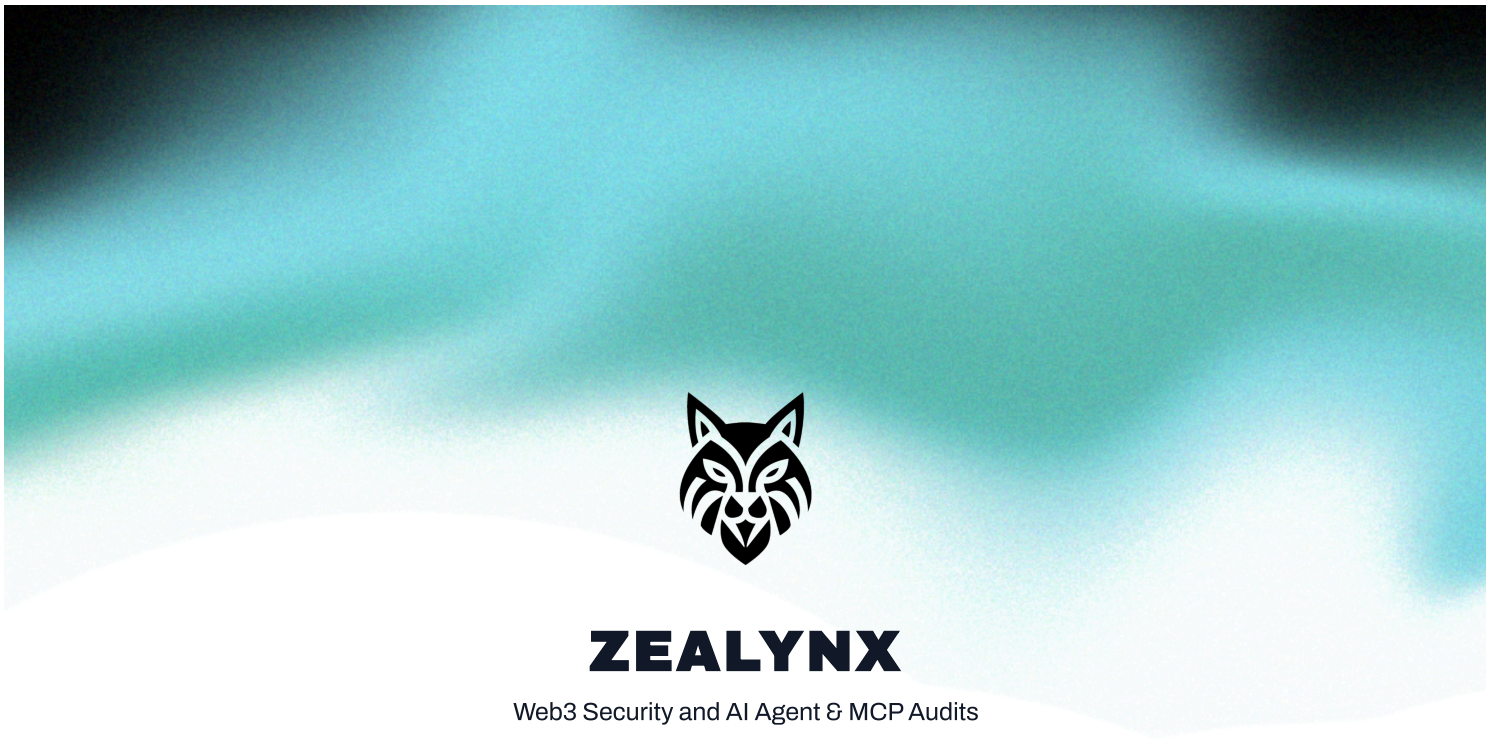

ZEALYNX

Web3 Security and AI Agent & MCP Audits

Vivid Verde Digital LLC
Smart Contract Audit

31 August 2026

Zealynx
contact@zealynx.io

Carlos Vendrell (Bloqarl) — Lead Security Auditor

Contents

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Badger DAO, Ample protocol, Lido, Inverter, Matchain, and Golden Grid.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

Plakxio is a gasless marketplace for football match badges. Buyers and sellers sign orders off-chain and never send a transaction or hold gas; a platform relayer submits every settlement and pays for it. One-of-one Special badges trade separately, through an ascending floor that ratchets on each sale.

3.2 About Vivid Verde Digital LLC

Three contracts on Base. `PlakxioBadge` is an ERC-1155 with four token classes and a per-operator class mask: it is a walled garden, so no holder can move a badge wallet-to-wallet and only a whitelisted settlement contract may initiate a transfer. `PlakxioSettlement` matches EIP-712 signed asks and bids submitted by a relayer, takes payment through an EIP-3009 authorization rather than an allowance, and settles pairs singly or in batches with per-pair gas, calldata and signature caps. `PlakxioSpecialMarket` runs the Special mechanic: a badge is permanently purchasable at or above its stored floor, which ratchets upward on every sale, and the purchase takes no signature from the holder.

The design consequence that shapes the threat model is that makers are offline by construction. They sign, and everything that happens afterwards happens without them.

3.3 Audit Scope

All six Solidity sources under `src/`, 1,391 lines, at tag `audit-v1.1` (commit `5c38893`). Deployment scripts, the test suite and off-chain services were out of scope, except where their behaviour is an assumption of in-scope code.

```
src/PlakxioSettlement.sol
src/PlakxioBadge.sol
src/PlakxioSpecialMarket.sol
src/PlakxioOrderTypes.sol
src/QuoteTreasuryManaged.sol
src/IERC3009.sol
```

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, and **meticulous manual security review.**

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

> **1. Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
> **2. Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
> **3. Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

This was a **two-day Founder Security Sprint** against a frozen tag, carried out by a single auditor. Findings were filed as issues as they were found, at the client's request, so remediation could begin before this report existed.

Where a defect could be demonstrated mechanically, a test was written and then subjected to a falsification step: the defective line was corrected and the unchanged test re-run, so each proof depends on the cited defect rather than on the harness. Every finding at Medium or above is backed by such a test, each paired with a negative control. Two days cannot be exhaustive — effort was concentrated on the payment path, the class-mask authority model and the Special mechanic.

The engagement surfaced **16 findings: 2 High, 6 Medium and 8 Low.** No Critical issues were found. A remediation review against tag `audit-v2` followed: **13 are fixed and verified, and 3 are accepted risk with reasoning.** In two of those three the client argued from their own code and corrected this report — one finding's stated impact and one finding's claim of permanence were overstated, and are recorded as such below.

6.1 The Key Findings

A whitelisted operator could take any holder's badges, and the holder's refusal was unreadable. The badge answered "approved" for any whitelisted operator against every account, and nothing required that operator to be a contract, so a plain address once whitelisted could move every badge of every permitted class out of any wallet without a signature or a payment. A holder who had explicitly refused that operator received a successful transaction and an event while the contract's answer stayed unchanged: the refusal was written to storage and never read.

A resting ask could be silently re-priced after it was signed. Fee parameters sat outside the order and were read from live storage at settlement, and the only bound on the fee was against the execution price rather than the seller's proceeds. An ordinary change to the flat minimum therefore re-priced every resting ask beneath it — a maker who modelled 95% of their sale price could receive a single base unit of it, with the badge delivered and nothing reverting to signal the change. Makers are offline by design and had no transaction available to react with.

A holder could be redeemed on terms they never agreed to. The Special floor was stored as a bare number while the profit-and-commission split that redeems it was re-read live at the next sale. A routine commission increase therefore re-priced everyone who had already bought in, retroactively and all at once, taking most of the profit the mechanic had committed to them — on a sale they had no way to decline, since the purchase takes no signature from the holder.

One mis-scaled administrative call could permanently strand inventory. The opening floor for never-traded Special badges was raise-only, with no upper bound and no inverse. A decimals slip on a six-decimal token would put every unsold Special beyond any buyer's reach at once, and no role — including the administrator who made the change — could undo it. Those holders have no other exit.

6.2 Architectural Security Observations

The weaknesses this sprint surfaced are not a scatter of unrelated defects. The protocol is hardened thoroughly against the adversary its authors expected — a hostile counterparty at the badge transfer, a code-bearing buyer, an abusive batch submitter — and much less against the two they had not modelled: **their own administrator acting normally, and the passage of time between a signature and its settlement.**

The dominant pattern is a value that is load-bearing when a party commits, re-read from live storage when the commitment is executed. The Special floor is stored without the split that redeems it. Fee parameters sit outside the order, so an ask settles under terms its maker never signed. Signature validity is resolved at settlement, so a maker who acquires code between signing and filling can withdraw an order a counterparty has already committed against. The cancel-all epoch lives in whichever settlement contract holds the class bit rather than with the badge it protects. These are one defect in four forms.

A second pattern is irreversibility without a bound. The opening floor is raise-only with no ceiling; a wrong Special mint writes two permanent flags with no reset and no burn; administrator renunciation was a single irreversible call. Each is defensible alone. Together the protocol had several one-way doors and no key, in a walled garden where badges cannot move by any other route.

The third concerns documentation asserting more than the code does. In two places a property was described as closed "by construction" where the code in fact achieves it with a guard. The guards are present and they hold — the risk is a later change that trusts the prose and removes one.

6.3 Security Strengths Observed

Several patterns here are stronger than is typical for a protocol arriving at its first external review, and two are stronger than we usually see at any stage.

The adversarial test suite is the standout. Eight dedicated suites drive a code-bearing or EIP-7702 buyer against the badge transfer — rejecting, reentering, spinning on gas, returning a forged revert, and detonating returndata and signature bombs — alongside hostile sellers. Most teams reach for this class of test after an incident rather than before an audit. It is also why the documentation overstatements above are a documentation problem rather than an exploitable one.

The comments carry reasoning rather than description. Why a fee bound is strictly less-than and what the previous bound permitted. Why revocation must never fail for a reason that does not affect its outcome. Why an emitted mask is the stored value and not the caller's argument, so an indexer cannot record authority the contract does not hold. That made it possible to identify where the code and the stated intent diverge.

Defensive depth is applied consistently rather than decoratively. Checks-effects-interactions is documented as load-bearing. The badge transfer is wrapped so a hostile buyer cannot author a revert that frames the seller, with the seller's balance proved first so a genuine short seller is still attributed correctly. The batch path bounds per-pair gas, total calldata and per-signature length.

The cancel-all epoch advances by an unpredictable jump rather than by one, because a knowable next epoch would let a maker pre-sign into it and have their own cancellation arm the order. Recognising that is not a common level of care.

The self-disclosure was accurate and not defensive. Exposures were declared before the review, described at their real blast radius, and paired with questions inviting challenge. Several of the most useful findings came from testing those stated rationales — possible only because they were written down.

Summary of Findings:

Vulnerability	Severity	Status
[H-1] Any whitelisted operator can take any holder's badges, and the holder's revocation is unreadable	High	Fixed
[H-2] Every freshly claimed Special is purchasable at a global dust floor in the next block	High	Not Fixed
[M-1] The Special floor is stored without the split that redeems it, so a commission raise strips a forced seller's promised profit	Medium	Fixed
[M-2] Below the cancellation threshold, a flatMinFee raise silently reprices resting asks instead of failing them	Medium	Fixed
[M-3] setInitialFloor has no upper bound and no inverse, so one mis-scaled call permanently freezes the never-traded Special inventory	Medium	Fixed
[M-4] mintSpecial never binds the token id to the match id, and has no inverse, so one wrong call destroys two matches' claims	Medium	Fixed
[M-5] Cancellation has no on-chain expression, so an order cancelled in the app stays fillable at its stale price until expiry	Medium	Fixed
[M-6] A payee who cannot receive the quote token permanently strands the badge, because the payout gates the badge transfer	Medium	Not Fixed
[L-1] Percentage fee truncation underpays the treasury	Low	Fixed
[L-2] settleBatch does not bound the decoded item count	Low	Fixed
[L-3] Ordinary badge ids have no supply bound, so the one-per-id semantic the id scheme implies is not contract-enforced	Low	Fixed
[L-4] SelfBuy compares addresses, and the floor ratchets from an unbounded payment, so a holder can price their own badge out of the mechanic permanently	Low	Not Fixed
[L-5] Signature validity is resolved from live state, so a code-bearing maker holds a revocable option on every order they sign	Low	Fixed
[L-6] settleBatch's gas-exhaustion exit emits nothing, so an unattempted pair is indistinguishable from a failed one	Low	Fixed
[L-7] Cancel-all is scoped per settlement contract while the badge is shared, and the deploy script arms both markets by default	Low	Fixed
[L-8] Admin renunciation is a single irreversible call, and the walled garden turns admin loss into permanent illiquidity for every holder	Low	Fixed

7. Audit Findings

7.1 High Severity Findings

[H-01] Any whitelisted operator can take any holder's badges, and the holder's revocation is unreadable

Affected files

`src/PlakxioBadge.sol#L258–L259` and `src/PlakxioBadge.sol#L300–L303` at commit `5c38893`

Description

`isApprovedForAll` returns `true` for any whitelisted operator against **every** account, unconditionally:

```
function isApprovedForAll(address account, address operator) public view
override returns (bool) {
    return isSettlement[operator] || super.isApprovedForAll(account,
operator);
}
```

OpenZeppelin's ERC1155 `safeTransferFrom` gates on `_checkAuthorized(_msgSender(), from)`, which consults exactly this override. The walled-garden gate that runs first checks only the operator's whitelist membership and the token's class bit:

```
function _checkOperatorMayMove(address operator, uint256 id) internal view {
    if (!isSettlement[operator]) revert NotSettlementContract(operator);
    _checkClassPermitted(operator, settlementClassMask[operator], id);
}
```

Neither path checks that `from` consented. `settlementClassMask` restricts **which class** an operator may move; nothing restricts **whose badge**. The security model therefore rests on an unstated property — that the whitelist only ever contains contracts whose own code enforces consent. `PlakxioSettlement` does enforce it, through EIP-712 order signatures. Nothing in `PlakxioBadge` requires a whitelisted address to.

`setSettlementContract` accepts a bare `address` with no `code.length` check and takes effect in the same transaction, so the whitelisted address need not be a contract at all.

Because the left operand of the `||` is already `true`, a holder calling `setApprovalForAll(operator, false)` gets a successful transaction and an `ApprovalForAll` event while the contract's answer stays `true`. The revocation is written to storage and never read.

Vulnerable Scenario:
1. A holder owns badges across classes and pre-emptively calls `setApprovalForAll(grantee, false)`.
2. The admin calls `setSettlementContract(grantee, true, CLASS_MASK_ALL)` where `grantee` is a plain EOA.
3. `isApprovedForAll(holder, grantee)` now returns `true`, overriding step 1.
4. `grantee` calls `safeBatchTransferFrom(holder, grantee, [normalId, premiumId, specialId], [1,1,1], "")`.
Whitelist membership passes, every class bit is set, and `_checkAuthorized` passes on the override.
5. All three badges move. No order signature is presented and no payment is made.
6. The holder cannot move a badge to safety: `safeTransferFrom` from their own address reverts `NotSettlementContract`, and no burn function exists.

Impact

Every badge holder loses every badge of every class named in the mask, without payment and without any signature of theirs. Special one-of-ones are included, so the ascending-floor mechanic is bypassed entirely for those tokens rather than ratcheted. Holders have no exit and no revocation the contract will honour.

Severity is adjusted one tier down from the matrix lookup because the path requires `DEFAULT_ADMIN_ROLE` — a fully-trusted role — to act outside its stated purpose. The trust assumption being violated is that the whitelist admits only consent-enforcing settlement contracts, which `setSettlementContract` does not check and cannot express.

Recommendations

Two options; they are independent and Option B preserves the no-approval seller UX.

Option A — require holder approval as well as the whitelist. The gate becomes a restriction rather than a grant, and a holder can revoke:

```
 function isApprovedForAll(address account, address operator) public view
override returns (bool) {
-    return isSettlement[operator] || super.isApprovedForAll(account,
operator);
+    return isSettlement[operator] && super.isApprovedForAll(account,
```

```
operator);
 }
```

This is a design change rather than a patch: the current flow relies on sellers never needing an approval, so the change fails 82 tests in the existing suite until the seller-side approval is introduced end to end. Weigh it against Option B accordingly.

Option B — delay the grant and require code. Keeps the UX and turns an unobservable one-transaction confiscation into an announced, cancellable one:

```
+uint256 public constant ACTIVATION_DELAY = 2 days;
+mapping(address account => uint256) public settlementActiveFrom;

 if (allowed && classMask == 0) revert EmptyClassMask(settlement);
+if (allowed && settlement.code.length == 0) revert
SettlementNotAContract(settlement);
+if (allowed) settlementActiveFrom[settlement] = block.timestamp +
ACTIVATION_DELAY;
```

```
 function _checkOperatorMayMove(address operator, uint256 id) internal view {
     if (!isSettlement[operator]) revert NotSettlementContract(operator);
+    if (block.timestamp < settlementActiveFrom[operator]) revert
SettlementNotActiveYet(operator);
     _checkClassPermitted(operator, settlementClassMask[operator], id);
 }
```

Revocation must stay instant — it writes `isSettlement = false` and zeroes the mask, and no delay should apply to it.

Independently of either option, `isApprovedForAll` returning `true` for an operator whose class mask excludes the token is misleading to any integrator reading it as the ERC-1155 authority signal. Returning `false` when `settlementClassMask[operator] == 0` costs nothing and removes the worst of that.

Remediation

Three controls landed: `setSettlementContract` refuses an address with no code, a settlement grant waits an activation delay before it may move badges, and a holder's opt-out is now honoured

on the view **and** on both the single and batch transfer paths. A zero class mask no longer reads as approved. Each step of the proof-of-concept was re-run against the fix and is closed.

One note for deployment: the activation delay is an immutable constructor argument with no lower bound, and a value of zero reinstates immediate activation on a contract that cannot be replaced without stranding every badge. A floor in the constructor would make the control code-enforced rather than deployment-dependent. Consent also remains opt-out rather than opt-in, so a holder who never refuses is still exposed to a bad whitelist entry — the residual is admin-gated, and the default admin role is itself now two-step and delayed.

[H-02] Every freshly claimed Special is purchasable at a global dust floor in the next block

Affected files

`src/PlakxioSpecialMarket.sol#L165-L167` and `src/PlakxioBadge.sol#L220-L230` at commit `5c38893`

Description

`currentFloor` returns one global `initialFloor` for every Special badge that has not yet traded:

```
function currentFloor(uint256 tokenId) public view returns (uint256) {
    uint256 floor = _floorOf[tokenId];
    return floor == 0 ? initialFloor : floor;
}
```

`mintSpecial` writes no per-badge price and no claim timestamp, so the market has nothing to gate on, and `buySpecial` has no opening window. From the block in which `SpecialClaimed` is emitted, a one-of-one badge is purchasable by anyone for the deployed `initialFloor` of `500_000` (0.50 USDC), regardless of what that particular badge is worth.

The claimant has no defence available. `buySpecial` takes no seller signature — consent is established at claim time — so they cannot refuse. They cannot raise their own floor: only a completed purchase writes `_floorOf`, and `SelfBuy` blocks the direct route. They cannot move the badge out of the walled garden, and no burn exists.

The mechanic's stated rationale for starting low is that the ascending floor finds the market-clearing price:

The mechanic promises liquidity and fairness — never profit: a rising floor is not rising value, and the auction naturally ends when the badge reaches its market-clearing price.

That rationale does not extend to the claimant. The ratchet is +15% per sale at deployed parameters, so reaching any meaningful price takes tens of forced transfers — and the claimant exits at the first one, at 0.478261 USDC, while the first buyer captures the whole gap between the global floor and the badge's value.

`setInitialFloor` is the only lever and it is global and raise-only, so protecting one high-value badge permanently re-prices every future low-value Special.

Vulnerable Scenario:
1. The issuance service calls `mintSpecial(claimant, id, matchId)`. `SpecialClaimed` is emitted.
2. `_floorOf[id]` is `0`, so `currentFloor(id)` returns `initialFloor` = `500_000`.
3. A bot watching that event calls `buySpecial(id, claimant, 500_000)` in the next block. Class, balance, `SelfBuy` and `BelowFloor` all pass.
4. `commission = 500_000 * 500 / 11_500 = 21_739`. The claimant receives `478_261`; the badge moves to the bot; `_floorOf` becomes `575_000`.
5. Anyone wanting the badge afterwards — the claimant included — must pay the bot at least `575_000`, and the bot captures every subsequent ratchet.

Impact

The claimant of a one-of-one match badge — the user the mechanic exists to reward — is bought out for 0.478261 USDC in the block after the mint, for any badge whose value exceeds that, with no action available to them at any point. The gap between the global floor and the badge's value accrues entirely to whoever polls the event fastest.

The platform also loses: its commission is 5% of a suppressed 0.50 USDC clearing price rather than of the badge's real one, and the ascending auction begins from a number unrelated to the asset.

The loss magnitude tracks the badges' market value, which is outside the audited scope. The mechanism, the absence of any claimant-side defence and the zero preconditions are established below.

Recommendations

Bind the starting price to the badge rather than to a global. Set it where the match is already known — the issuance path — and consult it before the global fallback:

```
+mapping(uint256 tokenId => uint256) public initialFloorOf;
+
+function setInitialFloorFor(uint256 tokenId, uint256 floor) external
onlyRole(DEFAULT_ADMIN_ROLE) {
+    if (floor < initialFloorOf[tokenId]) revert
InitialFloorBelowCurrent(floor, initialFloorOf[tokenId]);
+    initialFloorOf[tokenId] = floor;
+}
```

```
+
 function currentFloor(uint256 tokenId) public view returns (uint256) {
     uint256 floor = _floorOf[tokenId];
-    return floor == 0 ? initialFloor : floor;
+    if (floor != 0) return floor;
+    uint256 opening = initialFloorOf[tokenId];
+    return opening == 0 ? initialFloor : opening;
 }
```

If a per-badge price is not practical at issuance, the alternative is an opening window: record `specialClaimedAt[id] = block.timestamp` in `mintSpecial` and have `buySpecial` refuse a token whose `_floorOf` is still `0` until that timestamp plus a fixed delay has passed. That gives the claimant a bounded period to act without giving them a veto, so the no-seller-signature property of the mechanic is preserved either way.

Remediation

Accepted risk, and the impact stated in this finding was overstated. The client demonstrated, with passing tests, that a buyer purchasing at the opening floor captures the ratchet step rather than the gap to fair value: paying `500,000` leaves the badge takeable at `575,000` and returns `550,000` when it is taken — a profit of `0.05` USDC. Buying cheaply is what makes a position cheap to take back, and overpaying raises the floor and therefore buys retention.

We re-derived that arithmetic and accept the correction. What remains is that the claimant exits at the first rung and every later rung is a sale they are not part of, which the client characterises as a knowing mechanism-design trade rather than a costless property.

7.2 Medium Severity Findings

[M-01] The Special floor is stored without the split that redeems it, so a commission raise strips a forced seller's promised profit

Affected files

`src/PlakxioSpecialMarket.sol#L142–L149` at commit `5c38893`

Description

`buySpecial` writes the ratcheted floor as a bare integer and records nothing about the escalation parameters that produced it:

```
uint256 cBps = commissionBps;
uint256 escalated = BPS_DENOMINATOR + profitBps + cBps;
uint256 commission = (payment * cBps) / escalated;
uint256 newFloor = (payment * escalated) / BPS_DENOMINATOR;

// —— Effects ——
_floorOf[tokenId] = newFloor;
```

`newFloor` is a one-way compression of three inputs into one number. At the next sale the contract needs the split again and re-reads `profitBps`/`commissionBps` live, so a floor written under one split is decomposed under whatever the parameters are then. The comment directly above states the property this is meant to deliver:

the embedded commission is payment × c / (1 + p + c), so a sale at exactly the floor pays the prior owner exactly what they paid × (1 + p).

Nothing in storage enforces it. `_setEscalationParams` validates only the immutable ceilings and imposes no monotonicity in either direction, unlike `_setInitialFloor`, which was deliberately made raise-only. The parameter that gates entry is hardened; the two that decide who gets paid move freely.

The holder cannot avoid the redemption. `buySpecial` takes no seller signature, the badge cannot move wallet-to-wallet, and no burn exists — so a holder is redeemed under terms they never agreed to, on a sale they cannot decline.

No malice is required. A plain commission increase from 5% to its 15% ceiling produces the result below, and the floor is unchanged by it, so nothing observable signals the change.

Vulnerable Scenario:
1. Parameters are the deployed `profitBps = 1000`, `commissionBps = 500`, so `escalated = 11500`.
2. A buyer takes a Special at `payment = 100.00 USDC`. `_floorOf` becomes `115.00`. The mechanic's promise to them at that floor is `100.00 × 1.10 = 110.00`.
3. The admin calls `setEscalationParams(1000, 1500)` — commission to its immutable ceiling. `currentFloor` still returns `115.00`.
4. Anyone calls `buySpecial(id, holder, 115_000_000)` — a sale at exactly the floor.
5. `commission = 115e6 × 1500 / 12500 = 13_800_000`. The holder receives `101_200_000`.

Impact

A forced Special seller receives 101.20 USDC against the 110.00 the mechanic committed to — 88% of their promised profit, routed to the treasury instead. Every Special holder who bought under the previous split is re-priced simultaneously and retroactively by one call, and none of them can decline the sale that realises it.

Documented invariant INV-3 stays green throughout: the floor never decreases, it stops describing the terms the holder bought under.

Recommendations

Persist the split alongside the floor and decompose the next sale with the stored value. Live parameters continue to price the next escalation, which is correct — they should never re-price a commitment already made.

```
 mapping(uint256 tokenId => uint256) private _floorOf;
+mapping(uint256 tokenId => uint256) private _escalatedOf;
+mapping(uint256 tokenId => uint256) private _commissionBpsOf;
```

```
 uint256 cBps = commissionBps;
 uint256 escalated = BPS_DENOMINATOR + profitBps + cBps;
-uint256 commission = (payment * cBps) / escalated;
+uint256 escAtSale = _escalatedOf[tokenId] == 0 ? escalated :
_escalatedOf[tokenId];
+uint256 cAtSale = _escalatedOf[tokenId] == 0 ? cBps :
_commissionBpsOf[tokenId];
+uint256 commission = (payment * cAtSale) / escAtSale;
```

```
 uint256 newFloor = (payment * escalated) / BPS_DENOMINATOR;

 // ── Effects ──
 _floorOf[tokenId] = newFloor;
+_escalatedOf[tokenId] = escalated;
+_commissionBpsOf[tokenId] = cBps;
```

The `== 0` branch covers the never-traded case, where there is no prior split to honour and the live parameters are the right ones to use.

Remediation

The escalation terms are now stored per token, so a holder is redeemed on the terms they bought under rather than the terms in force at the time of the later sale. Verified at `audit-v2`.

A badge that has never traded has no stored terms, so its first sale still uses the live split.
The client accepts that case on the grounds that the claimer received the badge at no cost and the profit promise binds for buyers who paid.

[M-02] Below the cancellation threshold, a flatMinFee raise silently reprices resting asks instead of failing them

Affected files

`src/PlakxioSettlement.sol#L528–L533` and `src/PlakxioSettlement.sol#L569–L572` at commit `5c38893`

Description

The fee parameters sitting outside the order signature is already a documented exposure, and the documented consequence is a **loud** one: raising `flatMinFee` above a resting order's execution price makes the pair fail simulation, and both makers' orders are cancelled. This issue is about the band below that threshold, where the pair does not fail — it settles, and the seller absorbs the difference with nothing reverting.

`_finalize` computes the fee from live storage and checks it only against the buyer's payment:

```
uint256 fee = calculateFee(executionPrice);
// STRICTLY less than, not `>`. The old bound permitted fee == price, ...
if (fee >= executionPrice) revert FeeExceedsPrice(fee, executionPrice);
```

That guard bounds the fee against `executionPrice`, never against the seller's proceeds, so it guarantees the seller one base unit and nothing more. Anywhere `flatMinFee < executionPrice` the settlement completes normally.

The second half is that `FEE_BPS_CEILING` does not bound the realised fee. `calculateFee` takes the maximum of two branches and only the percentage branch is subject to it:

```
return Math.max((executionPrice * feeBps) / BPS_DENOMINATOR, flatMinFee);
```

`flatMinFee` is bounded by `flatMinCeiling`, deployed at `10_000_000` (10.00 USDC). Below an execution price of about 66.67 USDC the flat branch wins, so for the whole cent-value book — which `DeployBase.s.sol` describes as "the trades that are expected to be the most common kind" — the advertised 15% ceiling is not the ceiling that applies.

The scenario below uses `flatMinFee = 500_000`. That is not an extreme value: it is the default this parameter carried before the current `10_000`, per the comment at

`script/DeployBase.s.sol` ("0.01, not the 0.50 this defaulted to"). Reverting it is an ordinary product decision.

Vulnerable Scenario:
1. Deployed parameters: `feeBps = 500`, `flatMinFee = 10_000`, `flatMinCeiling = 10_000_000`.
2. A seller signs an ask at `600_000` (0.60 USDC). At signing, `calculateFee` returns `max(30_000, 10_000) = 30_000`, so they model proceeds of `570_000`. They go offline.
3. The admin calls `setFeeParams(500, 500_000)` — inside the ceiling.
4. The relayer settles the unchanged, still-valid ask at `600_000`. The fee is now `max(30_000, 500_000) = 500_000`, and `500_000 >= 600_000` is false, so the guard passes.
5. The seller receives `100_000` (0.10 USDC), the treasury `500_000`, and the badge is delivered.

Impact

A resting ask maker receives 0.10 USDC where their signature was priced against 0.57 — 82% of their proceeds, transferred to the treasury — while the badge is delivered as normal. Nothing reverts, so the change does not surface as a failed simulation and the maker, who is offline by design, has no signal. Every resting ask priced between the old and new flat minimum is affected at once.

Documented invariant INV-5 holds literally throughout: the seller is paid something. The amount it guarantees is one base unit.

The scenario above is not the worst the ceiling permits. `flatMinCeiling` is 10.00 USDC, so for a resting ask priced one base unit above it the fee becomes the whole price bar one: `FeeExceedsPrice` requires only `fee < executionPrice`, so the guaranteed floor on a seller's proceeds is exactly one quote base unit. A 10.000001 USDC ask signed against proceeds of 9.500001 settles for `0.000001` — 99.99999% redirected to the treasury, with the badge delivered and nothing reverting. The second test below asserts that case.

Recommendations

Two changes. They are not alternatives, and the order matters: the second is what closes the harm, and the first alone does not.

Bind the maker's floor into the signature. The maker's proceeds become a property of what they signed, so a fee change invalidates the orders it would re-price instead of repricing them:

```
struct Order {
    address maker;
```

```
     Side side;
     uint256 tokenId;
     uint256 price;
+    uint256 minProceeds;
     address quote;
     uint256 nonce;
     uint256 expiry;
     uint256 salt;
 }
```

with `ORDER_TYPEHASH` and `hashOrder` extended to match, the JSON artifact at `eip712/PlakxioOrder.json` updated in step, and in `_finalize`:

```
 uint256 fee = calculateFee(executionPrice);
+if (executionPrice - fee < ask.minProceeds) revert
ProceedsBelowSigned(executionPrice - fee, ask.minProceeds);
```

Clamp the realised fee to the immutable ceiling. This is worth doing alongside, because it bounds the exposure of orders signed before the field exists and needs no type-hash change:

```
-if (fee >= executionPrice) revert FeeExceedsPrice(fee, executionPrice);
+if (fee > (executionPrice * FEE_BPS_CEILING) / BPS_DENOMINATOR) revert
FeeExceedsPrice(fee, executionPrice);
```

It is a bound, not a closure. Sweeping `flatMinFee` across its whole range with the clamp applied still reprices the ask in this issue, silently, anywhere the new flat minimum sits between the fee at signing and the ceiling — at `flatMinFee = 30_789` the seller receives `569_211` against the `570_000` they signed against, and nothing reverts:

```
[FAIL: settled below the proceeds the maker signed against: 569211 < 570000;
 counterexample: args=[30789]] testFuzz_M02_fixKeepsSellerWhole(uint256)
```

Applying the clamp on its own therefore turns the loud band louder and leaves the silent band silent. Only the signed floor removes the dependence on a global.

Note the clamp also leaves `calculateFee` itself returning a figure above the ceiling, so any off-chain consumer quoting it should read the clamped value rather than the raw one.

Remediation

Both halves landed, and the realised-fee ceiling **refuses** rather than clamps, which is stronger than the recommendation: it removes the silent band rather than bounding it. `minProceeds` is now a signed field of the order, so a fee change invalidates the orders it would otherwise re-price.

This carries a client-side obligation. The protection is only as good as the value signed: a maker whose client leaves `minProceeds` at zero still settles below what they modelled — now bounded by the realised-fee ceiling to roughly 8% rather than the whole proceeds, but not removed. Sweeping the parameter across its full range confirms the finding is closed outright when a floor is signed, and not when it is left at zero. The order-building client should populate `minProceeds` on every ask and the relayer should refuse an ask that carries zero.

[M-03] setInitialFloor has no upper bound and no inverse, so one mis-scaled call permanently freezes the never-traded Special inventory

Affected files

`src/PlakxioSpecialMarket.sol#L196-L201` and `src/PlakxioSpecialMarket.sol#L165-L167` at commit `5c38893`

Description

`_setInitialFloor` rejects zero and rejects any value below the current one, and imposes no upper bound:

```
function _setInitialFloor(uint256 newInitialFloor) internal {
    if (newInitialFloor == 0) revert ZeroInitialFloor();
    uint256 current = initialFloor;
    if (newInitialFloor < current) revert
InitialFloorBelowCurrent(newInitialFloor, current);
    initialFloor = newInitialFloor;
    emit InitialFloorUpdated(newInitialFloor);
}
```

Every other economic parameter in the codebase is bounded above by an immutable and is freely reversible — `feeBps` by `FEE_BPS_CEILING`, `flatMinFee` by `flatMinCeiling`, `profitBps` and `commissionBps` by their own ceilings. This one has neither.

`currentFloor` returns `initialFloor` for every badge whose `_floorOf` is still `0`, so a raise applies retroactively to the entire never-traded Special inventory at once. There is no other writer of `initialFloor`, no per-token override, and `_floorOf` is written only by a completed `buySpecial` — which can no longer execute once the floor is out of reach. The state is terminal for every affected badge and no actor, including the admin, can undo it.

The monotonicity check exists to stop an admin under-pricing never-traded badges, and its rationale is recorded at L186-L194. That is a recoverable mistake: the badge sells once and the ratchet resumes. The check converts the opposite mistake into an unrecoverable one, and nothing bounds it.

The trigger is an ordinary decimals slip on a 6-decimal token, not malice.

Vulnerable Scenario:
1. `initialFloor` is the deployed `500_000` (0.50 USDC). A Special is claimed and has not traded.

2. The admin intends 0.50 USDC and passes an 18-decimal figure: `setInitialFloor(500_000_000_000_000_000)`.
It is larger than the current value, so L199 accepts it.
3. `currentFloor` now returns 5e17 for that badge and every other never-traded Special.
4. Every `buySpecial` on them reverts `BelowFloor`.
5. `setInitialFloor(500_000)` reverts `InitialFloorBelowCurrent`. There is no other path.

Impact

Every Special badge that has not yet traded becomes permanently unsaleable, and its holder permanently trapped: `buySpecial` is their only exit, the order book rejects Special ids, `PlakxioBadge` forbids wallet-to-wallet transfer, and no burn exists. The platform loses the commission stream on its entire unsold Special inventory. No role can recover the state.

Recommendations

Add an immutable ceiling, mirroring how `flatMinCeiling` bounds `flatMinFee` in the sibling contract. The monotonicity check keeps the edge it was written for; the ceiling closes the one it opened.

```
+uint256 public immutable initialFloorCeiling;
```

```
 ) QuoteTreasuryManaged(quoteCurrency_, treasury_) {
     if (admin == address(0) || address(badge_) == address(0)) revert
ZeroAddress();
     badge = badge_;
+    initialFloorCeiling = initialFloorCeiling_;
```

```
 function _setInitialFloor(uint256 newInitialFloor) internal {
     if (newInitialFloor == 0) revert ZeroInitialFloor();
+    if (newInitialFloor > initialFloorCeiling) revert
AboveCeiling(newInitialFloor, initialFloorCeiling);
     uint256 current = initialFloor;
     if (newInitialFloor < current) revert
InitialFloorBelowCurrent(newInitialFloor, current);
```

`AboveCeiling` already exists and carries the right shape. Size `initialFloorCeiling` against the quote token's decimals at deploy time, the same way `flatMinCeiling` is.

Remediation

`initialFloorCeiling` is immutable and bounds the raise-only floor, so a mis-scaled value is refused at the setter rather than permanently stranding the never-traded inventory. Verified at `audit-v2`.

[M-04] mintSpecial never binds the token id to the match id, and has no inverse, so one wrong call destroys two matches' claims

Affected files

`src/PlakxioBadge.sol#L220–L230` at commit `5c38893`

Description

`mintSpecial` validates the id's range, the match's claim status and the id's mint status, but never that the two correspond:

```
function mintSpecial(address to, uint256 id, uint256 matchId) external
onlyRole(MINTER_ROLE) {
    if (tokenClass(id) != TokenClass.Special) revert NotSpecialTokenId(id);
    if (specialClaimed[matchId]) revert SpecialAlreadyClaimed(matchId);
    if (specialIdMinted[id]) revert SpecialIdAlreadyMinted(id);
    specialClaimed[matchId] = true;
    specialIdMinted[id] = true;
    emit SpecialClaimed(matchId, id, to);
    _mint(to, id, 1, "");
}
```

There is no derivation and no registry tying `id` to `matchId`. Both flags are then written permanently — neither has a setter or a reset anywhere in the contract, and `PlakxioBadge` exposes no burn (OpenZeppelin's `_burn` is internal and not surfaced). `mintSpecial` has no inverse.

The consequence is that a single wrong-id call consumes **two** matches' entitlements: the match named in the call can never be re-issued because its claim flag is set, and the id used can never be issued for the match it actually belongs to because its mint flag is set.

The wrongly-issued badge is also unrecoverable. It cannot be burned, and the walled garden admits no mover other than a whitelisted operator, so the rightful winner's only route to it is to buy it through the Special market at the current floor — paying an unrelated address that keeps the proceeds.

This is the on-chain/off-chain token-id mismatch class that the id ceiling exists to surface loudly. It surfaces, and is then permanent.

Vulnerable Scenario:
1. Match 4711's canonical Special id is `2_000_004_711`. The issuance service resolves a stale record and calls `mintSpecial(wrongAddr, 2_000_009_999, 4711)`.
2. All three checks pass: the id is in the Special range, match 4711 is unclaimed, and id `2_000_009_999` is unminted. Both flags are written and the badge is minted.
3. The error is noticed. `mintSpecial(winnerA, 2_000_004_711, 4711)` reverts `SpecialAlreadyClaimed(4711)`.
4. Issuing `2_000_009_999` for match 9999, the match it belongs to, reverts `SpecialIdAlreadyMinted`.
5. The badge cannot be moved out of `wrongAddr` or destroyed.

Impact

Two matches lose their one-of-one Special entitlement permanently from one call, and the badge that was minted sits with an address that did not earn it, unrecoverable by the platform. The rightful winner's only path to their own prize is to purchase it at the market floor from that address. The on-chain claim registry and the off-chain record are left permanently inconsistent for both matches, with no reconciliation available.

Recommendations

Make the mismatch unrepresentable rather than recoverable. Deriving the id from the match removes the failure mode entirely and makes `specialIdMinted` redundant with `specialClaimed`:

```
 function mintSpecial(address to, uint256 id, uint256 matchId) external
onlyRole(MINTER_ROLE) {
     if (tokenClass(id) != TokenClass.Special) revert NotSpecialTokenId(id);
+    if (id != SPECIAL_ID_START + matchId) revert SpecialIdMatchMismatch(id,
matchId);
     if (specialClaimed[matchId]) revert SpecialAlreadyClaimed(matchId);
```

This requires `matchId < SPECIAL_ID_END − SPECIAL_ID_START` (1e9 matches), which is well above the ~38k/year the id scheme is sized for. Keep `specialIdMinted` as a redundant assertion if you prefer belt-and-braces; it costs one SLOAD and would now be unreachable.

If the id must remain independent of the match, then the registry needs an inverse — a `MINTER_ROLE` function that burns the badge and clears both flags, gated on the token still sitting with its original mint recipient so it can never reach a badge someone has paid for. That is strictly more surface than the derivation above, which is why the derivation is the recommendation.

Remediation

`mintSpecial` now requires the token id to derive from the ordinal, so the id and the entitlement cannot diverge and a single wrong call can no longer consume two matches' claims. Verified at `audit-v2`.

[M-05] Cancellation has no on-chain expression, so an order cancelled in the app stays fillable at its stale price until expiry

Affected files

`src/PlakxioSettlement.sol#L605–L618` and `src/PlakxioSettlement.sol#L284–L305` at commit `5c38893`

Description

`_validateOrder` gates an order on its quote, expiry, nonce epoch, replay flag and signature:

```
if (order.quote != currentQuote) revert QuoteCurrencyMismatch(order.quote,
currentQuote);
digest = orderDigest(order);
if (block.timestamp > order.expiry) revert OrderExpired(digest, order.expiry);
if (order.nonce != nonces[order.maker]) revert StaleNonce(digest, order.nonce,
nonces[order.maker]);
if (settledOrders[digest]) revert OrderAlreadySettled(digest);
```

`settledOrders[digest]` is written only by a completed fill in `_finalize`, and the only maker-callable revocation is `incrementNonce`, which is all-or-nothing by construction — its own comment describes individual cancellation as off-chain and itself as the backstop.

There is therefore no on-chain expression of a single cancelled order. A maker who cancels one order in the app has performed no state change; their signature remains fully valid until `expiry`. They will not use the backstop, because it destroys every other order they hold.

This sits against the stated model of the off-chain layer, which is that admission is a convenience and not a boundary because the contract re-checks everything. Cancellation is the one thing the contract does not re-check, so for this property the off-chain book is the only enforcement.

Submission requires RELAYER_ROLE, so an honest relayer following its own records will not present a cancelled order. What the contract does not have is any way to refuse one if the relayer's records are wrong, its key is compromised, or a stale cache is replayed — and the relayer can also supply the counterparty, since `SelfTrade` only compares the two makers' addresses.

Vulnerable Scenario:
1. A maker signs an ask at 10.00 USDC with a 30-day expiry. The relayer's infrastructure

stores it.
2. The maker cancels it in the app. `nonces[maker]` is unchanged; `settledOrders[digest]` is `false`; the expiry is weeks away.
3. The badge appreciates. The signature is still on-chain-valid at the stale price.
4. The order is submitted against any bid at 10.00 and settles. Every check in `_validateOrder` passes.

Impact

Every order a maker believes they have cancelled remains fillable at its signed price until it expires. The maker loses the difference between that price and the badge's current value, and their only defence is to void their entire open book — which requires them to be online, and which the design assumes they are not. The counterparty and the treasury receive what the maker lost.

Recommendations

Add a per-order revocation that reuses the existing single-fill mapping, so `_validateOrder` needs no new check and the epoch bump becomes the rare escape hatch it is documented as:

```
+error OrderCancelled(bytes32 orderHash);
+
+/// @notice Voids a single order. The on-chain counterpart of an app-side
cancel.
+function cancelOrder(PlakxioOrderTypes.Order calldata order) external {
+    if (msg.sender != order.maker) revert InvalidSignature(orderDigest(order),
order.maker);
+    settledOrders[orderDigest(order)] = true;
+    emit OrderCancelledEvent(orderDigest(order), msg.sender);
+}
```

Reusing `settledOrders` means a cancelled order surfaces as `OrderAlreadySettled` to any existing consumer, which is already a terminal state they handle. If distinguishing the two matters off-chain, use a separate `cancelledOrders` mapping and its own error instead — the extra SLOAD in `_validateOrder` is the only cost.

Users pay no gas in this system, so route the call the same way `incrementNonce` is routed.

Remediation

On-chain cancellation now exists: `cancelOrder` and a batch `cancelOrders`, both permissionless to submit with the maker's signature as the authority, alongside a maximum order lifetime. Keeping submission permissionless is the right trade — role-gating it would let a compromised relayer key destroy every open order, and gasless makers cannot send transactions at all. Verified at `audit-v2`.

[M-06] A payee who cannot receive the quote token permanently strands the badge, because the payout gates the badge transfer

Affected files

`src/PlakxioSpecialMarket.sol#L153-L157` and `src/PlakxioSettlement.sol#L653-L663` at commit `5c38893`

Description

Both trading paths push the payout to their recipients unconditionally, and the badge leg runs only after the push succeeds:

```
// PlakxioSpecialMarket.buySpecial
quote.safeTransferFrom(msg.sender, holder, payment - commission);
if (commission > 0) {
    quote.safeTransferFrom(msg.sender, treasury, commission);
}
badge.safeTransferFrom(holder, msg.sender, tokenId, 1, "");
```

```
// PlakxioSettlement._executePayment
quote.safeTransfer(seller, executionPrice - fee);
if (fee > 0) {
    quote.safeTransfer(treasury, fee);
}
```

The quote currency is a Circle FiatToken, whose `Blacklistable` module reverts any transfer to or from a blacklisted address. A payee who cannot receive therefore does not merely miss a payment — they block the badge transfer that the payment gates.

For a Special holder this is terminal. `buySpecial` is their only exit: the order book rejects Special ids, `PlakxioBadge` admits no wallet-to-wallet transfer, no burn is exposed, and neither trading contract has a rescue function. Their badge can never be sold, moved, or destroyed by anyone, including the admin. `_floorOf` freezes at its last value and the ascending mechanic ends for that token.

The loss is not the blacklisted quote balance. It is a separate asset the blacklist reaches through the payment path.

The ordinary-settlement leg has the same shape with the same permanence — a blacklisted seller cannot sell their badge through any route. The treasury leg has the same shape with a different blast radius: since `flatMinFee` is non-zero at deploy, `fee > 0` on every ordinary settlement, so a blacklisted treasury reverts every settlement and every commission-bearing Special sale at once. That variant is recoverable, in two `setTreasury` calls; the payee variants are not recoverable at all. All three share one fix, which is why they are reported together.

Vulnerable Scenario:
1. A user holds a Special badge and is blacklisted by the token issuer, for reasons unrelated to this protocol.
2. Any buyer calls `buySpecial`. The class, balance, self-buy and floor checks pass and `_floorOf` is written, then the holder payout reverts.
3. Every subsequent buyer, at every price, reverts identically.
4. The holder cannot move the badge: `safeTransferFrom` from their own address reverts `NotSettlementContract`, and no burn exists.

Impact

A blacklisted Special holder permanently loses the badge as an asset — not only its liquidity. A blacklisted ordinary seller is in the same position for their badge. In both cases no role can recover the state. When the blacklisted address is the treasury instead, both markets halt simultaneously for as long as the rotation takes, and per the documented failure policy every matched pair failing during that window costs both makers their orders.

Recommendations

Make the payee legs non-blocking so they can never block the badge leg. The badge transfer, not the payout, is the step with no alternative route.

```
+mapping(address account => uint256) public owed;
+
+function _payOrCredit(IERC20 quote, address from, address to, uint256 amount)
internal {
+    if (amount == 0) return;
+    try quote.transferFrom(from, to, amount) returns (bool ok) {
+        if (ok) return;
+    } catch {}
+    quote.safeTransferFrom(from, address(this), amount);
+    owed[to] += amount;
+}
+
```

```
+function withdrawProceeds(address to) external {
+    uint256 amount = owed[msg.sender];
+    owed[msg.sender] = 0;
+    quoteCurrency.safeTransfer(to, amount);
+}
```

with `buySpecial`'s two pushes and `_executePayment`'s seller and treasury pushes routed through it. `withdrawProceeds` takes a destination so a payee who cannot receive at their current address can still route the funds somewhere that can.

State the consequence for the documented invariant explicitly rather than leaving it implied: this makes the contract hold a balance across transactions, so "platform contracts never hold user funds outside an atomic transaction" needs restating as "never holds funds other than proceeds credited to a named payee and withdrawable only by them". That is a real change to the property and should be a deliberate decision rather than a side effect of the fix.

If holding balances is unacceptable, the narrower alternative for the ordinary path alone is a maker-signed `payoutTo` field in the order, letting a seller direct proceeds to a clean address. It does not cover `buySpecial`, which takes no seller signature, so the Special stranding would remain open.

Remediation

Partially addressed; the recommended escrow was declined, and one element of this finding was overstated. The client declined to credit and hold proceeds on a non-custody basis: balances of that kind would exist by construction only for restricted parties, and a withdraw-to-any-address path is a mechanism for routing value on behalf of one. That is an owner decision and we do not dispute it.

The finding also described the stranding as permanent. It is not — a restriction is reversible, and the badge trades normally once it lifts. Suspended rather than destroyed, and the client has pinned that with a test.

What was built instead is a legibility layer: a restricted buyer or holder is now refused by name and up front, and a single view answers both so a client can disable the action rather than discovering the state by simulation. The token remains the guarantee; the added checks can only make a refusal legible, never permissive.

7.3 Low Severity Findings

[L-01] Percentage fee truncation underpays the treasury

Affected files

`src/PlakxioSettlement.sol#L569–L572` at commit `5c38893`

Description

`calculateFee()` truncates the percentage fee before comparing it with `flatMinFee`:

```
return Math.max((executionPrice * feeBps) / BPS_DENOMINATOR, flatMinFee);
```

Integer division discards the remainder, so the fee actually charged is below the configured percentage whenever `executionPrice * feeBps` is not a multiple of `BPS_DENOMINATOR`. With the deployed `feeBps = 500`, an execution price of `1.000001 USDC` has an exact 5% fee of `50,000.05` base units; the contract charges `50,000`, while the smallest representable fee not below 5% is `50,001`.

At 5%, 19 of every 20 base-unit price residues carry a non-zero remainder, and the discarded fraction reaches `0.95` base units. The treasury therefore receives up to one base unit (`0.000001 USDC`) less than the configured rate on almost every settlement, and `_executePayment` pays that difference to the seller instead — `quote.safeTransfer(seller, executionPrice - fee)` is computed from the same truncated `fee`.

Impact

The treasury is under-paid relative to the configured rate on almost every settlement, and the difference is paid to the seller instead. The shortfall is at most one quote base unit (`0.000001` USDC) per settlement, so the loss is an accounting leak rather than a material one — but it is systematic and always in the same direction.

Recommendations

Round the percentage fee toward its recipient rather than truncating: take the ceiling of `executionPrice * feeBps / BPS_DENOMINATOR` before the `Math.max` against `flatMinFee`.

Keep the multiplication checked. Applying `Math.ceilDiv` to the checked product preserves the existing overflow revert. Replacing the whole expression with a 512-bit helper such as `Math.mulDiv(..., Math.Rounding.Ceil)` computes the same fee but removes that revert, retiring the guard `test_calculateFee_overflowReverts` exists to prove.

Three existing tests encode the truncating formula and need updating with the same rounding direction:

Test	File
`test_fee_boundary_units`	`test/ArithmeticTierAndCapInvariants.t.sol`
`testFuzz_settlement_conservationExact`	`test/ArithmeticTierAndCapInvariants.t.sol`
`testFuzz_calculateFee_isMaxOfBranches`	`test/PlakxioSettlement.t.sol`

Remediation

`calculateFee` now rounds toward the treasury using `Math.ceilDiv` over the **checked** product, so the overflow revert on `executionPrice * feeBps` is preserved — `Math.mulDiv` would have removed it. Verified at `audit-v2`.

[L-02] settleBatch does not bound the decoded item count

Affected files

`src/PlakxioSettlement.sol#L354–L370` at commit `5c38893`

Description

`settleBatch()` bounds total calldata, then allocates and returns `settled` from an independently decoded `items.length`:

```
if (msg.data.length > MAX_BATCH_CALLDATA) {
    revert BatchCalldataTooLarge(msg.data.length, MAX_BATCH_CALLDATA);
}

uint256 n = items.length;
if (n == 0) revert EmptyBatch();
settled = new bool[](n);
```

`MAX_BATCH_CALLDATA` is documented at L144-L159 as the load-bearing guard, on the reasoning that capping total bytes bounds everything the loop can be asked to do; L157 sizes it as "~100 honest pairs (each ~1.2 KB)". That reasoning holds only for canonical ABI encoding. The Solidity decoder accepts a dynamic array whose element offsets point at the same tuple body, so the marginal cost of an item is the 32-byte offset head rather than the ~736-byte tuple. A call carrying two distinct tuple bodies decodes to 3,774 items while measuring exactly 131,072 bytes — roughly 21x the assumed ceiling, with the byte cap respected.

n then drives three costs that `BATCH_GAS_FLOOR` was sized against a far smaller number: the `settled` allocation, the loop bound, and the return encoding. Once n is large enough, the gas EIP-150 retains for the parent after a pair consumes its `SETTLE_ONE_GAS_CAP` share is insufficient to ABI-encode the return. The call then runs out of gas and reverts wholesale, unwinding pairs that had already settled, where the design intends a partial result the relayer resubmits.

The item count is never bounded directly, so this rests on a property of the encoder rather than on a check. `settleBatch` is `onlyRole(RELAYER_ROLE)` and the relayer builds the encoding, and standard ABI encoders emit unique canonical offsets — so maker-supplied data alone does not reach it. What is reported is the missing bound, not an externally driven attack.

Impact

A batch that should return a partial result and let the relayer re-submit the remainder instead reverts wholesale, unwinding pairs that had already settled in the same call. The relayer loses the transaction gas and the makers in the batch lose their settlement turn until it is re-submitted. No funds are stolen and no state is corrupted.

Recommendations

Bound the decoded item count before allocating `settled`, in addition to the existing byte caps rather than instead of them. The byte caps bound what the loop copies; a count cap separately bounds allocation, iteration and return-data cost, and makes all three independent of how the calldata was encoded.

A limit of 100 matches the capacity already documented at L157:

```
+uint256 public constant MAX_BATCH_ITEMS = 100;
+error BatchTooManyItems(uint256 size, uint256 max);

 uint256 n = items.length;
 if (n == 0) revert EmptyBatch();
+if (n > MAX_BATCH_ITEMS) revert BatchTooManyItems(n, MAX_BATCH_ITEMS);
 settled = new bool[](n);
```

Remediation

`MAX_BATCH_ITEMS = 100` is asserted before the `settled` allocation, so the bound no longer depends on the calldata being canonically encoded. Verified at `audit-v2`.

[L-03] Ordinary badge ids have no supply bound, so the one-per-id semantic the id scheme implies is not contract-enforced

Affected files

`src/PlakxioBadge.sol#L201–L204` and `src/PlakxioBadge.sol#L220–L230` at commit `5c38893`

Description

`mint` validates the id's class and never the amount:

```
function mint(address to, uint256 id, uint256 amount, bytes calldata data)
external onlyRole(MINTER_ROLE) {
    _checkMintableId(id);
    _mint(to, id, amount, data);
}
```

`_checkMintableId` rejects only Special ids, so any ordinary id can be minted in any quantity, any number of times. There is no per-id supply registry and no `totalSupply`, so nothing on chain records or bounds how many units of a given id exist.

`mintSpecial` shows the opposite decision was taken where it was judged to matter: it hard-codes amount `1` and enforces one-of-one through `specialClaimed[matchId]` and `specialIdMinted[id]`. The id scheme itself implies the same one-per-id semantic for ordinary tiers — ids encode a tier and an ordinal, and the documented capacity model is one badge per match per tier.

The consequence is that the scarcity of an ordinary badge is a property of issuance-service behaviour rather than of the contract, and an off-chain indexer cannot detect a divergence from contract state because none is recorded. Settlement is unaffected: `SellerMissingBadge` checks `< 1` and each order fills once, so N units correctly support N independent asks.

Reported as the missing invariant rather than as an attack: reaching it requires `MINTER_ROLE`, and an issuer over-issuing is the issuer acting against its own users rather than an external party acting against them.

Impact

The scarcity of an ordinary badge is a property of issuance-service behaviour rather than of the contract, and nothing on chain records or bounds how many units of a given id exist. An off-chain

indexer cannot detect a divergence, because there is no contract state to diverge from. Settlement is unaffected: each order fills once, so N units correctly support N independent asks.

Recommendations

Mirror the Special path for ordinary ids:

```
+mapping(uint256 tokenId => bool) public ordinaryIdMinted;
+error OrdinaryIdAlreadyMinted(uint256 id);

 function _checkMintableId(uint256 id) internal pure {
     if (tokenClass(id) == TokenClass.Special) revert
SpecialTokenIdForbidden(id);
 }
```

with the flag set in `mint` and each `mintBatch` element, and the check raising `OrdinaryIdAlreadyMinted`. Note `_checkMintableId` is `pure` and would need to become `view`.

If a tier legitimately needs multiple units, store a per-id `maxSupply` at first mint and check subsequent mints against it instead. Either way the invariant becomes contract-enforced and an indexer can verify it.

Separately, `mint(to, id, 0, "")` currently succeeds and emits `TransferSingle` with a zero value, which an indexer treating any transfer event as an ownership grant would record as a badge that does not exist. Rejecting a zero amount closes that.

Remediation

A supply cap is established on first mint and re-asserted afterwards, with a per-id minted total and every mint path routed through one recorder, so the one-per-id semantic the id scheme implies is now enforced by the contract. Verified at `audit-v2`.

[L-04] SelfBuy compares addresses, and the floor ratchets from an unbounded payment, so a holder can price their own badge out of the mechanic permanently

Affected files

`src/PlakxioSpecialMarket.sol#L134–L149` at commit `5c38893`

Description

`SelfBuy` compares addresses, not economic parties:

```
if (msg.sender == holder) revert SelfBuy(msg.sender);
```

and `newFloor` derives from a caller-supplied `payment` with no bound relative to the current floor:

```
uint256 newFloor = (payment * escalated) / BPS_DENOMINATOR;
```

A holder using a second address they control passes the guard. The badge and almost all of the payment return to them in the same transaction, so the real cost of driving the floor to any value F is only the commission: `commissionBps / (profitBps + commissionBps)` of F, which is one third at deployed parameters. Working capital is recoverable within the call and so can be borrowed.

`_floorOf` is monotone with no reset path anywhere in the contract, so the result is permanent. Above a floor no rational buyer will meet, the contract's stated property — that every Special badge is permanently open to purchase at or above its floor — no longer holds for that token, and the platform's commission stream on it ends.

Documented invariant INV-3 stays green throughout: the floor never decreases. It stops describing a price anyone would pay.

`commissionBps` has an immutable ceiling but no floor, so at `commissionBps = 0` the round trip costs only gas.

Rated Low because the party who pays is the badge's own holder and the third-party harm is the loss of an option to buy rather than a loss of funds. The same shape is reachable by accident: a buyer passing an 18-decimal figure to a 6-decimal quote token irreversibly prices their own badge out, with no recovery for anyone.

Impact

A holder can drive their own badge's floor to a level no rational buyer will meet, for a cost of roughly one third of the target price, permanently ending the ascending mechanic for that token and the platform's commission stream on it. The party who pays is the badge's own holder, and the third-party harm is the loss of an option to buy rather than a loss of funds. The same terminal state is reachable by accident, through a buyer passing an 18-decimal figure to a six-decimal quote token.

Recommendations

Clamp the value the floor may be derived from, leaving the seller's proceeds untouched so overpaying-as-defence still works up to the cap:

```
+uint256 public immutable maxRatchetBps; // e.g. 30_000 — one sale may at most
triple the floor
```

```
-uint256 newFloor = (payment * escalated) / BPS_DENOMINATOR;
+uint256 cappedBasis = payment > (floor * maxRatchetBps) / BPS_DENOMINATOR
+    ? (floor * maxRatchetBps) / BPS_DENOMINATOR
+    : payment;
+uint256 newFloor = (cappedBasis * escalated) / BPS_DENOMINATOR;
```

`floor` is already loaded at L139. The seller still receives `payment - commission` in full; only the stored floor is bounded, so an honest overpay is still rewarded and an unbounded one-transaction self-lock becomes unreachable.

Remediation

Accepted risk. The escape is priced at roughly 3.8% of the floor being set, and the client's position is that a self-buy is economically identical to any other buyer's while revenue is ecosystem-wide rather than per-badge. That is consistent with the severity assigned here: the party who pays is the badge's own holder. The residual the client identifies — a buyer fat-fingering their own payment — is a client-side input-validation concern, since any on-chain cap would be arbitrary.

[L-05] Signature validity is resolved from live state, so a code-bearing maker holds a revocable option on every order they sign

Affected files

`src/PlakxioSettlement.sol#L605–L618` at commit `5c38893`

Description

`_validateOrder` resolves a maker's signature at settlement time through `SignatureChecker.isValidSignatureNow`:

```
if (!SignatureChecker.isValidSignatureNow(order.maker, digest, signature)) {
    revert InvalidSignature(digest, order.maker);
}
```

OpenZeppelin selects the verification path from live state — `signer.code.length == 0` chooses ECDSA, otherwise ERC-1271 — and the library's own header states the consequence:

> Unlike ECDSA signatures, contract signatures are revocable, and the outcome of this function can thus change through time. It could return true at block N and false at block N+1 (or the opposite).

The `Order` struct carries no field pinning which scheme the maker committed to, so for a code-bearing maker a signature is not a commitment but a standing offer they may withdraw at the moment of settlement. A contract maker whose `isValidSignature` reads a flag it owns can advertise at the top of the book, let a counterparty be matched, and flip the flag — one SSTORE — so the pair fails. The counterparty cannot cancel a matched order and is offline by design.

An EOA maker reaches the same position by submitting an EIP-7702 delegation, which makes `code.length` non-zero and invalidates all of their outstanding ECDSA signatures at once.

Rated Low because the running system's makers are reported as MPC EOAs with plain 65-byte ECDSA signatures, so the ERC-1271 path is contract-supported but unexercised today. It stops being latent if smart accounts are adopted, which is a documented pre-mainnet decision.

Impact

A code-bearing maker holds a revocable option on every order they have signed: they can advertise at the top of the book, let a counterparty be matched, and invalidate the signature at the moment of settlement. The counterparty cannot cancel a matched order and is offline by design, so they carry the failure. An EOA maker reaches the same position by submitting an EIP-7702 delegation, which invalidates all of their outstanding signatures at once.

Recommendations

Bind the validation scheme into the signature so it cannot change after signing:

```
 struct Order {
     address maker;
     Side side;
+    bool makerIsContract;
     uint256 tokenId;
     ...
 }
```

with `ORDER_TYPEHASH`, `hashOrder` and the JSON artifact updated to match, and in `_validateOrder`:

```
-if (!SignatureChecker.isValidSignatureNow(order.maker, digest, signature)) {
+bool ok = order.makerIsContract
+    ? SignatureChecker.isValidERC1271SignatureNow(order.maker, digest,
signature)
+    : ECDSA.recover(digest, signature) == order.maker;
+if (!ok) {
     revert InvalidSignature(digest, order.maker);
 }
```

The `false` branch removes revocability for the entire EOA maker population, which is the documented production case, and keeps working for a 7702-delegated EOA because `ecrecover` still resolves. The `true` branch leaves genuine contract makers on the revocable path, which is inherent to ERC-1271 and cannot be closed at this layer — bounding it there is an off-chain reputation question rather than a contract one.

Remediation

The verification scheme is now a signed field of the order, so it is fixed at signing and cannot be swapped underneath an order a counterparty has already committed against. Verified at `audit-v2`.

[L-06] settleBatch's gas-exhaustion exit emits nothing, so an unattempted pair is indistinguishable from a failed one

Affected files

`src/PlakxioSettlement.sol#L374–L379` and `src/PlakxioSettlement.sol#L186–L196` at commit `5c38893`

Description

`settleBatch` has three per-index outcomes and only two of them write to the log stream.

The oversized-signature skip and the failed self-call both emit `SettlementSkipped`. The gas-exhaustion exit does not:

```
for (uint256 i = 0; i < n; ++i) {
    if (gasleft() < SETTLE_ONE_GAS_CAP + BATCH_GAS_FLOOR) break;
```

`settled` is a return value, and `settleBatch` is called by a relayer EOA in a transaction, so the return data is not available to the submitter — logs are the only observable channel. An index that was never attempted therefore produces no `Settled` and no `SettlementSkipped`, which is indistinguishable from an index that was attempted and failed silently, or from a dropped log.

`Settled` also carries no batch index:

```
event Settled(
    bytes32 indexed askHash,
    bytes32 indexed bidHash,
    address seller,
    ...
);
```

so resolving which positions succeeded requires recomputing every EIP-712 digest in the batch and matching them against unindexed events.

The fail-safe direction is correct — re-submitting a pair that did settle reverts `OrderAlreadySettled` — so this is an observability gap rather than a double-settlement risk.

It matters because the off-chain layer decides whose order to cancel from these signals, and treating absence as failure cancels the orders of every maker in the untouched tail.

Impact

An index that was never attempted is indistinguishable from one that was attempted and failed. Because the off-chain layer decides whose order to cancel from these signals, treating absence as failure cancels the orders of every maker in the untouched tail of the batch. No double settlement is possible — re-submitting a settled pair reverts — so the exposure is observability rather than custody.

Recommendations

Emit the stop point, and index the successes:

```
+event BatchStopped(uint256 firstUnprocessedIndex, uint256 gasRemaining);
```

```
-if (gasleft() < SETTLE_ONE_GAS_CAP + BATCH_GAS_FLOOR) break;
+if (gasleft() < SETTLE_ONE_GAS_CAP + BATCH_GAS_FLOOR) {
+    emit BatchStopped(i, gasleft());
+    break;
+}
```

```
 event Settled(
+    uint256 indexed index,
     bytes32 indexed askHash,
     bytes32 indexed bidHash,
```

Adding a fourth indexed parameter is not possible, so if `index` should be indexed, demote one of the existing three. `bidHash` is the least useful as a topic given `askHash` already identifies the pair.

Remediation

The gas-exhaustion exit now emits its stop point, and the settlement event leads with the batch index, so a pair is attributed from its own log rather than by counting. The off-chain dual-decode that carried the migration has been retired. Verified at `audit-v2`.

[L-07] Cancel-all is scoped per settlement contract while the badge is shared, and the deploy script arms both markets by default

Affected files

`src/PlakxioSettlement.sol#L180–L181` and `script/DeployBase.s.sol` at commit `5c38893`

Description

`nonces` is per-settlement-contract storage:

```
/// @notice Per-user order-cancellation epoch; bumping voids all open orders.
mapping(address maker => uint256) public nonces;
```

while the badge, and a maker's understanding of "cancel everything I have open", are shared across every settlement contract whitelisted for that badge. `incrementNonce()` on one market performs no write the other market's `_validateOrder` can observe, so a maker who cancels through the app — which pins to the dollar market — has cancelled nothing on the euro book, after being told the cancellation succeeded.

Signature replay across the two markets is genuinely closed: the EIP-712 domain binds `verifyingContract`, and `_validateOrder` independently requires `order.quote` to equal the market's immutable `quoteCurrency`. Cancellation is the property that does not carry across.

This is not reachable on the frozen deployment, where the euro market is de-authorized — its badge `settlementClassMask` is 0 and its `RELAYER_ROLE` is revoked. What is reported is that the deployment script arms both whenever the euro market is configured. `EURC_ADDRESS` is optional and the script says so, but when it is set — as it was for the frozen deployment — the script whitelists the second market with `CLASS_MASK_ORDINARY` and grants it `RELAYER_ROLE` in the same run. The de-authorized state is therefore a manual post-deploy step that nothing in the code reproduces, and a fresh deployment carrying an `EURC_ADDRESS` has two live books and one cancel-all that covers one of them.

Impact

A maker who cancels everything through the application has cancelled nothing on the sibling market, after being told the cancellation succeeded. Their orders there remain fillable at their signed prices until expiry. Signature replay across the two markets is separately closed; it is cancellation that does not carry across.

Recommendations

Two independent changes; the second is worth doing whether or not the first is.

Share the epoch through the contract both markets already trust. The badge is the common dependency:

```
+mapping(address maker => uint256) public orderEpoch;
+event OrderEpochIncremented(address indexed maker, uint256 newEpoch);
+
+function incrementOrderEpoch(uint256 newEpoch) external {
+    if (newEpoch <= orderEpoch[msg.sender]) revert
EpochNotAdvancing(newEpoch);
+    orderEpoch[msg.sender] = newEpoch;
+    emit OrderEpochIncremented(msg.sender, newEpoch);
+}
```

on `PlakxioBadge`, with `PlakxioSettlement._validateOrder` comparing `order.nonce` against `badge.orderEpoch(order.maker)` instead of local storage, and `incrementNonce` computing the jump as it does today and forwarding it. One bump then voids a maker's orders on every market settling that badge.

Deploy the second market de-authorized. Have the script skip the whitelist and the `RELAYER_ROLE` grant for any market beyond the primary, so arming it is an explicit later action rather than the default that must be manually undone.

Remediation

The second market is removed from the deployment scripts rather than defaulted off, so a fresh deployment can no longer bring up a book that cancel-all does not cover. Verified at `audit-v2`.

[L-08] Admin renunciation is a single irreversible call, and the walled garden turns admin loss into permanent illiquidity for every holder

Affected files

`src/PlakxioBadge.sol#L17` and `src/QuoteTreasuryManaged.sol#L27` at commit `5c38893`

Description

All three contracts inherit OpenZeppelin `AccessControl` unmodified — `PlakxioBadge` directly, and `PlakxioSettlement` and `PlakxioSpecialMarket` through their shared `QuoteTreasuryManaged` base. `DEFAULT_ADMIN_ROLE` is its own admin, `grantRole` accepts any address with no acceptance step, and `renounceRole` executes in one call with only a self-confirmation:

```
function renounceRole(bytes32 role, address callerConfirmation) public virtual
{
    if (callerConfirmation != _msgSender()) {
        revert AccessControlBadConfirmation();
    }
    _revokeRole(role, callerConfirmation);
}
```

OpenZeppelin's own `AccessControlDefaultAdminRules` — two-step and delayed, and recommended in a comment in the very file being inherited — is available and unused.

The generic form of this is well known. What makes it worth raising here is what the walled garden does to the consequence. A handover that grants to a mistyped address and then renounces leaves `PlakxioBadge` with no reachable admin, so `setSettlementContract` can never be called again and the whitelist plus every class mask freeze permanently. Because the badge admits no wallet-to-wallet transfer and exposes no burn, every badge in every wallet becomes permanently immobile except through whichever settlement contract happened to be whitelisted at that moment — and the settlement contract has already been redeployed several times over this epic, so a replacement could never be whitelisted and a superseded one never revoked.

The same call on the other two contracts freezes `feeBps`, `flatMinFee`, the escalation parameters, `initialFloor` and `treasury`. Freezing `treasury` is the sharper one: it is the recovery lever for a treasury that can no longer receive the quote token, so losing it turns a bounded outage into a permanent one.

Impact

A handover that grants to a mistyped address and then renounces leaves the badge with no reachable admin. Because the badge admits no wallet-to-wallet transfer and exposes no burn, every badge in every wallet becomes permanently immobile except through whichever settlement contract happened to be whitelisted at that moment. The same call on the other two contracts freezes the fee parameters, the escalation split, the floor and the treasury address — the last of which is the recovery lever for a treasury that can no longer receive the quote token.

Recommendations

Replace `AccessControl` with `AccessControlDefaultAdminRules` on all three contracts:

```
-import {AccessControl} from
"@openzeppelin/contracts/access/AccessControl.sol";
+import {AccessControlDefaultAdminRules} from
"@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol";
```

```
-contract PlakxioBadge is ERC1155, AccessControl {
+contract PlakxioBadge is ERC1155, AccessControlDefaultAdminRules {
```

with the constructor taking an `initialDelay` and the initial admin, replacing the current `_grantRole(DEFAULT_ADMIN_ROLE, admin)`. This makes the default-admin transfer two-step and delayed and disables bare `renounceRole` for that role. `supportsInterface` already overrides both parents and needs its override list updated to match.

Remediation

All three contracts now use OpenZeppelin's two-step, delayed default-admin rules, so the handover cannot complete in a single call and a mistyped grant is recoverable within the window. Verified at `audit-v2`.