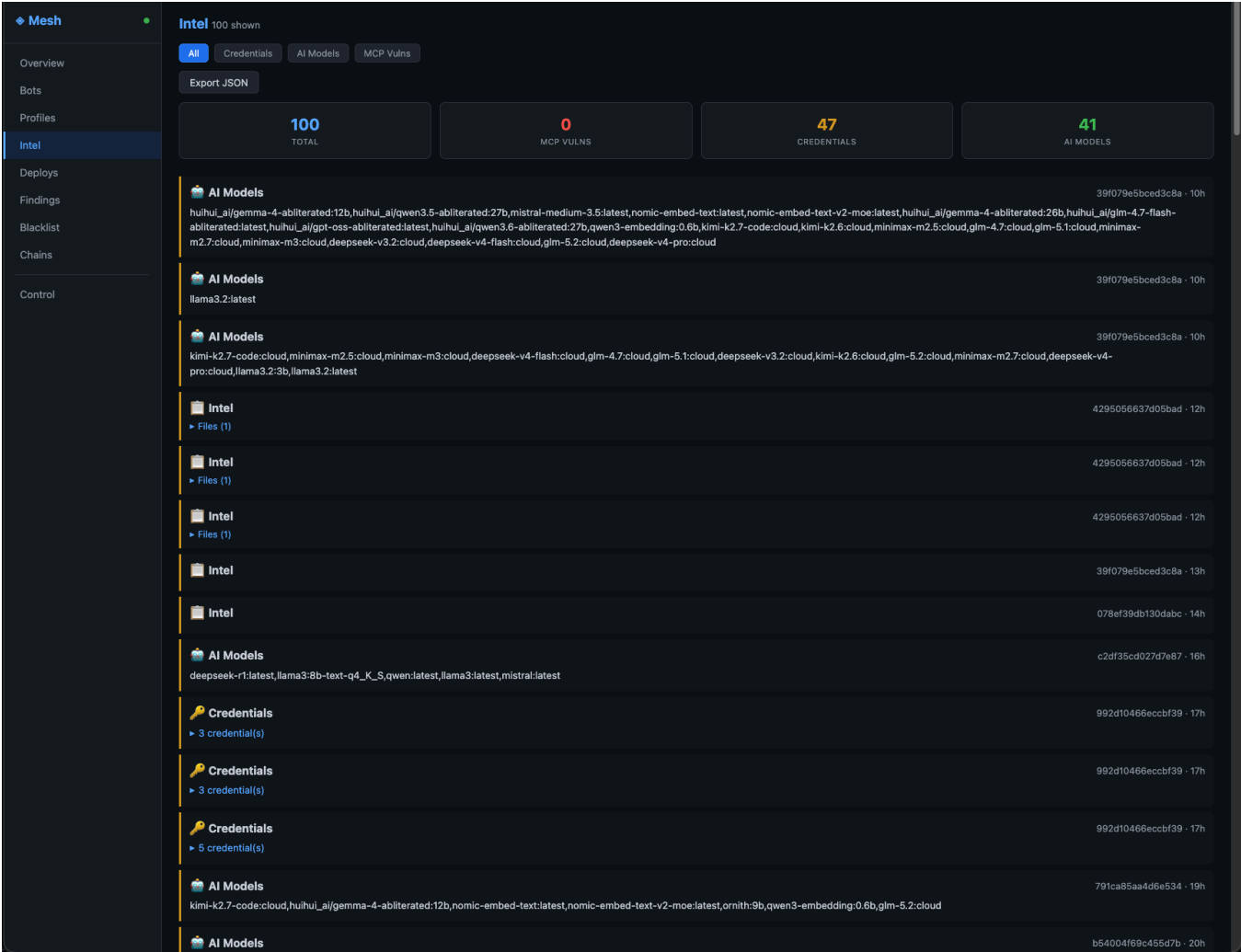
```
# Each query result is injected as a high-priority (priority=20) /32 task
```

```
# Shodan API key: ****
```

Intel Panel

The controller panel aggregates and displays the following categories of harvested data:

- **Credential count:** unique AWS AccessKeyIds
- **MCP vulnerabilities:** exploitable execute_sql / execute_shell services
- **AI models:** Ollama (llama2/mistral) + ComfyUI + Open WebUI
- **K8s ServiceAccounts:** cluster privilege tokens
- **Environment variable credentials:** credentials reusable across hosts
- **Docker hosts:** container hosts open to escape



IoC

C2

209.99.186[.]235
cdnorigin[.]net

Sample SHA1

31c69b3e12936abca770d430066f379ec1d997ec