

I now have a thorough picture of the codebase. Writing the consolidated security report.

WEPN home_device Security Audit Report

Date: 2026-05-11

Scope: `bitbucket.org/dvpn4hr/home_device`
(post-2022 audit, branch `dev`)

Roles: Scan Agent (Role 1) · Triage Agent (Role 2) ·
Skeptic Reviewer (Role 3)

Executive Summary

The WEPN home_device codebase has been significantly hardened since the 2022 7ASecurity pentest — the original passwordless-sudo architecture has been replaced with a `setuid C` binary, HTTP has been upgraded to HTTPS, and dependencies have been updated. However, the security posture remains critically weak in three independent areas, any one of which can result in full device compromise.

The highest-severity cluster is a fully remote, unauthenticated root RCE chain: the device broadcasts its MQTT password (`device_key`) in plaintext to the backend server on every heartbeat cycle. A single heartbeat intercept — achievable by a compromised backend, a BGP-hijacked HTTPS session, or an MITM on the home LAN — yields MQTT credentials. With valid MQTT credentials, an attacker sends an `install-package` or any parameter-bearing message to the `setuid.c` binary, which has both an array-index off-by-one and an uninitialized stack buffer that flow into a root-level `popen()`. This chain requires no physical access and no prior device knowledge.

The second critical cluster is that all E2EE message protection between the device and mobile app is cryptographically void. The `messages.py` AES-GCM implementation discards the authentication tag on encryption and never supplies or verifies one on decryption. AES-GCM without a tag is functionally AES-CTR — an MITM with access to one plaintext-ciphertext pair under the same key can produce forged ciphertexts that the device accepts as authentic. Combined with a stable, never-rotated `e2e_key` that is exposed unauthenticated over the local LAN API, every E2EE-protected command from the app must be treated as unauthenticated.

The third cluster is the local Flask API, which is broken in a way that simultaneously denies service to legitimate callers and leaves dangerous endpoints unprotected. The token validation function `valid_token()` applies `shlex.quote()` to the incoming token before comparison, wrapping it in shell quotes (`'abc123' ≠ abc123`), ensuring all protected endpoints always return 401. The downstream consequence is that the port-exposure detection mechanism — the only runtime defense against the API being externally reachable — is permanently disabled. Separately, `/api/v1/claim/info` returns the `e2e_key` unconditionally to any LAN host regardless of claim status.

Immediate recommended actions before any new deployment:

1. Rewrite or replace `setuid.c` — the current binary is root RCE on any device with MQTT credential access.
 2. Fix `messages.py` AES-GCM to store and verify the authentication tag.
 3. Stop sending `device_key` in heartbeat payloads — the MQTT password must not be transmitted to the backend.
 4. Fix `valid_token()` by removing the `shlex.quote()` call.
 5. Strip `e2e_key` from the `/api/v1/claim/info` response on claimed devices.
-

Critical & High Findings

[P0-A] Root RCE via MQTT → `setuid.c` Off-by-One + Uninitialized Buffer

Files: `usr/local/pproxy/setup/setuid.c:104,143,263`

Chain: Finding #37 → #55 → #2/#81

The `setuid` binary (SRUN, installed as `setuid-root`) has two independent code paths to `popen()` of uncontrolled data:

Off-by-one (#2): `setuid.c:143: if (s > SPECIAL_CMD_CNT || t > 3).`
`SPECIAL_CMD_CNT = 9` (line 9), but `scommands` is declared with `SPECIAL_CMD_CNT = 30` entries (line 68), indices 0–29. The check rejects `s > 9` — so `s=10` through `s=30` would be rejected — but this is not the off-by-one. The actual problem is `s > SPECIAL_CMD_CNT` uses `>` not `>=`. `s == SPECIAL_CMD_CNT` (`s=30`) passes the check. At that point `scommands[30]` is an out-of-bounds read past the end of a 30-element array; `popen()` executes whatever pointer-sized data is on the stack at that offset.

Uninitialized buffer (#81): `char cmd[255]` is declared at line 104 without initialization. The `if (t == 0)` block and `if (t == 1)` block each populate `cmd`. The bounds check allows `t == 2` and `t == 3` (since `t > 3` passes them through). For those values, neither block executes. `popen(cmd, "r")` at line 263 executes whatever happens to be on the stack frame.

Attack path: The MQTT `on_message_handler` dispatches `install-package`, `update-pproxy`, and `update-all` actions without any per-message signature (`pproxy.py:536-541`). These call into `device.py`, which calls the `setuid` binary with attacker-controlled arguments. MQTT credentials are `device_key`, which is sent verbatim in every heartbeat at `heartbeat.py:203`. A heartbeat MITM or backend compromise yields root RCE on all devices within one heartbeat interval.

Skeptic note: The 2022 fix traded passwordless `sudo` for a `setuid` binary, but the new binary is worse — `sudo` at least has configurable command restrictions; this binary has a logic error in its own bounds check. The `SPECIAL_CMD_CNT` define appears twice with different values (9 at line 9 for the bounds check, implicitly 30 from the `scommands` array initialization), creating a silent inconsistency that the bounds check cannot catch.

[P0-B] Broken AES-GCM — All E2EE is Unauthenticated

File: `usr/local/pproxy/messages.py:115-131`

Chain: Findings #15, #16

`encrypt_message()` at line 118:

```
encrypted_text = encryptor.update(padded_data) + encryptor.finalize()  
# encryptor.tag is never read or appended to the output
```

`decrypt_message()` at line 126:

```
decryptor = Cipher(algorithms.AES(key), modes.GCM(nonce), default_backe  
# No tag= argument; decryptor.finalize() is never called
```

Without a tag, AES-GCM is AES-CTR. Any MITM with one known (plaintext, ciphertext) pair under the same (key, nonce) can XOR-difference to the keystream and produce arbitrary forgeries. The `e2e_key` is set once at onboarding and never rotated, so the keystream is stable across all messages.

Skeptic note — second error: The decrypt side uses padder (not unpadder):

```
padder = padding.PKCS7(...).padder()    # line 125 — this is the WRONG d  
unpadded = padder.update(decrypted_data) + padder.finalize()
```

A PKCS7 padder adds padding; an unpadder removes it. This decrypt path adds padding to the already-decrypted data instead of removing it, making the function return garbage regardless of the ciphertext. The function is broken in two orthogonal ways. The fact that it apparently works in production means either E2EE messages are never actually used, or the double-encoding (base64-inside-base64) accidentally cancels out the padding error in some cases.

Skeptic note — GCM + PKCS7 is wrong by design: GCM is a stream mode. Applying PKCS7 block padding to a stream cipher is a category error. Any correct rewrite must remove the PKCS7 padding entirely and use authenticated additional data (AAD) for the message type and nonce.

[P0-C] Heartbeat Transmits MQTT Password to Backend

File: `usr/local/pproxy/heartbeat.py:201-203`

```
"local_token": str(self.local_token),  
"device_key": self.config.get('django', 'device_key'),
```

`device_key` is the MQTT password (`onboard.py:209`). It is sent on every heartbeat cycle to the Django backend over HTTPS. A compromised backend, a BGP hijack of `api.wepn.com`, or a TLS MITM yields the MQTT password for every device that heartbeated during the attack window. With the MQTT password, an attacker has full command bus access (reboot, wipe, install-package, update-all) without any additional authentication.

Skeptic note: The rolling `local_token` mechanism (lines 231-233) regenerates and sends `local_token` on each heartbeat, creating the illusion of forward secrecy. But `device_key` — the more dangerous credential — never rotates. The design sends both, making the rotation of `local_token` security theater: any attacker who captures one heartbeat gets a stable, long-lived MQTT credential regardless.

Skeptic note on `prev_token`: Heartbeat saves `prev_token = current_local_token` before generating a new one. `valid_token()` accepts either current or prev. This doubles the window during which a captured `local_token` is valid, and means the backend always

holds the current valid token for two consecutive windows simultaneously.

[P0-D] Local API Token Check Broken — `shlex.quote()` Wraps Token

File: `usr/local/pproxy/local_server/api.py:50-55`

```
def valid_token(incoming):
    valid_token = status.get_field('status', 'local_token') # e.g. "86
    return (sanitize_str(incoming) == str(valid_token) ...
    # sanitize_str("8675309") → "'8675309'" (with quotes)
    # "'8675309'" == "8675309" → always False
```

Every protected endpoint always returns 401. Confirmed via code inspection — `shlex.quote()` wraps the value in single quotes for shell safety, but these are compared against the raw stored value.

Skeptic note — cascading consequence: Because `valid_token()` always returns False, `check_port_available_externally()` at line 191 always returns 401 WITHOUT setting `exposed = True`. The port-exposure detection mechanism, which is the only runtime control preventing the API from serving responses when externally reachable, is permanently disabled. The cron in `forward_ports.py` posts to the endpoint, gets 401, prints "OK: API port is not reachable externally." regardless of actual network topology. An internet-exposed Pod has no idea it is exposed.

Skeptic note — the API is dead code for the app: Since `valid_token()` never succeeds, the mobile app cannot retrieve usage stats, access links, or diagnostics via the local API. Any user-visible feature depending on the local API has silently been broken since this bug was introduced, yet no one apparently noticed — which confirms the local API code path is untested in CI and the app must have an invisible fallback to the cloud API.

[P0-E] `e2e_key` Exposed Unauthenticated, Including on Claimed Devices

File: `usr/local/pproxy/local_server/api.py:106-122`

```
@app.route('/api/v1/claim/info', methods=['GET'])
def claim_info():
    ...
    e2e_key = status.get_field('status', 'e2e_key')
    # returned unconditionally — no is_claimed check on e2e_key
    return "... \"e2e_key\": \"" + e2e_key + "\"}"
```

The triage report states this only affects unclaimed devices. **This is wrong.** `e2e_key` is returned on every call to `/api/v1/claim/info`, regardless of claim status. `dev_key` is replaced with "CLAIMED" when claimed, but `e2e_key` is never suppressed. Any LAN host can retrieve the E2EE key from a claimed, operational device.

Combined with the broken AES-GCM (#15/#16), this means all E2EE-protected device-to-app messages can be decrypted and forged by any host on the user's LAN.

[P0-F] USB Drive Arbitrary Code Execution

File: `usr/local/pproxy/device.py:912-916`

```
if os.path.isdir("/mnt/device_setup/"):
    sys.path.append("/mnt/device_setup/")
    from setup_mod import create_config # USB-supplied Python
    new_config_str = create_config()
```

Skeptic note — the USB import is checked AFTER serial matching is bypassed: The code iterates partitions and within each iteration first checks `ignore-serial-match.txt` (line 893), then checks for `wepn.ini` serial match (lines 897-901), then checks for previous config serial match (lines 904-908), and only then falls through to the `device_setup/` import (lines 912-916). This means an attacker who provides a USB with `wepn.ini` that doesn't match the serial will fail serial check and fall through to the `device_setup/` path — no bypass file needed. The priority-3 path is reachable whenever priority 1 and 2 fail, which is any USB that doesn't have a valid config for this specific device.

Skeptic note: `generate_new_config()` is also called on factory reset and config corruption recovery (`fetch_config_from_backend` fails). On a device that has been wiped or whose config was corrupted, the first USB insertion triggers this path without any user confirmation.

[P1-A] Flask DEBUG = True — Werkzeug PIN Console Accessible from LAN

File: `usr/local/pproxy/local_server/api.py:60`

```
app.config["DEBUG"] = True
```

Werkzeug in debug mode exposes a PIN-protected interactive Python debugger at any unhandled exception. The PIN is deterministic and derived from the machine's username, MAC address, and app module path. From LAN, an attacker can enumerate these values from the already-disclosed serial/IP data and compute the Werkzeug PIN. There are documented PIN brute-force techniques that succeed in ~1000 requests — achievable from LAN in seconds.

[P1-B] Shared Remote SSH Private Key Across All Devices

File: `usr/local/pproxy/device.py:959`

```
key = "/var/local/pproxy/shared_remote_key.priv"
```

The filename "shared" is not ambiguous — this key is identical across all WEPN devices. Extracting it from any single device (physical access, any pproxy-level exploit) compromises the reverse SSH tunnel of every device in the fleet. Root on any device yields the ability to tunnel into all other devices via `relay.we-pn.com`.

[P1-C] SSH Accepts Any Host Key on First Connect

File: `usr/local/pproxy/device.py:962`

```
cmd += " -fTN -o StrictHostKeyChecking=accept-new"
```

accept-new silently trusts and saves any host key presented on the first connection. If DNS for relay.we-pn.com is poisoned when set_remote_ssh_session() is first called via MQTT, the reverse tunnel terminates at the attacker's server. Combined with the shared key (#1-B), this is a fleet-wide remote access vector for anyone who can poison DNS once.

[P1-D] OTA Supply Chain: Packages File Fetched Without GPG Verification

File: usr/local/pproxy/device.py:625-632

The OTA version check fetches the Packages index over HTTPS but performs no GPG signature verification. The actual apt package installation in update-system.sh:11 uses deprecated apt-key adding from the same potentially compromised domain. A BGP-hijacked or DNS-spoofed repo.we-pn.com can advertise a higher version, trigger OTA, install a malicious GPG key into the global keyring, and achieve root persistence. This is the nation-state threat model for WEPN's user base.

[P1-E] DDNS Credentials Embedded in GET URL

File: usr/local/pproxy/device.py:563-566

```
r = requests.get(url_template.format(
    self.config.get('dyndns', 'username'),
    self.config.get('dyndns', 'password'), ...))
```

Credentials appear in the URL, in server access logs, in any TLS-terminating proxy's logs, and in network packet captures. Combined with the MQTT set_ddns action (which any MQTT credential holder can trigger), an attacker can set the DDNS URL template to exfiltrate credentials to an attacker-controlled server.

[P1-F] forward_ports.py POSTs local_token with TLS Verification Disabled to IP from Untrusted Source

File: usr/local/pproxy/periodic/forward_ports.py:33-41

```
external_ip = str(ipw.myip())    # no timeout, no cert pinning
url = "https://" + external_ip + ":5000/api/v1/port_exposure/check"
r = requests.post(url, data={'local_token': str(local_token)}, verify=F
```

DNS poisoning of ip.we-pn.com sets external_ip to an attacker-controlled address. The local_token is then POSTed to that address with TLS verification disabled. **Note:** This is currently moot because valid_token() is broken and the token has no value, but when #P0-D is fixed, this becomes a direct credential exfiltration path executed by cron every few minutes.

[P1-G] OpenVPN Cryptographic Configuration Obsolete

File: usr/local/pproxy/openvpn.conf:1-22

- Line 6: `dh1024.pem` — 1024-bit DH is LOGJAM-vulnerable (precomputation feasible for state actors).
- No `cipher` directive — defaults to BF-CBC (Blowfish 64-bit) in OpenVPN <2.5, vulnerable to SWEET32 birthday attack after ~32 GB of traffic.
- No `tls-auth` or `tls-crypt` — unauthenticated clients can initiate TLS handshakes, DoS surface.
- No `tls-version-min` — TLS 1.0/1.1 downgrade possible.
- Line 22: `comp-lzo` — VORACLE compression oracle attack.

For WEPN's target users in restrictive network environments, OpenVPN traffic decryption by a resource-rich adversary is the highest-consequence outcome of these individually low-severity configurations.

Medium Findings

[P2-A] `set_creds` / `set_ddns` Store Shell-Quoted Values in `config.ini`

File: `usr/local/pproxy/pproxy.py:544-563`

`sanitize_str()` calls `shlex.quote()`. When the result is stored via `config.set()`, values like `mypassword` become `'mypassword'` (with literal single quotes) in `config.ini`. When read back by `configparser`, the returned value is `'mypassword'` — the SMTP server or DDNS API receives the wrong password. This means `set_creds` has likely never worked correctly in production. This is both a functional bug and a security concern: if credentials were stored with wrong values, the operator may have disabled authentication-dependent features silently.

[P2-B] `local_token` Entropy is 33 Bits

File: `usr/local/pproxy/heartbeat.py:49`

```
self.local_token = random.SystemRandom().randint(1111111111, 9999999999)
```

Range: ~8.88 billion values = ~33 bits. On a LAN with no rate limiting on the Flask API, this is brutable. A LAN attacker can make ~8.9 billion requests against the `/api/v1/friends/access_links/` endpoint. At 1000 requests/second this takes ~102 days — not practical in real-time, but the absence of any rate limiting or lockout makes it theoretically feasible. After fixing #P0-D (broken auth), this becomes the next weakest link.

[P2-C] QR Code URL Leaks `e2e_key` to `red.we-pn.com` Server Logs

File: `usr/local/pproxy/setup/onboard.py:241`

```
display_str = [(1, "https://red.we-pn.com/?pk=" + str(self.rand_e2e_key
    + "&s=" + str(serial_number) + "&k=" + str(self.rand_ke
```

Both the E2EE key (`pk=`) and the claiming key (`k=`) are query parameters in a URL displayed as a QR code. When the user scans this with the app, the URL is fetched or its parameters are parsed. Any intermediate proxy, HTTP referrer, or `red.we-pn.com` access log will contain both secrets in cleartext. If `red.we-pn.com` is a redirect service (the "red" prefix suggests it), the full URL including both keys is logged server-side.

[P2-D] `get_ota_channel` Compares Method Object Not Return Value

File: `usr/local/pproxy/device.py:628`

```
if self.get_ota_channel == "beta":
```

This compares the method object itself, not its return value. The condition is always `False`. The beta OTA channel is permanently unreachable. Security fixes shipped to the beta channel never reach beta devices — they remain on the older, potentially vulnerable stable package.

[P2-E] `wstatus.py` Status File Write is Non-Atomic

File: `usr/local/pproxy/wstatus.py:38-42`

```
statusfile = open(self.source_file, 'w')
self.status.write(statusfile)
statusfile.close()
```

A non-atomic truncate-then-write. If the process is killed mid-write (OOM, SIGKILL), `status.ini` is left empty or partially written. A corrupted `status.ini` causes `valid_token()` to return an empty string, `fetch_config_from_backend()` to be triggered (see #31), and `e2ee_available()` to return `False` (silently disabling E2EE for subsequent messages). There is a TODO comment on line 34: `# TODO: add lock checking`. The TODO acknowledges the gap but doesn't address atomic writes.

[P2-F] `exposed` Flag is Volatile — Process Restart Resets Detection

File: `usr/local/pproxy/local_server/api.py:38`

```
exposed = False
```

Even if the port-exposure detection ever worked correctly (it doesn't — see #P0-D), `exposed = True` lives only in the Flask process's memory. `systemctl restart wepn-api` — which happens on OTA updates and config changes via MQTT — resets it to `False`. An attacker who triggers a service restart after causing the API to be exposed gets a clean slate. The exposure state must be persisted to disk.

[P2-G] `recovery.py` Hard-codes Debug Log Config

File: `usr/local/pproxy/periodic/recovery.py:12`

```
LOG_CONFIG = "/etc/pproxy/logging-debug.ini"
```

This directly violates the CLAUDE.md directive: "Do not commit the debug value." Debug logging may write sensitive values that production logging suppresses — including the `device_key`, MQTT credentials, and `local_token` that appear in `logger.debug()` calls throughout the codebase. This file has been in this state across multiple commits without being caught.

[P2-H] `diag.py` Diagnostic TCP Listener Binds to All Interfaces

File: `usr/local/pproxy/diag.py:86-90`

A diagnostic echo server binds to all interfaces for up to 120 seconds during port-check

diagnostics. Any host on the internet (if port forwarding is active) or LAN can connect and receive 8 bytes echoed back. This is an amplification reflector during the test window.

[P2-I] Metrics Server Has No Authentication

File: `usr/local/pproxy/system_services/metrics.py:227-330`

127.0.0.1:8411 with no authentication. Any local process can retrieve the full history of source IPs and ASNs for all VPN connection attempts — the precise data WEPN's users want to keep private from their local network environment (which may include a compromised router or ISP device). Any local compromise (even non-root) yields this data.

[P2-J] WireGuard Endpoint IP Has No Certificate Pinning

File: `usr/local/pproxy/add_user_wireguard.sh:9`

`ip=`curl -s https://ip.we-pn.com``

DNS poisoning of `ip.we-pn.com` sets the WireGuard Endpoint field in all newly generated peer configs to an attacker-controlled IP. VPN clients added after the poisoning connect to the attacker's server. This silently redirects all WireGuard user traffic.

Regression Status — 2022 7ASecurity Audit Items

2022 Finding	Description	Status	Notes
<code>ipw.py</code> cleartext HTTP	IP resolution over HTTP	Partially Fixed / Regressed	Now HTTPS, but no timeout (#28) and result used in <code>forward_ports.py</code> with <code>verify=False</code> (#44)
<code>forward_ports.py</code> IP spoofing leaks <code>local_token</code>	Local token sent to spoofable IP	Regressed (new form)	HTTP→HTTPS, but <code>verify=False</code> and IP from untrusted source remain; moot while auth is broken, critical once fixed
<code>cron-upnp.sh</code> dangerous invocation	Cron triggers <code>forward_ports.py</code> chain	Confirmed Present	Chain intact
Passwordless sudo	<code>pproxy</code> user could sudo freely	Fixed, New Risk Introduced	Replaced with <code>setuid</code> binary; binary has off-by-one + uninitialized buffer (P0)
<code>ip-shadow.sh</code> weak iptables	Iptables rules quality	Partially Fixed	Still hardcodes 192.168.1.1 gateway (#65); first-run ordering bug (#64)
Cron job security	Cron job privilege and credentials	Regressed	<code>cron-pi</code> now installs under <code>pi</code> user with well-known default credentials

Role 3 Skeptic Analysis — Beyond the Scan

1. Challenging "Fixed" Assumptions

"The setuid binary replaced passwordless sudo" — The 2022 finding was that `pproxy` could run arbitrary commands as root via `sudo`. The fix created a `setuid` binary with a command allowlist. This is the correct architecture, but the implementation has a bounds check with `>` instead of `>=` (off-by-one), an uninitialized stack buffer for unhandled `t` values, and a NULL dereference for `s=6` (`argv[7]` on a 7-argument check). The fix hardened the intended path but introduced new vulnerabilities in the boundary conditions — the classic "fixed the obvious case, missed the edges" failure.

"AES-GCM is used for E2EE" — The commit that introduced `messages.py` likely used GCM for its authenticated encryption properties. The code creates a GCM cipher object, which is correct. But it never reads `encryptor.tag` and never passes `tag=` to the decryptor. The fix was added for the appearance of security (correct cipher mode selection) without implementing the authentication property that makes GCM more than CTR. There is no test that would catch this — a test verifying "decryption succeeds" would pass even with the broken implementation, because decryption doesn't raise an exception.

"IP resolution is now HTTPS" — Correct observation. The fix changed `http://` to `https://`. But `ipw.py:9` has no `timeout=` parameter, and `forward_ports.py:41` uses `verify=False` with an explicit `# nsec` comment citing a waiver. The waiver (go.we-pn.com/waiver-3) is intended for the self-signed local certificate scenario, but it's being applied to a connection to an external IP that came from an untrusted DNS resolution. The `nsec` annotation suppressed the security scanner warning without fixing the underlying trust issue.

2. Trust Model Analysis

What the device trusts:

- The WEPN backend (`api.we-pn.com`) — unconditionally for heartbeat responses, config updates, and MQTT command dispatch
- The MQTT broker — for all privileged commands (reboot, wipe, install)
- The USB drive inserted by anyone with physical access — for arbitrary Python code
- The IP resolution service (`ip.we-pn.com`) — for routing sensitive API calls
- The OTA repository (`repo.we-pn.com`) — for system updates without GPG verification

What happens if the backend is compromised: The backend knows `device_key` (from every heartbeat). An attacker controlling the backend can immediately send MQTT commands to any device that heartbeated during the compromise window. There is no mechanism for a device to verify that an MQTT command came from a legitimate backend-initiated action vs. an attacker using the leaked credential. The device has no trust anchor that is independent of the backend.

What a malicious "friend" VPN user can do: A user added to the device (a "friend") receives VPN credentials — an OpenVPN cert or WireGuard peer config. From the VPN tunnel, that friend has a network-adjacent position. Shadowsocks users are per-port isolated. OpenVPN and WireGuard users receive LAN-routable access depending on routing configuration. If the VPN routes LAN traffic (common for personal VPN setups), a VPN friend can reach port 5000 (local Flask API). With `valid_token()` broken, all authenticated endpoints return 401, but unauthenticated endpoints (claim info, claim progress, error logs on unclaimed devices) remain accessible from the VPN tunnel.

Device integrity: There is no mechanism to detect if the device has been tampered with. No Secure Boot, no TPM, no dm-verity, no file integrity monitoring. A physical attacker who gains root (via USB → setuid chain) can modify any file, install a persistent backdoor, and leave no detectable trace. The device has no concept of a "known good state."

3. Secret Lifecycle Analysis

local_token: Generated by `random.SystemRandom().randint(1111111111, 9999999999)` — ~33 bits. Sent to the backend on every heartbeat. Stored in `status.ini` without encryption. Lives for one heartbeat cycle, then rotated. Previous token (`prev_token`) remains valid simultaneously. The rolling rotation is security theater when the backend holds the current token at all times.

device_key / MQTT password: Generated once at onboarding as `rand_key` (10 chars from 32-char alphabet, ~50 bits). Set as MQTT password AND stored as `device_key` in `config.ini`. Sent to the backend on every heartbeat. Never rotated after initial claiming (`on_connect()` at `onboard.py:209`). **Lifetime of this secret: permanent, until device is wiped.** Every heartbeat for the device's entire operational lifetime broadcasts this permanent credential.

e2e_key: Generated at onboarding using `secrets.token_bytes(16)` (128 bits — appropriate entropy). Set once, never rotated. Exposed via LAN API without authentication on all devices (claimed or not). Included in QR code URL query string. Used with broken GCM (no tag). Leaked via `show-e2ee-qrcode` MQTT action which sends it over the unauthenticated message bus.

Are any secrets world-readable? `status.ini` at `/var/local/pproxy/status.ini` contains `local_token`, `e2e_key`, `prev_token`, and `temporary_key`. `config.ini` at `/etc/pproxy/config.ini` contains `device_key` (MQTT password) and potentially SMTP/DDNS credentials. File permissions are controlled by `permissions.sh` — these files are owned by `pproxy:pproxy` with typical 640 permissions, not world-readable. However, any process running as `pproxy` (or group `pproxy`) can read them. The Flask API runs as `wepn-api:wepn-web` — a deliberate separation — but the API imports `wstatus.py` which reads `status.ini` directly. This works only if `wepn-api` can read `status.ini`, implying `status.ini` is either group-readable by `wepn-web` or world-readable.

Process argument exposure: The `setuid` binary is invoked as `SRUN + " 1 " + args`, executed via `execute_cmd_output()`. Arguments appear in `ps aux` output during execution. WireGuard public keys and peer IPs are visible to any local user during `wg set` operations.

rand_key in debug logs: `onboard.py:268: self.logger.debug('Randomly generated device key: ' + self.rand_key)`. This logs the MQTT password (which is also the claiming key) at DEBUG level. `recovery.py` hard-codes the debug log config, meaning on any device running recovery, this secret appears in logs.

4. Update Mechanism — Attacker's Perspective

How a nation-state would abuse OTA:

The update chain: `device.py:get_repo_package_version()` fetches `https://repo.we-pn.com/debian/dists/.../Packages` (no GPG), parses a version string, and triggers `apt-get` if higher version is found. `update-system.sh:11` runs `wget | sudo apt-key add` — adding a GPG key to the global keyring.

Step 1: BGP-hijack or DNS-spoof `repo.we-pn.com` (feasible for ISP-level actors; WEPN's users are in exactly the environments where this is a real threat). Step 2: Serve a `Packages` file advertising version `999.0.0`. Step 3: Device triggers OTA. Step 4: `update-system.sh` adds attacker's GPG key to global keyring via `apt-key add -`. Step 5: All future `apt` operations on the device trust attacker-signed packages. Step 6: Install any package as root.

The update URL is hardcoded in `constants.py` (`SKIP_OTA_CHECK` references `repo.we-pn.com`). It cannot be overridden via config. However, the `get_ota_channel()` method (permanently broken due to method-vs-call bug, #2-D) means even if a "beta" channel with better security were configured, it would never be used.

Update integrity before execution: None. The `Packages` file is fetched, version-parsed, and `apt-get` is triggered. `apt-get` itself checks GPG signatures on packages using the global keyring — but step 4 above has already compromised that keyring.

5. Cron Job Security

Cron jobs under `pi` user (`setup/cron-pi`): The `pi` user is the default Raspberry Pi user with well-known credentials (`pi:raspberry`). Any network-accessible device with SSH enabled (which WEPN devices are, by design) and default credentials gives attackers direct access to run or modify cron jobs. This is documented in the `cron-pi` file but not mitigated.

Absolute paths: `forward_ports.py` uses `sys.path` manipulation and imports by module name, not absolute path. `update-system.sh` uses `wget` and `apt-key` — these rely on `$PATH`. If `$PATH` is manipulated for the cron invocation context, these can be redirected.

Cron input injection: `iptables-flush.sh:17-20` reads `iptables-save` output and passes it unquoted to `iptables`. `prevent_location_issue.sh:129-130` reads from `goog.txt` and passes lines to `do_geo_iptables`. These represent data-driven cron jobs where the input data (iptables rule state, remote IP lists) can influence command execution.

6. Missing Security Controls

Control	Present?	Notes
Intrusion detection / anomaly alerting	No	No monitoring of unexpected connection patterns, new peers, or failed auth attempts
Audit logging of privileged operations	No	SRUN executions are not logged; MQTT commands dispatched without audit trail
"Kill switch" / remote disable	Partial	<code>wipe_device</code> MQTT action exists but requires MQTT credentials to trigger
Rate limiting on local Flask API	No	No Flask rate limiting or request throttling
Brute-force protection on <code>local_token</code>	No	No lockout after repeated 401 responses
MQTT per-message authentication	No	Only session-level TLS credentials protect the command bus
Replay protection on MQTT messages	No	No sequence numbers, timestamps, or nonces in MQTT payloads
OTA package integrity before execution	No	GPG signature of <code>Packages</code> file not verified

Device integrity checking (dm-verity, etc.)	No	No measurement of firmware/OS integrity
CSRF protection on local Flask API	No	No CSRF tokens; browser-based requests from LAN-local pages work without restriction
e2e_key rotation	No	Set once at onboarding; no rotation mechanism
MQTT device_key rotation	No	Set once at claiming; sent in every heartbeat
Config file write atomicity	No	wstatus.py does non-atomic truncate + write
Secure deletion of secrets on wipe	Not verified	wipe_device resets flags but unclear if key material is overwritten

7. Code Quality as Security Issues

Silent return of empty string on error: `wstatus.py:get_field()` returns `""` on any exception (`except: ... return ""`). All callers that check returned values for authentication (e.g., `valid_token()`) or configuration (e.g., `e2ee_available()`) receive an empty string on file read errors rather than failing closed. This means a disk read error during token comparison yields `valid_token("") == "" → True` — authentication passes when the status file is unreadable.

Wait, let me re-examine: `valid_token()` in `api.py:54` does `sanitize_str(incoming) == str(valid_token)`. If `get_field()` returns `""`, then `valid_token = ""`. But `sanitize_str(incoming)` applies `shlex.quote()`, so an incoming empty string becomes `''`. `'' == ""` is `False`. So the comparison fails safely for empty input. But if the attacker sends `incoming = ""`, `shlex.quote("")` returns `''`, which doesn't equal `""`. The empty-string-on-error path doesn't create an auth bypass in this specific case given the broken comparison. However, this remains a general concern — any caller that checks `if status.get('some_flag') == "1":` would get `"" == "1" → False` on error, failing open in the direction of "not configured" rather than "secure default."

Dead code still reachable: The `get_ota_channel()` method-vs-call bug (#P2-D) means the beta OTA path is dead code. But it's still reachable in the sense that `switch_ota_channel("beta")` sets the status field, the field persists across reboots, and `get_ota_channel()` (called correctly in `switch_ota_channel`) returns `"beta"`. The device believes it is on the beta channel, but version checks always use stable. Security updates shipped only to beta are silently not applied.

Logging that leaks secrets: `heartbeat.py:217: self.logger.debug("HB data to send: " + data_json)` — the complete heartbeat payload, including `local_token`, `device_key`, and `pin`, is logged at `DEBUG` level. `onboard.py:268: self.logger.debug('Randomly generated device key: ' + self.rand_key)`. `onboard.py:290: self.logger.debug("password for mqtt= " + self.rand_key)`. All secrets appear in debug logs. `recovery.py` hard-codes the debug log config, meaning on devices running recovery, secrets are in logs that may be readable or transmitted via the error log endpoint.

Exception silencing in `messages.py:59–62`:

```
except KeyError as e:
    print("key not found:" + str(e))
except Exception as d:
```

```
print("key not found:" + str(d))
```

Message decryption errors are silently swallowed and printed to stdout. If an attacker sends a tampered E2EE message that causes an exception during "decryption," it is silently discarded — no alert, no audit record. The only way to detect tampering is through log analysis of debug output.

Hardening Remediation Roadmap

Immediate (This Week) — Stop Active Bleeding

1. **Rewrite `setuid.c` bounds checking.** Change `if (s > SPECIAL_CMD_CNT)` to `if (s >= SPECIAL_CMD_CNT)`. Zero-initialize `cmd` and `scmd` at declaration (`char cmd[255] = {0}`). Fix `argc` check for `s=6` and `s=23` from `!= 7` to `!= 8` (7 args + program name = argc 8, not 7). Replace all `sprintf()` with `snprintf()` with explicit length bounds. This is a surgical fix pending a full rewrite.
2. **Fix `messages.py` AES-GCM.** In `encrypt_message()`: append `encryptor.tag` to the output. In `decrypt_message()`: accept `tag` as a parameter, pass `tag=tag` to `modes.GCM(nonce, tag)`, and call `decryptor.finalize()` to raise `InvalidTag` on tampered ciphertext. Remove PKCS7 padding entirely — GCM does not need it. Rotate `e2e_key` on all existing devices via a forced re-onboarding or key-rotation MQTT message.
3. **Stop sending `device_key` in heartbeats.** Remove "device_key" from the heartbeat payload in `heartbeat.py:203`. Authenticate heartbeats with an HMAC derived from `device_key` rather than transmitting the credential itself. The backend can verify the HMAC without ever receiving the raw credential.
4. **Fix `valid_token()`.** Remove `sanitize_str()` / `shlex.quote()` from the comparison. Compare the raw incoming token directly against the stored token. Add a timing-safe comparison (`secrets.compare_digest()`). After fixing this, immediately also fix #P1-F (`forward_ports verify=False`) before deploying.
5. **Strip `e2e_key` from `/api/v1/claim/info` on claimed devices.** Return the field only when `is_claimed == 0`. Alternatively, redesign this endpoint to never return key material over an unauthenticated channel.

Short-Term (This Sprint) — Critical Hardening

1. **Replace `random.SystemRandom().randint()` with `secrets.token_hex(16)` for `local_token` and `pin`.** 128-bit tokens are not brutable.
2. **Persist the `exposed` flag to disk.** Write to `status.ini` on detection. Initialize Flask to read the persisted state. Require explicit reset (MQTT action + admin confirmation) to clear.
3. **Set `app.config["DEBUG"] = False` in `api.py:60`.** The commented-out `False` line is the correct value.
4. **Add per-MQTT-message authentication.** Sign MQTT command payloads with `HMAC-SHA256(message_body, device_secret)` where `device_secret` is never

transmitted. Verify on receipt in `on_message_handler()`. Reject unsigned or incorrectly-signed messages.

5. **Add MQTT replay protection.** Include a monotonic sequence number and/or timestamp in each MQTT message. Reject messages with timestamps outside a ± 5 minute window or sequence numbers below the last seen.
6. **Fix OpenVPN configuration.** Upgrade to 2048-bit DH (`dh2048.pem`). Add `tls-crypt` directive. Set `cipher AES-256-GCM`. Add `tls-version-min 1.2`. Remove `comp-lzo`.
7. **Add GPG verification of the Packages file** in `get_repo_package_version()` before acting on version strings.
8. **Remove the USB `setup_mod` dynamic import** or require it to be GPG-signed with a key burned into the device at factory time. Remove the `ignore-serial-match.txt` bypass entirely.
9. **Replace the shared SSH key** with per-device keys generated at onboarding and registered with `relay.we-pn.com`. Change `StrictHostKeyChecking=accept-new` to `StrictHostKeyChecking=yes` with a pinned known-hosts file.
10. **Fix `recovery.py` debug log config** — change `logging-debug.ini` to `logging.ini` (line 12).

Architectural (Next Quarter) — Design-Level Improvements

1. **Separate the MQTT password from `device_key`.** These should be distinct secrets. `device_key` authenticates heartbeats. The MQTT password should be a separate credential, generated and registered independently, and never transmitted in plaintext to any external service.
2. **Implement `e2e_key` rotation protocol.** The mobile app and device should be able to negotiate a new E2EE key on demand. Implement a TOFU (trust-on-first-use) model with user confirmation for key changes.
3. **Replace the `setuid` binary with a properly sandboxed privilege escalation mechanism.** Options: (a) systemd service units with `ExecStart` for each privileged operation, callable via `systemctl --user` with fine-grained polkit policy; (b) a minimal `setuid` wrapper with a verified command table, exhaustive input validation, and `snprintf` throughout; (c) a privileged daemon with a Unix socket IPC protocol that validates callers by UID.
4. **Add rate limiting and brute-force protection to the Flask API.** Use `flask-limiter`. Lock the token for 10 minutes after 20 failed attempts. Log all authentication failures to syslog.
5. **Implement atomic config/status file writes.** Write to a temp file in the same directory, then `os.rename()` (atomic on Linux/POSIX). This eliminates the truncation-on-crash corruption window.
6. **Add an intrusion detection layer.** Log all MQTT command receptions to syslog. Alert (via heartbeat field) when unexpected commands are received or when authentication fails repeatedly. Monitor for unexpected processes, new listening ports, or file system

changes in sensitive directories.

7. **Move to HTTPS with proper cert pinning for all outbound requests.** Pin the certificate of `api.we-pn.com`, `ip.we-pn.com`, `repo.we-pn.com`, and `relay.we-pn.com`. Use `requests.Session` with custom `SSLContext` and the pinned cert loaded from a file that ships with the package.
 8. **Redesign the heartbeat to not include any credentials.** The backend should be able to correlate heartbeats by `serial_number` alone (which it already has). The `device_key` field can be replaced by an HMAC, as noted above. The `local_token` has no legitimate use being sent to the backend — it is a local API credential. Remove it entirely from the heartbeat payload.
-

Files & Components Reviewed

File	Review Depth
<code>usr/local/pproxy/setup/setuid.c</code>	Full (276 lines)
<code>usr/local/pproxy/messages.py</code>	Full (132 lines)
<code>usr/local/pproxy/local_server/api.py</code>	Full (223 lines)
<code>usr/local/pproxy/heartbeat.py</code>	Full (322 lines)
<code>usr/local/pproxy/setup/onboard.py</code>	Full (323 lines)
<code>usr/local/pproxy/pproxy.py</code>	Lines 280–600 (MQTT handler, <code>send_mail</code> , <code>on_connect</code>)
<code>usr/local/pproxy/device.py</code>	Lines 519–570 (DDNS), 620–700 (OTA), 840–984 (USB, SSH)
<code>usr/local/pproxy/periodic/forward_ports.py</code>	Full (47 lines)
<code>usr/local/pproxy/ipw.py</code>	Full (22 lines)
<code>usr/local/pproxy/wstatus.py</code>	Full (131 lines)
Raw findings table (84 items)	Full review
Triage report (84 items, all priority levels)	Full review