SECAUDIT 030 - Skin Manifest Arbitrary Code Execution (RCE)

SECURITY AUDIT / PUBLIC REMEDIATION RECORD

Field	Value
Audit ID	2026-06-28-030
Date	2026-06-28
Severity	**CRITICAL** - unauthenticated/low-trust -> arbitrary PHP execution
Component	Skin manifest loading; skin installer
Status	**REMEDIATED in 0.7.440** - deployment gated on rebuilding every hosted skin ZIP/registry signature
Reporter	Sean (identified the trust-boundary crossing) + Claude (traced the path)
Related	006 (post-release integrity), 017 (SMACKBACK file integrity), `project_launch_security_arch`

1. Summary

Every skin ships a `manifest.php` that the CMS loads with PHP `include` - i.e. **executes** - in both public-facing pages and the authenticated admin panel. Because the file is executed rather than parsed as data, any skin present on disk can run arbitrary PHP in the CMS process: read/modify the database, exfiltrate secrets, write files, alter settings, or silently subvert the admin UI. The colloquial worry ("a bad skin could change admin options") understates the issue - it is full Remote Code Execution scoped to whatever the PHP process can do.

Neither of the two trust checkpoints prevents this:

1. **Install-time signature verification is optional and effectively disabled.**
2. **Runtime manifest loading performs no verification at all** (bare `include`).

2. Technical detail & evidence

2.1 Runtime: bare include, no verification

The active skin's manifest is included directly, e.g.:

```
$manifest = include "skins/{$active_skin}/manifest.php";
```

Observed in (non-exhaustive): `smack-globalvibe.php`, `smack-appearance-solo.php`, `smack-appearance-archive.php`, `smack-edit.php`, `smack-masthead.php`, `core/admin-header.php`, `gallery-wall.php`, `archive.php`, `index.php`, `core/community-component.php`, `skins/*/skin-footer.php`. No signature or hash check precedes any of these includes.

The shipped manifests are not inert data - they execute logic at load. Example (`skins/impact-printer/manifest.php`): runs `include(...manifest-inventory.php)` and `array_filter(...)` at top level before `return`. This confirms manifests are an executing-code surface by design.

2.2 Install-time: signature verify is opt-in

`core/skin-registry.php` -> `skin_registry_install()`:

```
function skin_registry_install(string $slug, string $download_url,
string $signature = '', string $public_key = ''): array {
...
```

```
// --- SIGNATURE VERIFICATION (optional) ---
if (!empty($signature) && !empty($public_key)
&& function_exists('sodium_crypto_sign_verify_detached')) {
... // verify; reject on failure
}
```

If either `$signature` or `$public_key` is empty, verification is **skipped** and the skin installs unconditionally. Registry entries carry the comment `"signature": ... // optional until signing is live`, confirming signing is not currently enforced. A sideload/manual-upload path (if present) would bypass the registry entirely - TO BE CONFIRMED and closed.

2.3 SMACKBACK does not close this

`core/auth-smack.php` (~lines 209-246) redirects admin pages to the breach screen only when `smackback_enabled=1` AND `smackback_status='breach'` AND mode ≠ `alert`. This is:

- **Reactive** - detects tampering against a stored baseline *after* the fact,

on a verify pass; it does not gate the include.

- **Opt-in** - no protection when SMACKBACK is disabled.
- **Blind to malicious-by-design skins** - a skin whose original files are

themselves malicious matches its own baseline and never trips a breach.

3. Impact

- Arbitrary PHP execution in admin and public contexts.
- DB read/write (content, users, settings, hashes), secret exfiltration,

file write, persistence, UI subversion (e.g. hiding security controls to social-engineer the owner).

- Trust model gap: skins are forkable, third-party-distributable deliverables,

yet are treated as fully trusted code the moment they are on disk.

4. Exploitation scenario

Owner installs a forked/third-party skin (gallery entry without a signature, or a manual zip). On the next page load the skin's `manifest.php` executes with full CMS privileges. No authentication bypass is required - the skin install itself is the delivery vector, and install does not require a valid signature.

5. Remediation (sequenced - see spec for detail)

The fix must **preserve the manifest's UI-control capability** (a required product feature) while removing code execution.

1. **Declarative manifests.** Replace executed `manifest.php` with parsed

`manifest.json` (+ optional `manifest.md` notes). Core reads via a single validating loader `load_skin_manifest()` and never `includes` skin code.

2. **Core-owned hideable-controls allowlist.** `hide_controls` is honored only

for non-essential controls; essential/security controls can never be hidden.

3. **Move computed manifest logic into trusted core** (e.g. `allowed_fonts`

glob patterns expanded by core against the font inventory).

4. **Mandatory signing as defense-in-depth, sequenced:** Packager signs ALL

skins first; THEN make `skin_registry_install()` verification mandatory (empty signature = reject). Do NOT flip before all skins are signed (would brick the gallery - cf. SMACKBACK false-breach lockout).

5. **Close any sideload/manual-upload path** (require signature there too).
6. **Defense-in-depth runtime check:** refuse to load a manifest whose recorded

signature/hash does not verify.

Note: step 1 alone collapses severity from CRITICAL (RCE) to LOW (a rogue skin can at most hide a non-essential control), which is why it is the priority.

6. Verification (post-remediation - to be a follow-up audit)

- Confirm no remaining `include` of any `skins/*/manifest.php` after migration.
- Confirm `load_skin_manifest()` rejects/ignores unknown keys and code payloads.
- Confirm essential controls cannot be hidden via `hide_controls`.
- Confirm install rejects unsigned skins once enforcement is enabled.

7. Closure record - 0.7.440

The critical manifest-execution path is closed:

- All 23 current `skins/*/manifest.php` files were converted to declarative

`manifest.json` and deleted.

- `core/skin-manifest.php` is the only runtime loader. It validates the slug,

limits the file to 1 MiB, parses with `JSON_THROW_ON_ERROR`, accepts a fixed top-level schema and fixed option-type catalogue, bounds nested data, and never executes values.

- Every public, admin, mobile, installer, registry, OH SNAP, and packaging

consumer was migrated. Repository scans find no `include` or `require` of a skin manifest and no remaining skin `manifest.php`.

- Before deletion, every JSON document was deep-compared against the array

returned by its trusted legacy PHP source. All 23 were byte-value equivalent after removing the new `schema_version`; normalized loader option counts and feature maps also matched.

- PHP syntax checks passed for all changed PHP files. Package smoke tests use

`manifest.json` as canonical data and add a build-time-only compatibility adapter inside the signed ZIP so pre-0.7.440 core can still load the package. The adapter is absent from source and ignored by new core, preserving rollback in both directions during the fleet transition.

Package signing is mandatory in 0.7.440 for both Skin Gallery and fresh-install downloads. Missing Sodium, signature, or canonical release public key fails closed; the ZIP is verified before any write or extraction. All ZIP entry names are traversal-checked before extraction. Smack Central already signs every built skin with the release key. **Deployment gate:** rebuild every hosted official skin ZIP and registry entry before deploying 0.7.440, because the checked-in historical registry contains empty signatures and would now be correctly rejected.

OH SNAP's authenticated skin-push route remains an explicit owner-authoring surface rather than a public/gallery install path. It accepts only a valid OH SNAP API key, now requires inert `manifest.json`, never executes metadata, and path-checks the ZIP before extraction. Because it deliberately deploys owner-authored skin PHP, compromise of an OH SNAP authoring key retains code-deployment impact; key scope/lifecycle belongs in the OH SNAP API audit.