

ZEALYNX

Web3 Security and AI Agent & MCP Audits

Vivid Verde Digital LLC

Generation D Mitigation Review

reviewed code `bcf18bb29265`

21 October 2026

Zealynx

contact@zealynx.io

Carlos Vendrell (Bloqarl) — Lead Security Auditor

Contents

1. About Zealynx

2. Disclaimer

3. Overview

3.1 Project Summary

3.2 About Vivid Verde Digital LLC

3.3 Audit Scope and Reference History

4. Audit Methodology

5. Severity Classification

6. Executive Summary

6.1 The Key Findings

6.2 Architectural Security Observations

6.3 Security Strengths Observed

7. Audit Findings

7.1 Medium Severity Findings

7.2 Low Severity Findings

7.3 Informational Severity Findings

8. Evidence and Limitations

Appendix A — August Sprint Carryover Re-verified

Appendix B — Identifier Mapping and Status Ledger

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.

This document is a mitigation review. It reports whether the remediations for previously reported findings hold in a specific, frozen revision of the code. It is not a new audit of that revision, and it is not an approval of any deployment.

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 under 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.

3.3 Audit Scope

This review covers the remediations for the findings of the 18 September 2026 full audit and the five findings raised in the October mitigation re-review, assessed against one frozen revision. The reference history below is immutable: each commit identifies exactly what a given result was established against.

Reference	Commit	What it is
`audit-v3`	`d8fa452dc423d00760d3159f290d82b7cc5e1857`	Original full-audit baseline. The 19 findings of the 18 September report were raised against this tree.

Reference	Commit	What it is
`audit-v3-rereview-handoff`	`f020448976eb7c4895e1626ef3228876cff2b7d2`	The tree at which the October re-review verified 17 of those remediations and raised the five new findings.
`generation-d`	**`bcf18bb2926589bb8538fa8c322eabc871ff2801`**	**The reviewed code for this document.** Frozen candidate, committed 16 October 2026. Seven finding dispositions, the sprint carryover and the additional generation-specific checks were established here. Seventeen earlier dispositions are carried from the prior re-review tag, as recorded in Appendix B.
Documentation provenance	`ecdd95decdf993e31b651bef95bd1f647f9bee57`	The client's completed re-review addendum and Generation D live-verification record. Cited for documentation and deployment history only — not a second reviewed code revision. The comparison from the frozen commit to this one shows no change under `src/`, `script/` or `test/`.

In scope at `generation-d`: all seven Solidity sources under `src/` and the deployment, upgrade and retirement scripts under `script/`, restricted to the remediations under review and to the access-control and retirement changes the generation introduced.

```
src/PlakxioBadge.sol
src/PlakxioSettlement.sol
src/PlakxioSpecialMarket.sol
src/QuoteTreasuryManaged.sol
src/PlakxioOrderTypes.sol
src/IERC3009.sol
src/IBlacklistable.sol
script/network.sh
script/DeployBase.s.sol
script/AdminHandover.sol
script/deploy_base.sh
script/upgrade_market.sh
script/retire_superseded.sh
script/post_retire_check.sh
```

Out of scope, and not reviewed: Base mainnet, which is not deployed; the off-chain application, relayer, monitoring and key-custody arrangements; and any claim about those made by the client, which this report labels as a client assertion wherever it appears.

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.

In the latest pass, seven full-audit/re-review findings and the sprint carryover were assessed against the frozen commit, together with the generation-specific access-control and retirement changes. Seventeen earlier remediation dispositions are carried from the prior re-review at the commit recorded in Appendix B; they were not re-verified in this pass. Where a remediation's own claim could be tested, we wrote our own test rather than relying on the client's — including an adversarial test of the one arithmetic claim we found hardest to accept on its face. Where a result was observable on chain we read it from two independent providers at a single pinned block, and we compared the deployed runtime bytecode against our own compilation of the frozen source.

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 document closes the remediation cycle for the Plakxio engagement against the frozen `generation-d` candidate (`bcf18bb29265`). It accounts for **24 findings**: the 19 raised in the 18 September full audit, and 5 raised in the October mitigation re-review. One carried finding from the earlier Founder Security Sprint was also re-verified and is reported in Appendix A, outside these totals.

20 of the 24 are fixed without the named residuals below. The remaining four have qualified dispositions: two are fixed with a bounded residual retained (L-01, M-11), one is partly addressed with its exposure retained by design (M-12), and one is resolved by the agreed documentation route rather than by a code change (L-10).

The strongest single result is independent of any claim made to us. We compiled the frozen source and compared it against the deployed runtime bytecode of all three contracts on Base Sepolia: the executable code is identical once compiler-recorded immutables are masked, differing only in the trailing metadata region. The results below therefore describe code that is actually running, not a tree we were told about.

This report states a bounded conclusion. We found no blocker in the reviewed code at this revision. That is not clearance for Base mainnet, which is not deployed and was not reviewed, and it is not verification of the off-chain application, monitoring or custody arrangements, which remain client assertions throughout.

6.1 The Key Findings

- **The condition we attached to the highest-priority item is met.** The activation delay's interim 60-second floor is replaced by a six-hour minimum and a 365-day maximum, refused at construction, with 72 hours enforced for production deployment by the scripts and re-read from the chain before promotion.
- **The one remaining partly-addressed finding is a retained design exposure, not an oversight.** A Special holder still cannot refuse, move or burn to escape a hostile Special-class operator approved by a compromised administrator Safe. We consider the rationale supportable and the exposure is documented, not eliminated.
- **Two residuals survive their fixes and are bounded.** A maker can predict their own cancel-all nonce roughly one block ahead (self-exposure only), and a privileged first write can still set a dust floor on a never-traded Special, bounded by the configured opening price.

- **Our own adversarial test of the ratchet's hardest claim held.** Three hundred writes, including the reset-and-repeat sequence, could not erode a holder's promised proceeds beyond the single specified base unit.
- **Two extended retirement harnesses did not finish.** Their results are not counted as passes anywhere in this report; the retirement dispositions rest on cited code inspection instead.

6.2 Architectural Security Observations

The pattern that produced the five re-review findings is worth stating because it is the pattern a mitigation cycle exists to catch: **four of the five were introduced by remediations for earlier findings.** The class-aware refusal that closed M-01 created the Special exemption now reported as M-12 and the misleading success signal reported as L-09; the stored-terms read that closed L-04 changed a public signature with no overload, reported as L-10; and the floor registry that answered an earlier finding moved the ratchet to a contract that did not enforce it, reported as M-11.

None of these were regressions in the ordinary sense. Each was a correct fix whose consequence landed somewhere the original finding did not look. That is an argument for the re-review step itself rather than against the remediation work, and it is why this report carries the lineage explicitly.

A second pattern is **the guard that is correct only for the author's own caller.** The floor registry was safe because the client's own market happened to ratchet by construction, so no test of theirs could reach the write that breaks it; the treasury guard was correct in one ordering and absent in the reverse. Both are now enforced where the value lives rather than where the well-behaved caller happens to be, which is the right place.

A third, visible in the generation's new work, is **fail-closed as a deliberate choice.** The treasury probe refuses an underfunded grant rather than skipping its own check; the per-badge authority view refuses rather than answering a misleading `false`; the retirement wrapper now exits non-zero when it absorbed warnings; and a failed on-chain probe refuses a promotion rather than being reasoned from. Each of these was the specific thing we asked for, and each is implemented as a refusal rather than a best effort.

6.3 Security Strengths Observed

The client's own adversarial rounds found things we would otherwise have reported. In three separate cases the remediation we reviewed was the *second* cut, because the client's red team had already broken the first: a zero-padded value that walked past the production delay gate, a gas-starved authority view that answered a misleading `false`, and a rounding allowance that renewed on every write and could be walked down. Each is documented with the failure that motivated the change rather than presented as having been right the first time.

Disclosure remained accurate in the direction that costs the discloser. Where a control's benefit reaches only to a boundary, the client said so — the external monitor's worst-case gap reaching the six-hour floor rather than clearing it — and where a residual survived a fix, it was named in the contract's own documentation rather than left for us to find. The one case we tested hardest, the non-accumulating rounding allowance, held exactly as described.

A withdrawn control was recorded rather than erased. A stored-floor ceiling was added, built upon, and then removed as contrary to the product rule that a Special is always for sale at any price above its floor. The removal, its reason and the prose it left behind are written down.

Precision improved under challenge rather than defensively. On the one finding where our own earlier wording was tighter than the facts supported, the client's revised comment was *more* conservative than ours, not less.

Summary of Findings:

Vulnerability	Severity	Status
[M-01] Special holders can pre-bank an opt-out while the market is unlisted or revoked and escape the ascending auction	Medium	Fixed
[M-02] At feeBps == FEE_BPS_CEILING, opposite rounding makes 19 of every 20 prices unsettleable	Medium	Fixed
[M-03] Upgrade scripts leave DEFAULT_ADMIN on the broadcast key	Medium	Fixed
[M-04] Retiring the old market before its replacement activates halts trading	Medium	Fixed
[M-05] Upgrade scripts can register markets with an invalid quote token	Medium	Fixed
[M-06] Retirement can leave historical market authority active	Medium	Fixed
[M-07] Deployment artifacts are not bound to the configured RPC chain	Medium	Fixed
[M-08] Concurrent deployments can promote another run's unverified artifact	Medium	Fixed
[M-09] Base deployment silently retains DEFAULT_ADMIN when ADMIN_ADDRESS is unset	Medium	Fixed
[M-10] Retirement can deauthorize the replacement while preserving the superseded market	Medium	Fixed
[M-11] The Special floor registry enforces no monotonicity, so a whitelisted Special-class operator can write the floor downward	Medium	Fixed · residual
[M-12] A holder has no on-chain way to refuse an operator for their Special badges, and the delay that replaces it is floored at 60 seconds	Medium	Partly addressed

Vulnerability	Severity	Status
[L-01] Gasless makers cannot invalidate unenumerated orders	Low	Fixed · residual
[L-02] A narrowed class mask is misreported as a buyer rejection	Low	Fixed
[L-03] The gas model documents transaction paths the contracts do not have	Low	Fixed
[L-04] purchaseBlocked can miss a blocked treasury when a badge uses stored commission	Low	Fixed
[L-05] Gasless holders cannot submit the operator opt-out required by the trust model	Low	Fixed
[L-06] Setting a market as its own treasury permanently traps revenue	Low	Fixed
[L-07] Retirement reports success before permission changes are durable on Base	Low	Fixed
[L-08] Readiness views omit a blacklisted market from mandatory payment parties	Low	Fixed
[L-09] The holder's refusal reports success for Special badges it does not bind	Low	Fixed
[L-10] purchaseBlocked changed signature with no overload, breaking off-chain callers silently	Low	Addressed · docs
[L-11] Nothing enforces the 72-hour activation delay at a production deploy: an env default, a 60s floor, and a self-skipping assertion	Low	Fixed
[I-01] StressLive reports success when every settlement item fails	Informational	Fixed

Fixed — the remediation holds at the commit recorded in Appendix B. **Fixed · residual** — the remediation holds and a named, bounded residual is retained and documented. **Partly addressed** — the stated condition is implemented and a named exposure is retained by design. **Addressed · docs** — closed by the agreed documentation route with no code change.

7. Audit Findings

7.1 Medium Severity Findings

[M-01] Special holders can pre-bank an opt-out while the market is unlisted or revoked and escape the ascending auction

Fixed

Tracker: issue #17 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

src/PlakxioBadge.sol#L364-L403

src/PlakxioBadge.sol#L530-L637

Description

`setSettlementOptOut` rejects an opt-out from the Special market only when the operator's current class mask includes `CLASS_MASK_SPECIAL`. The mask is zero before whitelisting and after revocation, so a holder can store the opt-out during either window. Assigning the Special mask later does not clear or revalidate it.

The badge transfer paths subsequently enforce the stored flag without considering the token class. The Special market therefore cannot transfer that holder's badge, even though Special badges are intended to remain purchasable through the ascending auction.

Vulnerable Scenario:

1. A holder predicts the future Special market address, or waits until the current market is revoked.
2. While the address has a zero class mask, the holder calls `setSettlementOptOut(market, true)`.
3. The administrator grants or restores the market's Special class permission.
4. A buyer submits a valid purchase, but the badge transfer enforces the pre-banked opt-out and the transaction reverts.

Impact

A holder can remove a Special badge from the forced-sale mechanism at no cost, preventing purchases and the associated treasury commissions until the holder voluntarily clears the flag.

PoC

Save the following as `test/poc/OptOutSpecialEscape.t.sol` and run it against commit `d8fa452dc423d00760d3159f290d82b7cc5e1857`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";
import {PlakxioSpecialMarket} from "../../src/PlakxioSpecialMarket.sol";

contract QuoteStub {
    function decimals() external pure returns (uint8) {
        return 6;
    }
}

contract OptOutSpecialEscape is Test {
    PlakxioBadge badge;
    QuoteStub quote;

    address admin = address(0xA11CE);
    address issuer = address(0x1553E);
    address holder = address(0xB0B);
    address cleanHolder = address(0xC1EA);
    address buyer = address(0xBEEF);

    uint256 specialMask;
    uint256 constant ORDINAL = 893;
    uint256 constant ID = 2_000_000_000 + ORDINAL;
    uint256 constant CONTROL_ID = ID + 1;

    function setUp() public {
        vm.warp(1_000_000);
        quote = new QuoteStub();
        vm.startPrank(admin);
        badge = new PlakxioBadge(admin, "u", uint48(2 days), 2 days);
        badge.grantRole(badge.MINTER_ROLE(), issuer);
        vm.stopPrank();
        specialMask = badge.CLASS_MASK_SPECIAL();
    }

    function _deployMarket(address deployer) internal returns (PlakxioSpecialMarket
m) {
        vm.prank(deployer);
        m = new PlakxioSpecialMarket(
            admin,
            badge,
            IERC20(address(quote)),
            admin,
```

```
            3000,
            1500,
            1000,
            500,
            500_000,
            1_000_000_000,
            uint48(2 days)
        );
    }

    function _marketCanMove(PlakxioSpecialMarket m, address from, uint256 id)
internal returns (bool moved) {
        uint256 before = badge.balanceOf(from, id);
        vm.prank(address(m));
        (bool ok,) = address(badge).call(
            abi.encodeWithSignature(
                "safeTransferFrom(address,address,uint256,uint256,bytes)",
                from,
                buyer,
                id,
                1,
                bytes("")
            )
        );
        moved = ok && badge.balanceOf(from, id) == before - 1 &&
badge.balanceOf(buyer, id) >= 1;
    }

    function test_optOutBeforeWhitelist_badgeBecomesPermanentlyUnsellable() public
{
        address deployer = address(0xDEAD);
        address predicted = vm.computeCreateAddress(deployer,
vm.getNonce(deployer));

        vm.prank(holder);
        badge.setSettlementOptOut(predicted, true);

        PlakxioSpecialMarket market = _deployMarket(deployer);
        assertEq(address(market), predicted, "address prediction holds");

        vm.prank(admin);
        badge.setSettlementContract(address(market), true, specialMask);
        vm.warp(block.timestamp + 3 days);

        vm.prank(issuer);
        badge.mintSpecial(holder, ID, ORDINAL);
        vm.prank(issuer);
```

```
        badge.mintSpecial(cleanHolder, CONTROL_ID, ORDINAL + 1);

        vm.prank(holder);
        vm.expectRevert(PlakxioBadge.CannotOptOutOfSpecial.selector);
        badge.setSettlementOptOut(address(market), true);

        assertFalse(_marketCanMove(market, holder, ID), "HARM: escaped Special is
unmovable by the market");
        assertEq(badge.balanceOf(holder, ID), 1, "HARM: holder still owns it");
        assertTrue(_marketCanMove(market, cleanHolder, CONTROL_ID), "CONTROL: a
normal Special does sell");
    }

    function test_optOutDuringRevocationWindow_badgeBecomesPermanentlyUnsellable()
public {
        PlakxioSpecialMarket market = _deployMarket(admin);
        vm.prank(admin);
        badge.setSettlementContract(address(market), true, specialMask);
        vm.warp(block.timestamp + 3 days);

        vm.prank(issuer);
        badge.mintSpecial(holder, ID, ORDINAL);
        vm.prank(issuer);
        badge.mintSpecial(cleanHolder, CONTROL_ID, ORDINAL + 1);

        vm.prank(admin);
        badge.setSettlementContract(address(market), false, 0);
        assertEq(badge.settlementClassMask(address(market)), 0, "revocation zeroes
the mask");

        vm.prank(holder);
        badge.setSettlementOptOut(address(market), true);

        vm.prank(admin);
        badge.setSettlementContract(address(market), true, specialMask);
        vm.warp(block.timestamp + 3 days);

        assertFalse(_marketCanMove(market, holder, ID), "HARM: escaped Special is
unmovable by the market");
        assertEq(badge.balanceOf(holder, ID), 1, "HARM: holder still owns it");
        assertTrue(_marketCanMove(market, cleanHolder, CONTROL_ID), "CONTROL: a
normal Special does sell");
    }
}
```

```
forge test --match-path "test/poc/OptOutSpecialEscape.t.sol" -vv
```

Observed on the frozen source:

```
Ran 2 tests for test/poc/OptOutSpecialEscape.t.sol:OptOutSpecialEscape
[PASS] test_optOutBeforeWhitelist_badgeBecomesPermanentlyUnsellable() (gas:
2465997)
[PASS] test_optOutDuringRevocationWindow_badgeBecomesPermanentlyUnsellable() (gas:
2476802)
Suite result: ok. 2 passed; 0 failed; 0 skipped
```

Recommendations

Make the holder opt-out token-aware at transfer time so it applies only to non-Special badges. Pass the token id to `_checkHolderPermits` in the single-transfer path and apply the same check to every id in the batch path.

```
-function _checkHolderPermits(address holder, address operator) internal view {
-    if (settlementOptOut[holder][operator]) revert HolderOptedOut(holder,
operator);
+function _checkHolderPermits(address holder, address operator, uint256 id)
internal view {
+    if (settlementOptOut[holder][operator] && tokenClass(id) !=
TokenClass.Special) {
+        revert HolderOptedOut(holder, operator);
+    }
 }
```

After the explicit settlement, activation, class-mask, and holder checks, call OpenZeppelin's internal safe-transfer functions so the inherited id-less approval check does not reapply the stale flag. Keep `isApprovedForAll` conservative for ordinary transfers. Do not exempt an operator merely because its mask contains the Special bit, because `CLASS_MASK_ALL` must still respect ordinary opt-outs.

Remediation

Fixed. `_checkHolderPermits` takes the id and is class-aware at the move, and `isApprovedForAll` is conservative in the safe direction (it may say "not approved" for a Special the operator can move, never the reverse). Verified at the tag, not from the fix description.

Two consequences of this fix are filed separately rather than reopening this one: #37 (a holder now has no way to refuse an operator for Specials at all) and #38 (the refusal still reports success for the class it does not bind).

[M-02] At feeBps == FEE_BPS_CEILING, opposite rounding makes 19 of every 20 prices unsettleable

Fixed

Tracker: issue #18 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

src/PlakxioSettlement.sol#L860-L886

src/PlakxioSettlement.sol#L950-L973

src/PlakxioSettlement.sol#L1017-L1025

Description

`setFeeParams` permits `feeBps == FEE_BPS_CEILING`. At that setting, `calculateFee` rounds the percentage fee upward while `_finalize` rounds the realised ceiling downward. When the product has a remainder, the fee is one quote-token base unit above its ceiling and settlement reverts.

With the 1,500-bps ceiling, a price is accepted only when `price * 1500` is divisible by 10,000, which reduces to `price % 20 == 0`. Consequently, only one of every twenty integer prices can settle. `quoteFee` repeats the refusal. With `flatMinFee = 10,000`, `minSettleablePrice` returns 66,667 even though the first settleable price at this fee setting is 66,680.

Vulnerable Scenario:

1. An administrator sets `feeBps` to the permitted maximum of 1,500.
2. Makers sign otherwise valid orders at a price that is not a multiple of 20 base units.
3. A relayer submits the match.
4. The upward-rounded fee exceeds the downward-rounded ceiling, so settlement reverts before payment or badge transfer.

The live Base Sepolia setting was 500 bps at block 46,678,271, so the condition was latent at that snapshot.

Impact

At a legal fee configuration, 19 of every 20 integer execution prices become untradeable, leaving valid signed orders unfillable until the administrator lowers the fee.

PoC

Save the following as `test/poc/FeeCeilingUnsettleable.t.sol` and run:

```
forge test --match-path 'test/poc/FeeCeilingUnsettleable.t.sol' -vv
```

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {AuthorizationFixture} from "../helpers/AuthorizationFixture.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";
import {PlakxioSettlement} from "../../src/PlakxioSettlement.sol";
import {PlakxioOrderTypes} from "../../src/PlakxioOrderTypes.sol";
import {MockUSDC} from "../mocks/MockUSDC.sol";

contract FeeCeilingUnsettleable is AuthorizationFixture {
    PlakxioBadge internal badge;
    PlakxioSettlement internal settlement;
    MockUSDC internal usdc;

    address internal admin = makeAddr("admin");
    address internal minter = makeAddr("minter");
    address internal relayer = makeAddr("relayer");
    address internal treasury = makeAddr("treasury");

    uint256 internal sellerKey = 0xA11CE;
    uint256 internal buyerKey = 0xB0B;
    address internal seller;
    address internal buyer;

    uint256 internal constant TOKEN_ID = 42;
    uint256 internal constant TOKEN_ID_2 = 43;
    uint256 internal constant FLAT_MIN_CEILING = 10e6;
    uint256 internal constant INITIAL_FEE_BPS = 500;
    uint256 internal constant INITIAL_FLAT_MIN = 1e6;

    function setUp() public {
        seller = vm.addr(sellerKey);
        buyer = vm.addr(buyerKey);

        badge = new PlakxioBadge(admin, "uri", ADMIN_DELAY, SETTLEMENT_DELAY);
        usdc = new MockUSDC();
        settlement = new PlakxioSettlement(
            admin,
            badge,
            IERC20(address(usdc)),
            treasury,
            FLAT_MIN_CEILING,
            INITIAL_FEE_BPS,
            INITIAL_FLAT_MIN,
            ADMIN_DELAY
        );
```

```
        vm.startPrank(admin);
        badge.grantRole(badge.MINTER_ROLE(), minter);
        badge.setSettlementContract(address(settlement), true,
badge.CLASS_MASK_ORDINARY());
        skip(3 days);
        settlement.grantRole(settlement.RELAYER_ROLE(), relayer);
        vm.stopPrank();

        vm.startPrank(minter);
        badge.mint(seller, TOKEN_ID, 1, TEST_SUPPLY_CAP, "");
        badge.mint(seller, TOKEN_ID_2, 1, TEST_SUPPLY_CAP, "");
        vm.stopPrank();

        usdc.mint(buyer, 10_000e6);
    }

    function _order(address maker, PlakxioOrderTypes.Side side, uint256 tokenId,
uint256 price, uint256 salt)
        internal
        view
        returns (PlakxioOrderTypes.Order memory)
    {
        return PlakxioOrderTypes.Order({
            maker: maker,
            side: side,
            tokenId: tokenId,
            price: price,
            quote: address(usdc),
            nonce: 0,
            expiry: block.timestamp + 1 days,
            salt: salt,
            makerIsContract: false,
            minProceeds: 0
        });
    }

    function _sign(uint256 key, PlakxioOrderTypes.Order memory order) internal view
returns (bytes memory) {
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(key,
settlement.orderDigest(order));
        return abi.encodePacked(r, s, v);
    }

    function _tradeHappens(uint256 tokenId, uint256 price, uint256 salt) internal
returns (bool) {
        PlakxioOrderTypes.Order memory ask =
```

```
            _order(seller, PlakxioOrderTypes.Side.Ask, tokenId, price, salt);
        PlakxioOrderTypes.Order memory bid =
            _order(buyer, PlakxioOrderTypes.Side.Bid, tokenId, price, salt + 1);
        bytes memory askSig = _sign(sellerKey, ask);
        bytes memory bidSig = _sign(buyerKey, bid);
        PlakxioSettlement.PaymentAuthorization memory auth =
            _authorize(usdc, address(settlement), buyerKey, price);
        bytes memory data =
            abi.encodeCall(PlakxioSettlement.settle, (ask, askSig, bid, bidSig,
price, auth));
        vm.prank(relayer);
        (bool ok,) = address(settlement).call(data);
        return ok && badge.balanceOf(buyer, tokenId) == 1;
    }

    function test_feeBpsMayEqualTheCeiling() public {
        uint256 ceilingBps = settlement.FEE_BPS_CEILING();
        vm.prank(admin);
        settlement.setFeeParams(ceilingBps, INITIAL_FLAT_MIN);
        assertEq(settlement.feeBps(), ceilingBps);
    }

    function test_atTheCeiling_anOrdinaryTradeCannotSettle() public {
        uint256 ceilingBps = settlement.FEE_BPS_CEILING();
        vm.prank(admin);
        settlement.setFeeParams(ceilingBps, INITIAL_FLAT_MIN);

        uint256 sellerBefore = usdc.balanceOf(seller);
        // 100.000001 USDC: an ordinary price with a non-zero division remainder.
        assertFalse(_tradeHappens(TOKEN_ID, 100_000_001, 100), "trade unexpectedly
happened");
        assertEq(badge.balanceOf(seller, TOKEN_ID), 1, "seller lost badge");
        assertEq(usdc.balanceOf(seller), sellerBefore, "seller was paid");

        assertTrue(_tradeHappens(TOKEN_ID_2, 100_000_000, 200), "control trade
failed");
    }

    function test_atTheCeiling_nineteenOfEveryTwentyPricesAreUnsettleable() public
{
        uint256 ceilingBps = settlement.FEE_BPS_CEILING();
        vm.prank(admin);
        settlement.setFeeParams(ceilingBps, INITIAL_FLAT_MIN);

        uint256 settleable;
        for (uint256 price = 100_000_000; price < 100_000_100; ++price) {
            (, bool ok) = settlement.quoteFee(price);
```

```
            if (ok) ++settleable;
        }
        assertEq(settleable, 5, "unexpected settleable count");
    }

    function test_atTheCeiling_minSettleablePriceIsItselfUnsettleable() public {
        uint256 ceilingBps = settlement.FEE_BPS_CEILING();
        vm.prank(admin);
        settlement.setFeeParams(ceilingBps, INITIAL_FLAT_MIN);

        (, bool ok) = settlement.quoteFee(settlement.minSettleablePrice());
        assertFalse(ok, "minimum price unexpectedly settles");
    }
}
```

Observed at commit `d8fa452dc423d00760d3159f290d82b7cc5e1857`:

```
[PASS] test_atTheCeiling_anOrdinaryTradeCannotSettle() (gas: 358871)
[PASS] test_atTheCeiling_minSettleablePriceIsItselfUnsettleable() (gas: 25176)
[PASS] test_atTheCeiling_nineteenOfEveryTwentyPricesAreUnsettleable() (gas: 194924)
[PASS] test_feeBpsMayEqualTheCeiling() (gas: 23355)
Suite result: ok. 4 passed; 0 failed; 0 skipped
```

Recommendations

Use the same rounding direction for the percentage fee and its realised ceiling in both `_finalize` and `quoteFee`:

```
-uint256 ceiling = (executionPrice * FEE_BPS_CEILING) / BPS_DENOMINATOR;
+uint256 ceiling = Math.ceilDiv(executionPrice * FEE_BPS_CEILING, BPS_DENOMINATOR);
```

This permits the representable integer fee to exceed the exact mathematical 15% value by less than one quote-token base unit. If the ceiling must remain a strict mathematical bound, reject `feeBps == FEE_BPS_CEILING` instead and test the remaining fee and dust-price boundaries.

Remediation

Fixed. `_realisedCeiling` rounds up with `Math.ceilDiv`, so the ceiling and the fee now round the same way, and `minSettleablePrice` is recomputed from the three real constraints rather than the one. The asymmetry the finding described is gone.

[M-03] Upgrade scripts leave DEFAULT_ADMIN on the broadcast key

Fixed

Tracker: issue #21 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/UpgradeSettlement.s.sol#L92-L120

script/UpgradeSpecialMarket.s.sol#L74-L107

Description

Both upgrade scripts derive the new market's initial `DEFAULT_ADMIN_ROLE` holder from `DEPLOYER_PRIVATE_KEY`. They neither read `ADMIN_ADDRESS` nor call `beginDefaultAdminTransfer()` after deployment. `ADMIN_TRANSFER_DELAY` only configures the delay for future transfers; it does not initiate a handover.

Vulnerable Scenario:

1. An upgrade is executed using a broadcast account that is intended to relinquish control to a separate cold administrator.
2. The new market is registered and made operational, but the broadcast account remains its default admin and `pendingDefaultAdmin()` remains empty.
3. If that account is later compromised, the attacker can exercise the new market's administrative functions indefinitely.

Proof of concept:

Save the following as `test/poc/UpgradeAdminRetention.t.sol` and run `forge test --match-path test/poc/UpgradeAdminRetention.t.sol -vv`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test, Vm} from "forge-std/Test.sol";
import {UpgradeSettlement} from "../../script/UpgradeSettlement.s.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";
import {PlakxioSettlement} from "../../src/PlakxioSettlement.sol";
import {MockUSDC} from "../mocks/MockUSDC.sol";

contract UpgradeAdminRetention is Test {
    event SettlementContractSet(address indexed settlement, bool allowed, uint256
classMask);

    function test_upgradeIgnoresTheConfiguredColdAdmin() public {
        uint256 deployerKey = 0xD00D;
        address deployer = vm.addr(deployerKey);
        address coldAdmin = makeAddr("coldAdmin");
        MockUSDC usdc = new MockUSDC();

        vm.prank(deployer);
        PlakxioBadge badge = new PlakxioBadge(deployer, "uri", 2 days, 1 days);

        vm.setEnv("BADGE_ADDRESS", vm.toString(address(badge)));
        vm.setEnv("USDC_ADDRESS", vm.toString(address(usdc)));
        vm.setEnv("TREASURY_ADDRESS", vm.toString(makeAddr("treasury")));
        vm.setEnv("RELAYER_ADDRESS", vm.toString(makeAddr("relayer")));
        vm.setEnv("DEPLOYER_PRIVATE_KEY", vm.toString(deployerKey));
        vm.setEnv("ADMIN_ADDRESS", vm.toString(coldAdmin));

        vm.recordLogs();
        new UpgradeSettlement().run();
        Vm.Log[] memory logs = vm.getRecordedLogs();
        address settlement;
        for (uint256 i = 0; i < logs.length; ++i) {
            if (
                logs[i].emitter == address(badge) && logs[i].topics.length == 2
                    && logs[i].topics[0] == SettlementContractSet.selector
            ) settlement = address(uint160(uint256(logs[i].topics[1])));
        }

        PlakxioSettlement market = PlakxioSettlement(settlement);
        assertEq(market.defaultAdmin(), deployer);
        (address pending,) = market.pendingDefaultAdmin();
        assertEq(pending, address(0));

        address redirectedTreasury = makeAddr("redirectedTreasury");
        vm.prank(deployer);
```

```
        market.setTreasury(redirectedTreasury);
        assertEq(market.treasury(), redirectedTreasury);
    }
}
```

Observed on the frozen source:

```
[PASS] test_upgradeIgnoresTheConfiguredColdAdmin()
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

The same retained-admin state was reproduced through `UpgradeSpecialMarket.run()`.

Impact

A later compromise of the broadcast key can redirect future treasury commissions and change fee, floor, or escalation parameters on the replacement market. The key cannot seize escrowed principal because the markets retain no principal between calls. Recovery requires the legitimate badge administrator to deauthorize and replace the affected market.

Recommendations

Read and validate `ADMIN_ADDRESS` before starting the broadcast, then begin the two-step handover on each new market before `stopBroadcast()`:

```
 address deployer = vm.addr(deployerKey);
+address finalAdmin = vm.envAddress("ADMIN_ADDRESS");
+require(finalAdmin != address(0) && finalAdmin != deployer, "ADMIN_ADDRESS must be
a separate cold admin");
 vm.startBroadcast(deployerKey);
 // deploy and configure the new market
+market.beginDefaultAdminTransfer(finalAdmin);
 vm.stopBroadcast();
```

Apply the same sequence to both upgrade scripts and log that the cold administrator must accept after the configured delay. Update the settlement upgrade comment so it does not claim parity with the base deployment until the handover is actually initiated. Add tests for both upgrade paths, zero and deployer-equal addresses, and delayed acceptance.

This makes upgrades explicitly two-phase and requires a second cold-admin transaction after the delay. Because the change is confined to deployment scripts, it does not increase the deployed market bytecode.

Remediation

Fixed. The upgrade scripts call `_readHandover(deployer, true)`, so an unset `ADMIN_ADDRESS` reverts `AdminAddressUnset` instead of defaulting, and

`beginDefaultAdminTransfer` runs on every target. The shared `AdminHandover` module also refuses a handover in flight to a different admin and an admin that cannot accept.

[M-04] Retiring the old market before its replacement activates halts trading

Fixed

Tracker: issue #22 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/UpgradeSettlement.s.sol#L71-L86

script/UpgradeSpecialMarket.s.sol#L106-L112

src/PlakxioBadge.sol#L364-L402

Description

`setSettlementContract()` makes a newly registered market active only at `block.timestamp + settlementActivationDelay`, while revoking an old market takes effect immediately. However, `UpgradeSettlement.s.sol` instructs the operator to retire the old market immediately after verification, and `UpgradeSpecialMarket.s.sol` prints the same retirement command without checking whether the replacement is active.

If verification finishes before the activation delay, following this procedure leaves the old market revoked and the replacement inactive. Reauthorizing the old market does not restore service immediately because every grant starts a new activation delay. The default deployment delay is two days. The current Base Sepolia badge uses 120 seconds, so this is not a current incident when verification takes longer than two minutes.

The comment that the overlap is “measured in minutes” also does not hold for longer delays: either the old market is retired before activation and trading stops, or it remains authorized until the replacement activates.

Vulnerable Scenario:

Add the following test as `test/poc/UpgradeRetOutage.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test} from "forge-std/Test.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";

contract PocMover {
    function move(PlakxioBadge badge, address from, address to, uint256 id)
external {
        badge.safeTransferFrom(from, to, id, 1, "");
    }
}

contract UpgradeRetOutage is Test {
    address internal holder = address(0xA11CE);
    address internal buyer = address(0xB0B);
    uint256 internal constant ID = 81;
    uint256 internal constant DEFAULT_DELAY = 2 days;

    function test_documented_outage() public {
        PlakxioBadge badge = new PlakxioBadge(address(this), "uri", uint48(2
days), DEFAULT_DELAY);
        badge.grantRole(badge.MINTER_ROLE(), address(this));
        badge.mint(holder, ID, 1, 1, "");

        PocMover oldMarket = new PocMover();
        PocMover newMarket = new PocMover();
        badge.setSettlementContract(address(oldMarket), true,
badge.CLASS_MASK_ORDINARY());
        skip(DEFAULT_DELAY + 1);

        badge.setSettlementContract(address(newMarket), true,
badge.CLASS_MASK_ORDINARY());
        uint256 activeFrom = badge.settlementActiveFrom(address(newMarket));
        skip(10 minutes);
        badge.setSettlementContract(address(oldMarket), false, 0);

        assertFalse(badge.isApprovedForAll(holder, address(oldMarket)));
        assertFalse(badge.isApprovedForAll(holder, address(newMarket)));

vm.expectRevert(abi.encodeWithSelector(PlakxioBadge.NotSettlementContract.selector,
address(oldMarket)));
        oldMarket.move(badge, holder, buyer, ID);
        vm.expectRevert(
            abi.encodeWithSelector(PlakxioBadge.SettlementNotActiveYet.selector,
address(newMarket), activeFrom)
        );
```

```
        newMarket.move(badge, holder, buyer, ID);

        vm.warp(activeFrom);
        newMarket.move(badge, holder, buyer, ID);
    }
}
```

Run `forge test --match-contract UpgradeRetOutage -vv`.

Observed output:

```
[PASS] test_documented_outage()
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

Impact

An operator following the documented upgrade procedure can halt all ordinary-market settlements, or all Special-market buyouts, for the remainder of the activation delay. No funds are lost, and service returns when the replacement activates, but the production default makes the outage last almost two days.

Recommendations

Split upgrades into registration and retirement stages. Print the replacement's `settlementActiveFrom`, keep its off-chain order admission closed, and make the retirement helper refuse to revoke the old market unless the replacement is whitelisted for the expected class and `block.timestamp >= settlementActiveFrom(replacement)`. Then switch order admission and retire the old market as one procedure.

This keeps the old market authorized during the activation delay. Keeping the replacement's order book closed during that period avoids a meaningful cross-market cancellation window; the overlap should be documented as activation-delay-bound rather than “measured in minutes.”

Remediation

Fixed. The upgrade is a two-stage operation with staged and promoted manifests, and the retirement refuses to run while an upgrade is staged but not promoted. The inversion the finding described — retiring the old market while the replacement was unpromoted — is no longer reachable.

[M-05] Upgrade scripts can register markets with an invalid quote token

Fixed

Tracker: issue #26 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/DeployBase.s.sol#L130-L154

script/UpgradeSettlement.s.sol#L92-L118

script/UpgradeSpecialMarket.s.sol#L74-L106

Description

Both upgrade scripts read `USDC_ADDRESS` and pass it directly to a new market without validating it. Unlike `DeployBase`, they do not call `_assertQuoteTokenLive()`. The shared market constructor only rejects the zero address, stores the quote token immutably, and the upgrade scripts immediately register the new market as a trusted badge operator.

Consequently, a stale address from another network can produce a registered market whose quote token has no code. The existing `DeployBase` guard is also incomplete because it checks only code presence and six decimals, not that the address is the canonical token for the active chain or that it supports the required EIP-3009 `receiveWithAuthorization()` flow.

Vulnerable Scenario:

An operator runs `UpgradeSettlement` with a wrong-network `USDC_ADDRESS`. The upgrade succeeds, permanently stores the codeless address as `quoteCurrency`, and registers the unusable market. Save the following test as `test/poc/UpgradeAcceptsCodelessQuoteInline.t.sol` and run `forge test --match-path test/poc/UpgradeAcceptsCodelessQuoteInline.t.sol -vv`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test, Vm} from "forge-std/Test.sol";
import {UpgradeSettlement} from "../../script/UpgradeSettlement.s.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";
import {PlakxioSettlement} from "../../src/PlakxioSettlement.sol";

contract UpgradeAcceptsCodelessQuoteInline is Test {
    event SettlementContractSet(address indexed settlement, bool allowed, uint256
classMask);

    function test_upgradeWiresAQuoteTokenWithNoCode() public {
        uint256 deployerKey = 0xD00D;
        address deployer = vm.addr(deployerKey);
        address codelessQuote =
address(0xDeaD000000000000000000000000000000000BEEf);

        vm.prank(deployer);
        PlakxioBadge badge = new PlakxioBadge(deployer, "uri", 0, 120);
        vm.setEnv("BADGE_ADDRESS", vm.toString(address(badge)));
        vm.setEnv("USDC_ADDRESS", vm.toString(codelessQuote));
        vm.setEnv("TREASURY_ADDRESS", vm.toString(makeAddr("treasury")));
        vm.setEnv("RELAYER_ADDRESS", vm.toString(makeAddr("relayer")));
        vm.setEnv("DEPLOYER_PRIVATE_KEY", vm.toString(deployerKey));

        assertEq(codelessQuote.code.length, 0);
        vm.recordLogs();
        new UpgradeSettlement().run();

        Vm.Log[] memory logs = vm.getRecordedLogs();
        address settlement;
        for (uint256 i; i < logs.length; ++i) {
            if (logs[i].emitter == address(badge) && logs[i].topics[0] ==
SettlementContractSet.selector) {
                settlement = address(uint160(uint256(logs[i].topics[1])));
            }
        }

        assertNotEq(settlement, address(0));
        assertEq(address(PlakxioSettlement(settlement).quoteCurrency()),
codelessQuote);
        assertTrue(badge.isSettlement(settlement));
    }
}
```

Observed output (gas and timing omitted):

```
Ran 1 test for
test/poc/UpgradeAcceptsCodelessQuoteInline.t.sol:UpgradeAcceptsCodelessQuoteInline
[PASS] test_upgradeWiresAQuoteTokenWithNoCode()
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

The test executes the real upgrade script and confirms that it accepts and registers the codeless quote token.

Impact

The newly registered ordinary or Special market cannot complete trades because both payment paths require `receiveWithAuthorization()` and subsequent ERC-20 transfers through the immutable quote-token address. Restoring service requires deploying and registering another market. Replacing the ordinary market also invalidates resting order signatures because they bind to its EIP-712 domain, while replacing the Special market requires its floor and term state to be reconstructed or migrated.

Failed trades revert atomically, so this causes a trading outage and recovery cost rather than partial payment or badge loss.

Recommendations

Create one shared quote-token preflight and call it from `DeployBase`, `UpgradeSettlement`, and `UpgradeSpecialMarket` before broadcasting. For each supported chain, require the configured address to equal Circle's canonical token address, then retain the code-presence and six-decimal checks as diagnostics. The exact-address check is what establishes support for the EIP-3009 flow the markets require.

Add regressions that reject a wrong-chain codeless address and a six-decimal token without EIP-3009 support, together with a positive control showing that the canonical token still completes a representative signed trade.

Remediation

Fixed, and beyond what the finding asked. The preflight checks the canonical address for the chain, code presence, ERC-20 shape, exactly 6 decimals, and probes for EIP-3009. It also refuses a treasury equal to the quote token and a treasury at an address the run is about to deploy to.

[M-06] Retirement can leave historical market authority active

Fixed

Tracker: issue #27 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/retire_superseded.sh#L26-L59

Description

`retire_superseded.sh` applies identities from the current deployment artifact to every historical market. It reads `BADGE` from the current `PlakxioBadge` entry and incorrectly reads `RELAYER` from `.treasury`, then queries and revokes those identities on each old market.

These identities are not stable across generations. A full base deployment creates a new badge, while treasury and relayer are separate configuration values. An old market remains bound to its own immutable `badge()`, and `RELAYER_ROLE` may contain historical or rotated members. Consequently, the script can observe a zero mask on the current badge and no role for the current treasury, skip both revocations, and print `done.` while the old badge and actual old relayer remain authorized.

Vulnerable Scenario:

1. An old generation authorizes `oldMarket` on `oldBadge`, grants `RELAYER_ROLE` to `oldRelayer`, and receives a valid signed ask, bid, and buyer payment authorization.
2. A later base deployment creates a new badge and replaces the current deployment artifact. The current treasury is not `oldRelayer`.
3. The administrator runs `retire_superseded.sh`. The current badge reports a zero mask for `oldMarket`, and `oldMarket` reports that the current treasury lacks `RELAYER_ROLE`, so the script sends neither revocation and reports completion.
4. If `oldRelayer` is compromised, it can submit the still-valid signed trade through `oldMarket`, which continues to use `oldBadge` and accepts the obsolete relayer.

Save the following as `test/poc/RetirementIdentityPoC.t.sol` and run `forge test --match-path test/poc/RetirementIdentityPoC.t.sol --via-ir -vv`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {AuthorizationFixture} from "../helpers/AuthorizationFixture.sol";
import {MockUSDC} from "../mocks/MockUSDC.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";
import {PlakxioOrderTypes} from "../../src/PlakxioOrderTypes.sol";
import {PlakxioSettlement} from "../../src/PlakxioSettlement.sol";

contract RetirementIdentityPoC is AuthorizationFixture {
    function _order(address maker, PlakxioOrderTypes.Side side, address quote,
uint256 salt)
        internal
        view
        returns (PlakxioOrderTypes.Order memory)
    {
        return PlakxioOrderTypes.Order({
            maker: maker,
            side: side,
            makerIsContract: false,
            tokenId: 42,
            price: 100e6,
            minProceeds: 0,
            quote: quote,
            nonce: 0,
            expiry: block.timestamp + 1 days,
            salt: salt
        });
    }

    function _sign(PlakxioSettlement market, uint256 key, PlakxioOrderTypes.Order
memory order)
        internal
        view
        returns (bytes memory)
    {
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(key, market.orderDigest(order));
        return abi.encodePacked(r, s, v);
    }

    function test_retirementUsesTheWrongGenerationIdentities() public {
        address admin = makeAddr("admin");
        address treasury = makeAddr("currentTreasury");
        address oldRelayer = makeAddr("oldRelayer");
        uint256 sellerKey = 0xA11CE;
        uint256 buyerKey = 0xB0B;
```

```
        address seller = vm.addr(sellerKey);
        address buyer = vm.addr(buyerKey);

        MockUSDC usdc = new MockUSDC();
        PlakxioBadge oldBadge = new PlakxioBadge(admin, "old", 2 days, 60);
        PlakxioBadge currentBadge = new PlakxioBadge(admin, "current", 2 days,
60);
        PlakxioSettlement oldMarket = new PlakxioSettlement(
            admin, oldBadge, IERC20(address(usdc)), treasury, 10e6, 500, 1e6, 2
days
        );

        vm.startPrank(admin);
        oldBadge.setSettlementContract(address(oldMarket), true,
oldBadge.CLASS_MASK_ORDINARY());
        oldMarket.grantRole(oldMarket.RELAYER_ROLE(), oldRelayer);
        oldBadge.grantRole(oldBadge.MINTER_ROLE(), admin);
        vm.stopPrank();
        skip(60);
        vm.prank(admin);
        oldBadge.mint(seller, 42, 1, 1, "");
        usdc.mint(buyer, 1_000e6);

        PlakxioOrderTypes.Order memory ask =
            _order(seller, PlakxioOrderTypes.Side.Ask, address(usdc), 1);
        PlakxioOrderTypes.Order memory bid =
            _order(buyer, PlakxioOrderTypes.Side.Bid, address(usdc), 2);
        bytes memory askSig = _sign(oldMarket, sellerKey, ask);
        bytes memory bidSig = _sign(oldMarket, buyerKey, bid);
        PlakxioSettlement.PaymentAuthorization memory auth =
            _authorize(usdc, address(oldMarket), buyerKey, 100e6);

        vm.startPrank(admin);
        if (currentBadge.settlementClassMask(address(oldMarket)) != 0) {
            currentBadge.setSettlementContract(address(oldMarket), false, 0);
        }
        if (oldMarket.hasRole(oldMarket.RELAYER_ROLE(), treasury)) {
            oldMarket.revokeRole(oldMarket.RELAYER_ROLE(), treasury);
        }
        vm.stopPrank();

        assertEq(oldBadge.settlementClassMask(address(oldMarket)),
oldBadge.CLASS_MASK_ORDINARY());
        assertTrue(oldMarket.hasRole(oldMarket.RELAYER_ROLE(), oldRelayer));
        uint256 snapshot = vm.snapshotState();

        vm.prank(oldRelayer);
```

```
        oldMarket.settle(ask, askSig, bid, bidSig, 100e6, auth);
        assertEq(oldBadge.balanceOf(buyer, 42), 1);
        assertEq(usdc.balanceOf(buyer), 900e6);
        assertEq(usdc.balanceOf(seller), 95e6);
        assertEq(usdc.balanceOf(treasury), 5e6);

        assertTrue(vm.revertToState(snapshot));
        vm.startPrank(admin);
        oldBadge.setSettlementContract(address(oldMarket), false, 0);
        oldMarket.revokeRole(oldMarket.RELAYER_ROLE(), oldRelayer);
        vm.stopPrank();
        vm.prank(oldRelayer);
        vm.expectRevert();
        oldMarket.settle(ask, askSig, bid, bidSig, 100e6, auth);
    }

    function test_sameBadgeAndRelayerControlRetiresTheMarket() public {
        address admin = makeAddr("admin");
        address relayerAndTreasury = makeAddr("relayerAndTreasury");
        MockUSDC usdc = new MockUSDC();
        PlakxioBadge badge = new PlakxioBadge(admin, "same", 2 days, 60);
        PlakxioSettlement market = new PlakxioSettlement(
            admin, badge, IERC20(address(usdc)), relayerAndTreasury, 10e6, 500,
1e6, 2 days
        );

        vm.startPrank(admin);
        badge.setSettlementContract(address(market), true,
badge.CLASS_MASK_ORDINARY());
        market.grantRole(market.RELAYER_ROLE(), relayerAndTreasury);
        badge.setSettlementContract(address(market), false, 0);
        market.revokeRole(market.RELAYER_ROLE(), relayerAndTreasury);
        vm.stopPrank();

        assertEq(badge.settlementClassMask(address(market)), 0);
        assertFalse(market.hasRole(market.RELAYER_ROLE(), relayerAndTreasury));
    }
}
```

Observed output (gas and timing omitted):

```
Ran 2 tests for test/poc/RetirementIdentityPoC.t.sol:RetirementIdentityPoC
[PASS] test_retirementUsesTheWrongGenerationIdentities()
[PASS] test_sameBadgeAndRelayerControlRetiresTheMarket()
Suite result: ok. 2 passed; 0 failed; 0 skipped
```

The control shows the same procedure succeeds when the badge and relayer identities match. Separately, source review shows that suppressed broadcast-discovery errors can produce an empty loop followed by `done.` when deployment history is unavailable; this secondary condition was not runtime-tested.

Impact

A compromised obsolete relayer can execute still-valid maker-signed trades after administrators were told that the generation was retired. In the reproduced 100 USDC trade, the buyer paid 100 USDC, the seller received 95 USDC, the treasury received 5 USDC, and the badge moved to the buyer through the obsolete market.

This is not arbitrary theft: valid ask and bid signatures, a live buyer payment authorization, ownership, an in-band price, an unexpired order, and residual historical badge and relayer authority are all required. No currently executable deployment was identified, so this report does not claim an active incident.

Recommendations

For each historical market, obtain its actual badge by calling `badge()` and revoke that market on the returned contract. Reconstruct the complete live `RELAYER_ROLE` membership from `RoleGranted` and `RoleRevoked` events starting at the market's deployment block, or from an authoritative append-only registry, and revoke every remaining member rather than substituting one configured address.

After each transaction confirms, require the historical badge to report `isSettlement(market) == false` and `settlementClassMask(market) == 0`, and verify that every reconstructed relayer lacks the role. Treat missing broadcast history, an empty historical-market set, incomplete event coverage, RPC or transaction failures, and failed postconditions as fatal; print `done.` only after every historical market passes these checks.

Future deployment artifacts should record each generation's market, badge, deployment block, and initial relayer grants. The retirement process must still replay later role events so rotation and multiple concurrent relayers are handled correctly.

Remediation

Fixed. The historical operator set is derived from the broadcast history rather than from the current artifact, a missing or unreadable history is fatal rather than silent, and a run that retires nothing is treated as an error instead of a result. `done.` now prints only when every historical market passed.

[M-07] Deployment artifacts are not bound to the configured RPC chain

Fixed

Tracker: issue #28 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/deploy_base.sh#L51-L290

script/retire_superseded.sh#L22-L55

Description

The operational scripts do not establish that their RPC endpoint and deployment artifacts belong to the same chain. `deploy_base.sh` selects the artifact namespace independently through `NETWORK_KEY`, which defaults to `base-sepolia`. Its final chain check compares the staged `chainId`, generated from `BASE_RPC`, with `cast chain-id` from that same RPC. A healthy RPC for the wrong chain therefore passes, after which the script promotes the wrong-chain addresses into the Base Sepolia artifact namespace.

`retire_superseded.sh` hardcodes the Base Sepolia chain directory and address artifact, but performs its reads and privileged `cast send` operations through an unconstrained `CHAIN_RPC_URL`. The same deployer and nonce sequence can produce identical contract addresses on different chains, so the mismatched script can reach real contracts instead of failing on absent code.

Vulnerable Scenario:

An operator runs either script with a stale but healthy RPC URL. Deployment records chain `31337` under the default Base Sepolia namespace because both sides of its chain check query that RPC. In the retirement test, Base Sepolia history and exclusions applied through a local RPC reporting chain `8453` de-whitelisted the same-address market there and revoked its relayer role, while the intended chain `84532` remained unchanged. The two local chains produced the same contract addresses from the same deployer and nonce sequence.

Two disposable local Anvil harnesses invoked the unmodified deployment and retirement scripts. They produced:

```
deployment_wrong_namespace_chain=31337
ChainBindingDeploy PASS
wrong_rpc_source_mask_role=11/true
wrong_rpc_target_mask_role=0/false
correct_rpc_source_mask_role=0/false
ChainBindingRetire PASS
```

Impact

A stale or copied RPC value can replace the authoritative deployment artifact with addresses from another chain, disrupting client routing and signature-domain resolution until rollback. The retirement path can instead disable a live market on the wrong chain by clearing its settlement mask and revoking its relayer role. Recovery requires privileged reauthorization and may incur the badge activation delay. The reproduced scenarios did not demonstrate theft or direct loss of funds.

Recommendations

Define an explicit mapping from every supported `NETWORK_KEY` to its expected numeric chain ID. Before simulation, broadcast, any `cast send`, or artifact mutation, query the RPC chain and fail unless it matches that expected ID.

Derive the retirement history directory and address artifact from the selected network, then require their recorded chain association to match both the expected chain and the RPC. Treat missing, malformed, or conflicting chain metadata as an error. Add regression tests covering correct and stale RPC endpoints, unset or unsupported network keys, mismatched artifacts and history, and same-address deployments across chains. Adding a network should require an explicit chain mapping rather than inheriting a default namespace.

Remediation

Fixed. `NETWORK_KEY` maps to exactly one chain id in `script/network.sh`, and the RPC is then required to serve that id. The tautological comparison — both sides read from the same RPC — is gone.

Worth saying: you also closed the gap the tautology was hiding, by detecting a local development node via `web3_clientVersion`. A fork of the right chain passes every chain-id check, and that was not in the finding.

[M-08] Concurrent deployments can promote another run's unverified artifact

Fixed

Tracker: issue #29 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/deploy_base.sh#L51-L293

script/DeployBase.s.sol#L243-L295

Description

All deployment runs using the same `NETWORK_KEY` share `deployments/<network>/addresses.staging.json`. The wrapper deletes this file at startup, the Solidity script recreates it during simulation and broadcast, the wrapper reads it throughout verification, and the final `mv` promotes whichever file occupies that path. There is no same-network lock or run identifier binding the verified candidate to the promoted candidate.

Consequently, two authorized jobs running with `REDEPLOY=true` can replace each other's staging artifact. An ordinary overlap is more likely to produce a visible failure, but a narrower ordering silently publishes the wrong candidate: run A completes verification, run B replaces the staging file and is interrupted, then A promotes B's unverified file and reports success.

Vulnerable Scenario:

The deterministic regression harness ran the real wrapper in separate Bash processes. Its `forge` and `cast` shims controlled only the external CLI responses so the relevant filesystem ordering could be reproduced safely. It produced:

```
single_run_promoted= A
interrupted_B_staging= B
race_promoted_by_A= B
different_key_A_promoted= A
harm_assertion= True
✅ Deployed and verified. Artifacts written to deployments/concurrent-race/
(addresses.json + abi/).
```

The single-run and different-network controls both promoted A. Under the same-network interleaving, A exited successfully but its final `addresses.json` was byte-for-byte B's interrupted staging candidate.

Impact

The authoritative artifact consumed by clients, relayers, and indexers can temporarily point to contracts that the successful run did not verify. This can make clients construct signatures for

the wrong EIP-712 verifying contract and disrupt routing or indexing until operators identify and roll back the artifact. No fund loss was demonstrated. Exploitation requires two authorized same-network redeployments and a narrow verify-to-promote ordering, placing this finding at the low edge of Medium severity.

Recommendations

Acquire an atomic exclusive lock scoped to `NETWORK_KEY` before checking, deleting, or creating any deployment artifact. Hold it through broadcast, all verification, ABI export, address promotion, and cleanup. A second same-network run should fail closed before touching staging, while different network keys should remain independent.

Do not automatically remove an existing lock. Record enough ownership information for diagnosis and require an explicit stale-lock recovery procedure after confirming no deployment is active. Cleanup should release a lock owned by the current process on normal exit and `TERM`.

Add regressions confirming that a concurrent same-key run is rejected, different-key runs remain concurrent, the first run promotes its own candidate, and `TERM` releases the lock. This serializes deployments for one network, but it removes the cross-run publication window without delaying deployments to other networks.

Remediation

Fixed. An exclusive per-`NETWORK_KEY` lock, taken by atomic `mkdir` and recording its owner before the artifact is touched. A lock the run did not create is never removed by it, including on the trap path, so a crashed run cannot delete a live run's lock.

[M-09] Base deployment silently retains DEFAULT_ADMIN when ADMIN_ADDRESS is unset

Fixed

Tracker: issue #30 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/DeployBase.s.sol#L220-L237

Description

`DeployBase._handoverAdmin()` reads `ADMIN_ADDRESS` with the deployer as its default and immediately returns when both addresses match. An unset variable therefore skips all three `beginDefaultAdminTransfer()` calls without warning, while the deployment continues successfully.

The deployed Base Sepolia badge and both markets retain the deploying EOA as `defaultAdmin`, report no pending admin, and emitted no `DefaultAdminTransferScheduled` event from deployment through the review block. A focused test of the real deployment script also confirmed that leaving `ADMIN_ADDRESS` unset schedules no transfer on any contract, whereas setting it schedules all three.

Vulnerable Scenario:

The following steps help understand the issue:

1. An operator runs the base deployment without setting `ADMIN_ADDRESS`.
2. The script defaults the intended admin to the broadcast account and silently skips the handover.
3. The deployment completes with the badge and both markets administered by the operational deployment key.
4. Compromise or misuse of that key can change market permissions, grant minting authority, and modify treasury or fee configuration.

Impact

A successful deployment can leave every protocol contract under the broadcast key instead of the intended long-term administrator. This creates a live administrative-key exposure across the badge and both markets. The current evidence establishes the unsafe state, but not how the deployment key is stored, so the issue is rated Medium rather than High.

Recommendations

Require `ADMIN_ADDRESS` with `vm.envAddress()` and reject both the zero address and the deployer unless an explicit, visibly logged emergency override is enabled. Before promoting

deployment artifacts, assert that all three contracts report the expected pending admin.

For existing deployments, begin the transfer on the badge and both markets, accept each transfer from the intended administrator after the delay, and verify `defaultAdmin()` on all three contracts.

Remediation

Fixed. The silent no-op is gone: an unset `ADMIN_ADDRESS` reverts. Keeping the deploy key now requires an explicit `ADMIN_KEEP_DEPLOYER` equal to the deployer, conflicts with a named admin are refused, the run prints the banner, and `adminKeptByDeployer` goes into the artifact. An emergency configuration is now impossible to reach by forgetting a variable.

[M-10] Retirement can deauthorize the replacement while preserving the superseded market

Fixed

Tracker: issue #31 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/retire_superseded.sh#L28-L59

script/UpgradeSettlement.s.sol#L91-L125

Description

`UpgradeSettlement.run()` deploys and authorizes a replacement settlement but only prints its address; it never stages or promotes that address into `deployments/base-sepolia/addresses.json`. The retirement script then treats the two settlements named in that artifact as the live pair and targets every other settlement found in broadcast history.

Following the documented upgrade-then-retire sequence with the stale artifact therefore reverses the intended cutover: the superseded market is excluded, while the replacement is de-whitelisted and loses its relayer role. A deterministic run of the audited retirement script produced:

```
artifact names replacement: sends targeting replacement = 0
artifact names superseded:  sends targeting replacement = 2
```

The two sends in the second case were the badge de-whitelisting transaction and the relayer-role revocation, after which the script printed `done.` and exited successfully.

Vulnerable Scenario:

The following steps help understand the issue:

1. An operator deploys and verifies a replacement with `UpgradeSettlement`.
2. The upgrade script leaves `addresses.json` pointing to the superseded market.
3. The operator runs `retire_superseded.sh` as the next upgrade step.
4. The script preserves the superseded market and removes both required permissions from the replacement.

Impact

The replacement cannot execute settlements, while the authority that retirement was intended to remove remains on the superseded market. Recovery requires restoring both permissions, and re-authorizing the badge permission re-arms its activation delay. Failed settlements revert

atomically, so the demonstrated consequence is an inverted cutover and trading outage rather than partial asset movement.

Recommendations

Make the upgrade produce an authenticated per-upgrade manifest that explicitly records both the superseded and replacement settlements. Validate the manifest, replacement bytecode, badge binding, class mask, and relayer before sending any retirement transaction.

Retire only the recorded superseded address after the replacement has been promoted and verified. Fail closed if the promoted artifact and the upgrade manifest disagree, and verify the final permissions on both contracts before reporting completion.

Remediation

Fixed. Lineage is reconstructed from the signed manifests, a candidate that any manifest names as a replacement and that no later manifest superseded is never retired, and the artifact is held to the latest manifest. The global `FORCE_RETIRE_NEWER=true` override is refused by name and must now name the markets it is for.

[M-11] The Special floor registry enforces no monotonicity, so a whitelisted Special-class operator can write the floor downward

Fixed · residual

Tracker: issue #36 · **Re-review ID:** RR-01 · **Verified at:** `generation-d`

Severity: Medium

Location: `src/PlakxioBadge.sol` — `setSpecialTerms`

Introduced by: the floor-registry fix (email finding, 2026-09-16), verified at `audit-v3-rereview-handoff`

What's wrong

Moving the Special floor and terms onto the badge was the right fix, and `setSpecialTerms` validates a lot: caller is a whitelisted settlement contract, activation delay elapsed, class permitted, the id really is Special, completeness, `uint128` representability, escalation at or above par, and the commission bounded by the escalation above par.

It does not check that the floor is going **up**. The write is unconditional:

```
specialTerms[tokenId] = SpecialTerms(floor, escalated, commissionBps);
```

So any whitelisted, active Special-class operator can write a floor *lower* than the one already stored.

Why it matters

The ascending ratchet is the guarantee the registry exists to provide — it is what makes "no free escape" true and what the original finding was about. That guarantee is now enforced nowhere at the site where the value lives. `PlakxioSpecialMarket` happens to satisfy it by construction (`newFloor = payment * escalated / BPS`), so nothing in the suite can see the gap — the same blind spot described in §5.1 of the re-review handoff: a guard only your own market exercises is a guard no test can see.

The premise is the one you asked us to rate in §10(c)/§5.10, and we rate it the same way you did: a whitelisted settlement contract that is not `PlakxioSpecialMarket` is in the threat model. This is a live instance of that class — and it is reachable by a *correct-looking* future market with an ordinary arithmetic bug, not only by a hostile one.

Recommendation

One comparison, next to the checks already there:

```
uint256 current = specialTerms[tokenId].floor;
if (floor < current) revert FloorBelowStored(tokenId, floor, current);
```

Whether equality should be permitted is your call (a re-write of identical terms is harmless; refusing it is stricter). Then remove the check and confirm a test dies, per your own round-3 practice.

Remediation

Fixed for the stored-floor and holder-proceeds regression. The first write from zero remains reachable and is retained as a documented residual.

`setSpecialTerms` now refuses a record for an unminted Special, refuses a floor below the stored floor (equality permitted, because an honest sale writes an equal floor when profit and commission are both zero), and refuses a record that would lower the proceeds the stored record promises its holder.

We tested the part of the remediation that was hardest to accept on its face — the claim that the one-base-unit rounding allowance cannot be accumulated. It holds: 300 adversarial writes, including the reset-to-zero-and-repeat sequence, did not erode the holder's promise by more than the single specified base unit, and the floor never fell. The allowance is granted only against a stored record whose commission is zero, in which case the stored promise *is* the floor, so a record carrying a commission admits no further erosion. Live simulations across two providers reproduced `FloorBelowStored(2001000739, 574999, 575000)` and the proceeds guard on the deployed badge.

Not closed by this ratchet. A hostile whitelisted Special-class operator can still make the *first* terms write on a minted, never-traded Special at a one-base-unit floor. The market falls back to its configured `initialFloor` only while the stored floor is zero, so once that record exists the badge is priced at the dust floor. Our independent test confirmed this privileged first-write path; exposure is bounded by `initialFloor`, and it requires a hostile or arithmetically incorrect whitelisted operator behind the admin Safe and the activation delay.

On the client's request for a judgement: retain the holder-proceeds rule. It places a small amount of arithmetic on the badge, departing from our earlier guidance to keep arithmetic out of it, and we accept that departure — it is one multiplication and two divisions over values the same function has already validated, and it closes the committed-split problem through the registry's own write path. For a subsequent generation we recommend a lower bound on the first write, or recording the opening price at mint. That change is not implemented in the reviewed code.

[M-12] A holder has no on-chain way to refuse an operator for their Special badges, and the delay that replaces it is floored at 60 seconds

Partly addressed

Tracker: issue #37 · **Re-review ID:** RR-02 · **Verified at:** `generation-d`

Severity: Medium

Location: `src/PlakxioBadge.sol:1063-1066` — `_checkHolderPermits`

Introduced by: the M-01 fix (#17), verified at `audit-v3-rereview-handoff`

You disclosed this yourself in re-review handoff §10(j). Filing it so it has a tracked home rather than living only in correspondence.

What's wrong

The M-01 fix makes the holder's refusal class-aware at the move, which is correct. The way it does that is to exempt Special entirely:

```
function _checkHolderPermits(address holder, address operator, uint256 id)
internal view {
    if (tokenClass(id) == TokenClass.Special) return;
    if (settlementOptOut[holder][operator]) revert HolderOptedOut(holder,
operator);
}
```

A holder therefore has **no on-chain way to refuse an operator for their Special badges**, whatever the operator's mask.

Why it matters

This closes M-01 and we are not asking you to reopen it — a Special is a forced-sale instrument and a holder who could refuse the market would defeat the mechanic. The problem is what is left in its place.

The holder's refusal was the one control the trust model gives a holder against a wrong or compromised operator grant. It is now absent on the highest-value class. Your stated compensating controls are the activation delay, the alerts and the Safe's hardware keys — and **the activation delay's floor is still the 60-second interim placeholder** (`MIN_SETTLEMENT_ACTIVATION_DELAY = 60 seconds`; see our comment on #3). So the compensating control for the class with no holder-side defence is itself a one-minute minimum.

Answering §10(j) directly: the residual is acceptable **only** once the delay floor is a real number. Fix the floor and this becomes a reasonable design trade. Leave the floor at 60 seconds and a

compromised admin can whitelist a hostile Special-class operator and move every Special a minute later, with nothing a holder can sign.

Recommendation

1. Merge the 1-hour floor you already wrote (see #3). That is the load-bearing part.
2. If you want a holder-side defence for Specials specifically, the shape we would suggest is not a refusal — it is a holder-signed *acknowledgement window*: a Special-class operator newly whitelisted cannot move badge `id` until `settlementActiveFrom` **and** some longer per-class delay has passed, so the holder's protection is time rather than veto. That keeps the forced sale intact and gives the holder the thing a refusal was giving them: notice.

Related: the refusal also reports success it does not deliver — filed separately.

Remediation

Partly addressed. The delay condition is implemented and verified; the Special-holder exemption is retained by design and has not been eliminated.

The condition we attached to this finding is met. The badge enforces a six-hour minimum activation delay at construction, and production deployment of 72 hours is enforced separately (see L-11). That implementation is verified in the reviewed code and on the deployed testnet instance.

The second recommendation — a longer, Special-only acknowledgement window — was not built. We consider the stated rationale supportable: a delay protects only a party who can act inside it, and a Special holder has no action available, because there is no refusal by design, no wallet-to-wallet move inside the walled garden, and no burn. The only party who can act on a wrong or hostile Special-class grant is the administrator, by revoking, and revocation is never delayed. A longer timer would give the administrator more time to do what the first delay already times, and give the holder notice of something they cannot change. What replaces it is legibility rather than veto: the per-badge authority view described under L-09.

The exposure is retained, not removed. Under the documented compromised-Safe assumption a hostile Special-class operator can be whitelisted and, after the delay, move every Special, with nothing a holder can sign. The compensating controls are the delay, the two-step delayed administrator transfer, off-chain alerting and incident procedure. On the alerting: a claimed external-monitor worst-case gap of three to six hours reaches the six-hour testnet floor's boundary rather than establishing a margin inside it, and we state that as the client did rather than claiming more. Off-chain monitoring and key custody were not independently verified by this review.

7.2 Low Severity Findings

[L-01] Gasless makers cannot invalidate unenumerated orders

Fixed · residual

Tracker: issue #19 · **Verified at:** `generation-d`

Affected files

src/PlakxioSettlement.sol#L575

Description

`incrementNonce()` is the only cancellation path that invalidates every order in a maker's nonce epoch, including orders the maker cannot enumerate. However, it reads and updates `nonces[msg.sender]`, while settlement checks `nonces[order.maker]`.

The documented production account model uses gasless Particle MPC EOAs that do not submit transactions. `cancelOrder()` and `cancelOrders()` support this model through maker signatures that anyone can relay, but `incrementNonce()` has no equivalent signature-authorized path. A relayer calling it advances only the relayer's nonce and leaves the maker's orders valid.

A maker can therefore cancel every known order, but cannot invalidate unknown orders after losing client state or recovering from a signing compromise. Those orders may still settle until they expire, for up to 365 days.

Recommendations

Add a permissionless, signature-authorized cancel-all function. Bind the signed message to the maker, the maker's current nonce, and a deadline; support both EOA and ERC-1271 makers; and advance the maker's nonce using the existing unpredictable-jump logic.

Retain `incrementNonce()` for makers that can submit transactions. This adds one selector and typed-data flow without changing existing order or cancellation signatures. Test invalid signatures, expired authorizations, replay across nonce epochs, and ERC-1271 makers.

Remediation

Fixed; a bounded residual is retained and documented.

The gasless cancel-all path stands as verified: `incrementNonceFor` binds the maker's current nonce, checks it before signature recovery, caps the authorization's life by `MAX_AUTHORIZATION_LIFETIME`, and supports an ERC-1271 maker. The jump is hashed over `maker` rather than `msg.sender`, so a deliverer cannot choose it by choosing its own address.

One property is narrower than the original fix note claimed, and the generation-d change for it is commentary rather than a new entropy source. On Base's two-second cadence every input to the jump is known once block N−1 is sealed, so the post-jump nonce can be computed roughly one block *before* delivery rather than only inside it. The contract comment now states that bound exactly.

We retain this as a Low residual rather than treating it as closed. It is maker self-exposure: a fresh order at the post-jump nonce still requires the maker's own signature, and a deliverer who front-runs the cancel-all gains nothing, because its effect does not depend on who delivers it. Cancel-all therefore remains a reliable statement that every order signed *before* it is dead, and is not a statement that the maker will not sign again immediately afterwards — a risk system that needs the latter must key on the maker's next signature. The same construction, and the same residual, applies to the holder opt-out nonce. The existing *Fixed* label refers to the implemented original fix; the predictability has not been eliminated.

[L-02] A narrowed class mask is misreported as a buyer rejection

Fixed

Tracker: issue #20 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

src/PlakxioSettlement.sol#L936-L945

src/PlakxioBadge.sol#L530-L551

src/PlakxioBadge.sol#L620-L645

Description

`PlakxioSettlement._finalize()` uses `badge.isApprovedForAll()` to check whether the settlement may transfer the ask token. However, `PlakxioBadge.isApprovedForAll()` returns `true` for any active settlement with a nonzero class mask without checking whether that mask includes the token's class.

If an administrator narrows the settlement's mask to another nonzero class, the precheck continues to pass for excluded tokens after the activation delay. The subsequent class-aware `safeTransferFrom()` call reverts with `ClassNotPermittedForOperator`, but the bare catch converts this operational failure into `BuyerRejectedBadge`.

The transaction remains atomic, so state changes, payments, authorization consumption, and order marks are reverted. The demonstrated impact is limited to exposing the wrong failure classification: an operator-authorization failure that should remain neutral is instead attributed to the buyer. This requires a legitimate mask change, completion of its activation delay, an affected resting order, and a settlement attempt.

Recommendations

Before the transfer catch, compare `settlementClassMask(address(this))` with the bit corresponding to `badge.tokenClass(ask.tokenId)`. Revert with `SettlementNotAuthorized` when the token's class is excluded.

Retain the bare catch for genuine receiver-hook failures, as it prevents a buyer contract from forging trusted error data. Add tests for an excluded class after mask activation, a zero mask, pending activation, seller opt-out, genuine receiver rejection, and a successful transfer. This check adds a class lookup and mask read to the settlement path.

Remediation

Fixed. An operator-authority fault is no longer attributed to the buyer, and the zero-mask case is pinned by test. The fix rode with #17 in `a2b3e9b`, as your comment said.

[L-03] The gas model documents transaction paths the contracts do not have

Fixed

Tracker: issue #23 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

docs/gas_model.md#L3-L21

Description

`docs/gas_model.md` is presented as the current Base Sepolia gas guide, but its transaction model does not match the implementation. It describes ERC-4337 `UserOperation` submission through an EntryPoint and paymaster, names the deleted `buySpecial` entry point, and instructs users to approve USDC. The current architecture uses MPC EOAs and signed relaying: special-market purchases call `buySpecialFor`, both payment paths use EIP-3009 `receiveWithAuthorization`, and neither path reads an allowance.

An integrator following this guide can build a nonexistent UserOperation/paymaster flow, call a selector the market no longer exposes, and request an approval the payment path never consumes. The resulting Special purchase flow is incompatible with the current contracts.

The mismatch can be reproduced from the repository root:

```
set -e
rg -q 'ERC-4337 UserOperation' docs/gas_model.md
! rg -q -i 'entrypoint|useroperation|validateuserop' src
rg -q 'function buySpecialFor\(' src/PlakxioSpecialMarket.sol
! rg -q 'function buySpecial\(' src/PlakxioSpecialMarket.sol
rg -q 'receiveWithAuthorization\(' src/PlakxioSettlement.sol
rg -q 'receiveWithAuthorization\(' src/PlakxioSpecialMarket.sol
! rg -q '\.approve\(|allowance\(' src/PlakxioSettlement.sol
src/PlakxioSpecialMarket.sol
printf '%s\n' 'PASS gas_model.md claims ERC-4337' 'PASS src/ has no ERC-4337
integration symbols' 'PASS buySpecial is absent; buySpecialFor is present' 'PASS
both payment paths use receiveWithAuthorization' 'PASS neither payment path reads
an allowance'
```

Observed output:

```
PASS gas_model.md claims ERC-4337
PASS src/ has no ERC-4337 integration symbols
PASS buySpecial is absent; buySpecialFor is present
PASS both payment paths use receiveWithAuthorization
PASS neither payment path reads an allowance
```

Recommendations

Update the current transaction rows and contract-side requirements to describe MPC EOA signed relaying, `buySpecialFor`, and EIP-3009 `receiveWithAuthorization`. Remove the obsolete ERC-4337, EntryPoint, paymaster, USDC approval, and `buySpecial` instructions. If smart-account support remains planned, label it clearly as future behavior rather than part of the current Base Sepolia flow.

This is a documentation-only change with no bytecode or deployment-size cost.

Remediation

Fixed. The document now describes the transaction paths that exist. The ERC-4337 flow, the `EntryPoint`, the paymaster, the `buySpecial` entry point and the USDC `approve` step are all gone, replaced by a signer/submitter split that matches the contracts, including the EIP-3009 pull on both payment paths.

[L-04] purchaseBlocked can miss a blocked treasury when a badge uses stored commission

Fixed

Tracker: issue #24 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

src/PlakxioSpecialMarket.sol#L355-L429

Description

For a previously traded Special badge, `_executePurchase` calculates the treasury payout from the commission stored when the current holder bought the badge. `purchaseBlocked`, however, checks the treasury only when the current global `commissionBps` is nonzero, and its signature has no `tokenId` from which to read the stored terms.

If the global commission is reduced to zero after a badge has stored a nonzero commission, a blacklisted treasury is therefore omitted from the preflight result. The view returns `(false, address(0))`, but the purchase still calculates a nonzero treasury payout from the stored terms and reverts when transferring it.

Add the following test as `test/poc/PurchaseBlockedLiveVsStored.t.sol`:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {AuthorizationFixture} from "../helpers/AuthorizationFixture.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";
import {PlakxioSpecialMarket} from "../../src/PlakxioSpecialMarket.sol";
import {MockBlacklistUSDC} from "../mocks/MockBlacklistUSDC.sol";

contract PurchaseBlockedLiveVsStoredTest is AuthorizationFixture {
    PlakxioBadge internal badge;
    PlakxioSpecialMarket internal market;
    MockBlacklistUSDC internal usdc;

    address internal admin = makeAddr("admin");
    address internal minter = makeAddr("minter");
    address internal treasury = makeAddr("treasury");

    uint256 internal constant CLAIMER_KEY = 0xC444;
    uint256 internal constant BUYER_1_KEY = 0xB111;
    uint256 internal constant BUYER_2_KEY = 0xB222;
    address internal claimer = vm.addr(CLAIMER_KEY);
    address internal buyer1 = vm.addr(BUYER_1_KEY);
    address internal buyer2 = vm.addr(BUYER_2_KEY);

    uint256 internal constant TRADED_ID = 2_000_000_007;
    uint256 internal constant FRESH_ID = 2_000_000_008;
    uint256 internal constant INITIAL_FLOOR = 1e6;
    uint256 internal constant PROFIT_BPS = 1_000;
    uint256 internal constant COMMISSION_BPS = 500;

    function setUp() public {
        badge = new PlakxioBadge(admin, "uri", ADMIN_DELAY, SETTLEMENT_DELAY);
        usdc = new MockBlacklistUSDC();
        market = new PlakxioSpecialMarket(
            admin,
            badge,
            IERC20(address(usdc)),
            treasury,
            3_000,
            1_500,
            PROFIT_BPS,
            COMMISSION_BPS,
            INITIAL_FLOOR,
            1_000_000_000,
            ADMIN_DELAY
        );
```

```
        vm.startPrank(admin);
        badge.grantRole(badge.MINTER_ROLE(), minter);
        badge.setSettlementContract(address(market), true,
badge.CLASS_MASK_SPECIAL());
        skip(3 days);
        vm.stopPrank();

        vm.startPrank(minter);
        badge.mintSpecial(claimer, TRADED_ID, 7);
        badge.mintSpecial(claimer, FRESH_ID, 8);
        vm.stopPrank();

        usdc.mint(buyer1, 1_000e6);
        usdc.mint(buyer2, 1_000e6);

        // Persist a 5% commission in TRADED_ID's terms.
        _buySpecialAs(market, usdc, BUYER_1_KEY, TRADED_ID, claimer,
INITIAL_FLOOR);
    }

    function test_baselineViewBlocksWhileLiveCommissionIsNonzero() public {
        usdc.setBlacklisted(treasury, true);

        (bool blocked, address blockedBy) = market.purchaseBlocked(buyer2,
buyer1);
        assertTrue(blocked);
        assertEq(blockedBy, treasury);
    }

    function test_viewMissesStoredCommissionAndPurchaseRevertsAtomically() public {
        vm.prank(admin);
        market.setEscalationParams(PROFIT_BPS, 0);
        usdc.setBlacklisted(treasury, true);

        (bool blocked, address blockedBy) = market.purchaseBlocked(buyer2,
buyer1);
        assertFalse(blocked);
        assertEq(blockedBy, address(0));

        uint256 payment = market.currentFloor(TRADED_ID);
        (
            PlakxioSpecialMarket.SpecialPurchase memory p,
            bytes memory sig,
            PlakxioSpecialMarket.PaymentAuthorization memory a
        ) = _prepSpecialBuy(market, usdc, BUYER_2_KEY, TRADED_ID, buyer1, payment);
```

```
        uint256 buyerBefore = usdc.balanceOf(buyer2);
        uint256 holderBefore = usdc.balanceOf(buyer1);
        vm.expectRevert();
        market.buySpecialFor(p, sig, payment, a);

        assertEq(usdc.balanceOf(buyer2), buyerBefore);
        assertEq(usdc.balanceOf(buyer1), holderBefore);
        assertEq(badge.balanceOf(buyer1, TRADED_ID), 1);
        assertFalse(usdc.authorizationState(buyer2, a.nonce));

        // Restoring the treasury path makes the same signed purchase succeed.
        usdc.setBlacklisted(treasury, false);
        market.buySpecialFor(p, sig, payment, a);
        assertEq(badge.balanceOf(buyer2, TRADED_ID), 1);
    }

    function test_freshBadgeAtLiveZeroDoesNotPayTreasury() public {
        vm.prank(admin);
        market.setEscalationParams(PROFIT_BPS, 0);
        usdc.setBlacklisted(treasury, true);

        (bool blocked,) = market.purchaseBlocked(buyer2, claimer);
        assertFalse(blocked);

        uint256 treasuryBefore = usdc.balanceOf(treasury);
        _buySpecialAs(market, usdc, BUYER_2_KEY, FRESH_ID, claimer,
market.currentFloor(FRESH_ID));
        assertEq(badge.balanceOf(buyer2, FRESH_ID), 1);
        assertEq(usdc.balanceOf(treasury), treasuryBefore);
    }
}
```

Run:

```
forge test --match-path test/poc/PurchaseBlockedLiveVsStored.t.sol -vv
```

Observed output:

```
Ran 3 tests for
test/poc/PurchaseBlockedLiveVsStored.t.sol:PurchaseBlockedLiveVsStoredTest
[PASS] test_baselineViewBlocksWhileLiveCommissionIsNonzero() (gas: 53087)
[PASS] test_freshBadgeAtLiveZeroDoesNotPayTreasury() (gas: 239703)
[PASS] test_viewMissesStoredCommissionAndPurchaseRevertsAtomically() (gas: 300652)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 14.43ms (4.11ms CPU
time)

Ran 1 test suite in 23.21ms (14.43ms CPU time): 3 tests passed, 0 failed, 0 skipped
(3 total tests)
```

The failed purchase is atomic: buyer and holder balances, badge ownership, and the payment authorization remain unchanged. The same signed purchase succeeds after the treasury is unblacklisted. The practical impact is a misleading UI or preflight result and wasted transaction-submitter gas, not loss of user funds. The preconditions were not active in the current deployment snapshot, but remain reachable through a normal commission update and a change to the quote token's blacklist.

Recommendations

Add `tokenId` to `purchaseBlocked` and derive the effective commission through one helper shared with `_executePurchase`: use stored terms for a previously traded badge and the live commission for a never-traded badge. Migrate callers to the token-aware signature and remove the two-address version. If ABI compatibility requires a transition, make the legacy view conservatively check the treasury unconditionally and explicitly deprecate it so it cannot return this false negative.

This requires an ABI and client migration. A conservative legacy view also temporarily reports fresh, zero-commission badges as blocked when the treasury is blacklisted, so token-aware callers should be migrated promptly. Add regression tests for stored nonzero commission with live zero, a never-traded badge at live zero, and the ordinary live-nonzero case.

Remediation

Fixed. `_termsFor` reads the badge's stored terms, so the view no longer answers from live configuration, and `purchase` itself reverts `TreasuryCannotReceive` rather than leaving the caller to the view.

The fix changed the function's signature with no overload at the old selector. That fails safe but breaks off-chain callers silently, and the handoff's per-finding table does not note it — filed as #39.

[L-05] Gasless holders cannot submit the operator opt-out required by the trust model

Fixed

Tracker: issue #25 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

src/PlakxioBadge.sol#L540-L584

Description

All holder actions are expected to be signature-delivered because holders are gasless EOAs. However, `setSettlementOptOut` records the refusal against `_msgSender()`, and the badge contract provides no signature-authorized alternative. A relayer therefore opts itself out instead of the holder, leaving the holder's badge transferable by an active ordinary settlement operator.

An admin must first whitelist and activate the operator. The intended settlement contract also moves badges only against signed orders, and order cancellation remains available. The issue specifically prevents the holder-refusal control from bounding a wrong or compromised ordinary-operator grant.

The following Foundry test uses a malicious ordinary operator to show that a direct holder call blocks the transfer, while the same call submitted by a relayer changes only the relayer's state and allows the operator to take the badge:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test} from "forge-std/Test.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";

contract MaliciousOrdinaryOperator {
    function take(PlakxioBadge badge, address from, address to, uint256 id)
external {
        badge.safeTransferFrom(from, to, id, 1, "");
    }
}

contract GaslessHolderOptOutTest is Test {
    PlakxioBadge internal badge;
    MaliciousOrdinaryOperator internal operator;
    address internal admin = makeAddr("admin");
    address internal minter = makeAddr("minter");
    address internal holder = makeAddr("holder");
    address internal recipient = makeAddr("recipient");
    address internal relayer = makeAddr("relayer");

    function setUp() public {
        badge = new PlakxioBadge(admin, "uri", uint48(2 days), 2 days);
        operator = new MaliciousOrdinaryOperator();

        vm.startPrank(admin);
        badge.grantRole(badge.MINTER_ROLE(), minter);
        badge.setSettlementContract(address(operator), true,
badge.CLASS_MASK_ORDINARY());
        vm.stopPrank();

        vm.prank(minter);
        badge.mint(holder, 7, 1, 1, "");
        skip(2 days);
    }

    function test_directHolderOptOutStopsTheOperator() public {
        vm.prank(holder);
        badge.setSettlementOptOut(address(operator), true);

        vm.expectRevert(
            abi.encodeWithSelector(
                PlakxioBadge.HolderOptedOut.selector, holder, address(operator)
            )
        );
        operator.take(badge, holder, recipient, 7);
```

```
    }

    function test_relayerCannotRecordTheHoldersRefusal() public {
        vm.prank(relayer);
        badge.setSettlementOptOut(address(operator), true);

        assertTrue(badge.settlementOptOut(relayer, address(operator)));
        assertFalse(badge.settlementOptOut(holder, address(operator)));

        operator.take(badge, holder, recipient, 7);
        assertEq(badge.balanceOf(recipient, 7), 1);
    }
}
```

Run it from the repository root:

```
forge test --match-path test/poc/GaslessHolderOptOut.t.sol -vvv
```

Observed output:

```
Ran 2 tests for test/poc/GaslessHolderOptOut.t.sol:GaslessHolderOptOutTest
[PASS] test_directHolderOptOutStopsTheOperator() (gas: 52708)
[PASS] test_relayerCannotRecordTheHoldersRefusal() (gas: 90200)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 28.60ms (2.13ms CPU
time)

Ran 1 test suite in 50.26ms (28.60ms CPU time): 2 tests passed, 0 failed, 0 skipped
(2 total tests)
```

Recommendations

Add a relayed EIP-712 opt-out function whose signed payload binds the holder, operator, `optedOut` value, per-holder nonce, and deadline. Verify the holder's signature before updating the holder's state.

Have the direct and relayed setters use the same internal update function so both enforce the Special-class restriction. A direct update must also advance the holder's nonce to invalidate outstanding signatures, preventing an older signed `optedOut = false` message from re-enabling an operator. Add tests for a successful relayed opt-out, expired and replayed signatures, direct-update invalidation, and the Special-class guard.

Remediation

Fixed. `setSettlementOptOutFor` gives the gasless holder the path: EIP-712, a per-(holder, operator) nonce advanced by an unpredictable jump, a deadline capped by `MAX_AUTHORIZATION_LIFETIME`, ERC-1271, and the scheme bound as signed so a later-

code-bearing EOA cannot change interpretation. A stale `optedOut = false` cannot re-enable a refused operator.

What the refusal does *not* reach is Special, by #17's design — see #37 and #38.

[L-06] Setting a market as its own treasury permanently traps revenue

Fixed

Tracker: issue #32 · **Verified at:** `generation-d`

Affected files

src/QuoteTreasuryManaged.sol#L62-L79

src/PlakxioSettlement.sol#L1130-L1160

src/PlakxioSpecialMarket.sol#L371-L385

Description

The shared treasury setter rejects only the zero address, so either market can be configured as its own treasury through its constructor or admin setter.

Both payment paths first pull quote tokens into the market and then transfer the fee or commission to `treasury`. With `treasury == address(this)`, a standard ERC-20 self-transfer succeeds without changing the balance. The trade completes, but protocol revenue remains in a non-upgradeable contract with no withdrawal or rescue path. Changing the treasury later affects only future payments.

A controlled test left both market balances at zero with an external treasury. After changing only the treasury destination, Settlement retained `10,000,000` quote-token units and SpecialMarket retained `4,347,826`; redirecting the treasury did not recover either balance. Only an administrator can create this condition, and user principal is paid normally, so the issue is Low.

Recommendations

Reject `address(this)` in the shared `_setTreasury()` function so the guard covers both constructors and both external setters:

```
if (newTreasury == address(0) || newTreasury == address(this)) revert
InvalidTreasury();
```

Do not add a privileged rescue path solely for this case, because that would unnecessarily expand the administrator's ability to withdraw assets from the trading contracts.

Remediation

Fixed. The earlier "partially fixed" assessment is superseded for the reviewed implementation.

The guard is now symmetric. The badge's grant side inspects the enumerable live-operator set and refuses a candidate that any live operator already names as its treasury

(`SettlementIsATreasury`), and refuses a candidate whose own treasury is itself or a live operator (`TreasuryIsASettlement`). The markets' own setter continues to refuse the badge and live operators (`InvalidTreasury`), so the property holds in both orderings. The pairing probe is a one-word `staticcall` under a bounded gas cap, and a grant that cannot fund the probe is refused by name (`TreasuryProbeUnfunded`) rather than skipping the check — we checked the arithmetic: with twice the probe budget in the frame the probe always receives its full allowance after EIP-150, so the fail-closed property holds.

Read-only simulations from the Safe against the deployed badge, agreeing across two providers, reproduced `SettlementIsATreasury` for treasury-then-whitelist and `InvalidTreasury` for the reverse ordering, with a permitted control confirming the gate discriminates rather than refusing every grant.

Retained limits, which the contract states in its own documentation: a future operator that cannot answer `treasury()` within the probe budget, or whose own treasury setter does not consult the badge, is not covered by a universal pairing guarantee; an operator can temporarily obstruct the grant of the one address it names as its treasury until it is revoked; and a retired market, a token implementation behind a proxy, or any other exit-less recipient still requires operator verification. No rescue path was added, as recommended.

[L-07] Retirement reports success before permission changes are durable on Base

Fixed

Tracker: issue #33 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

script/retire_superseded.sh#L45-L59

Description

Both retirement transactions use `--confirmations 1`, discard the transaction output, and are followed by an unconditional `done.` message. The script preserves no transaction hash, does not wait for a `safe` or `finalized` block, and does not re-read the permissions after the selected safety level is reached.

Base exposes distinct `latest`, `safe`, and `finalized` block tags, while OP Stack finality distinguishes unsafe, safe, and finalized blocks. One sequencer confirmation is therefore not proof that a privileged retirement is durable. If its block is reorganized, the superseded market's permissions return without a new grant transaction, but the operator has no saved hashes with which to reconcile the run. See the Base RPC block-tag documentation and OP Stack finality documentation.

The source-level behavior is:

```
50: ... --confirmations 1 >/dev/null
55: ... --confirmations 1 >/dev/null
59: echo "done."
```

The script is idempotent and re-reads both permissions on a later run, so the state is recoverable by rerunning it. The issue is that no step requires that rerun after finality, while the first run reports completion.

Recommendations

At minimum, document and enforce a second idempotent run after the relevant Base head is finalized. Capture each transaction hash so the operator can reconcile the first run.

For stronger automation, wait until each successful receipt is at or behind the chosen `safe` or `finalized` head, confirm its block remains canonical, then re-read every target's class mask and relayer role at that head. Exit nonzero unless all targets are fully retired.

Remediation

Fixed. `done.` now requires a finality phase after the send tail, defaulting to `RETIRE_FINALITY=safe`, and the run record carries the finality head. The script no longer reports success while its permission changes are still revertible.

[L-08] Readiness views omit a blacklisted market from mandatory payment parties

Fixed

Tracker: issue #34 · **Verified at:** `audit-v3-rereview-handoff`

Affected files

src/PlakxioSettlement.sol#L988-L1015

src/PlakxioSettlement.sol#L1130-L1160

src/PlakxioSpecialMarket.sol#L371-L385

src/PlakxioSpecialMarket.sol#L404-L429

Description

`settlementBlocked()` checks the buyer, seller, and fee-bearing treasury. `purchaseBlocked()` checks the buyer, holder, and commission-bearing treasury. Neither checks whether the quote token restricts the market contract itself.

The market is a mandatory payment party: `receiveWithAuthorization()` sends quote tokens to `address(this)`, and the subsequent payouts originate from that address. If the quote token blacklists the market, every positive trade reverts even though the corresponding readiness view returns `(false, address(0))`.

Controlled tests first completed both trade types with clean accounts. After blacklisting only the relevant market, both readiness views still reported unblocked and the same signed trades reverted with `Blacklisted(market)`. All token and badge state rolled back atomically. The consequence is false availability, wasted transaction gas, and slower diagnosis; creating the condition requires the external quote token's privileged blacklister.

Recommendations

After the existing party checks, make both readiness views test the market address:

```
if (quoteBlacklisted(address(this))) return (true, address(this));
```

Appending this check preserves the current `blockedBy` precedence while making the view account for every mandatory transfer participant.

Remediation

Fixed. The readiness views report the market's own blacklisting — `quoteBlacklisted(address(this))` is checked in both — so a blacklisted market is no longer omitted from the mandatory payment parties.

[L-09] The holder's refusal reports success for Special badges it does not bind

Fixed

Tracker: issue #38 · **Re-review ID:** RR-03 · **Verified at:** `generation-d`

Severity: Low

Location: `src/PlakxioBadge.sol` — `setSettlementOptOut` / `setSettlementOptOutFor` / `isApprovedForAll`

Introduced by: the M-01 fix (#17) together with the relayed opt-out (#25), verified at `audit-v3-rereview-handoff`

What's wrong

A holder refusing an operator that holds `CLASS_MASK_ALL` gets every signal of success:

- the transaction succeeds — `_setSettlementOptOut` only refuses a mask of **exactly** `CLASS_MASK_SPECIAL`, so an all-class operator is a permitted target
- `SettlementOptOutSet` is emitted
- `isApprovedForAll(holder, operator)` returns `false`

and the operator keeps moving that holder's Special badges, because `_checkHolderPermits` exempts the class.

The refusal does bind the operator's ordinary moves, which is the intended behaviour and is why the mask check is `==` and not `&` — that part is right and we verified it. The issue is purely that nothing tells the holder the refusal is partial.

Why it matters

This is the state a holder is most likely to be in: operators are granted `CLASS_MASK_ALL` in practice, so "I refused them" and "they can still take my Special" are simultaneously true, and every observable says the refusal took. An integrator reading `isApprovedForAll` is told the operator is not approved while a Special transfer from that operator will succeed — the view and the gate disagree in the direction that misleads.

No funds move that would not otherwise move; the harm is that a holder and an integrator both act on a protection they do not have.

Recommendation

Make the partiality legible. Any of these is enough:

- emit the scope: `SettlementOptOutSet(holder, operator, optedOut, effectiveMask)` where the mask excludes `CLASS_MASK_SPECIAL`

- add a view that answers per id — `operatorMayMove(holder, operator, id)` — and point integrators at it instead of `isApprovedForAll`
- at minimum, state it in the natspec of `setSettlementOptOut` in the words a holder would read, and in the client surface that offers the refusal

Related: #37 (the underlying exemption) and #3 (the activation-delay floor that stands in for the missing control).

Remediation
Fixed.

The remediation takes the two options this finding offered together: a per-badge view and the scope stated in the contract's own documentation. `operatorMayMove(holder, operator, id)` answers whether a given operator may move a given holder's badge now, and it is built by an external self-call into `requireOperatorMayMove`, so the transfer gate's own reverts become the answer rather than a separately maintained restatement of the policy that could drift from it. The gate itself is unchanged. `isApprovedForAll` now directs integrators to the per-badge view, and the refusal setter's documentation states, in a holder's terms, that the refusal does not reach Special badges and that the transaction, the event and `isApprovedForAll` will not say so.

We independently re-verified the failure mode the client's own adversarial round found in the first cut of this view: a gas scan across every value from 2,000 to 60,000 produced no misleading `false` answer — every read returned either the authority answer or a named refusal. A gas-starved read can therefore no longer report "cannot move" for a badge the operator can move, which would have reproduced this finding's harm inside the view that fixes it.

Retained and documented: an empty batch falls outside a single-token view's domain, and the settlement and market still restate the gate in their own readiness views rather than asking this one — consistent in the reviewed code, but a drift risk later. Client-facing refusal copy and its rendered behaviour were not verified from the contract repository; the client reports that copy as awaiting its own approval, and this result does not certify frontend adoption.

[L-10] purchaseBlocked changed signature with no overload, breaking off-chain callers silently

Addressed · docs

Tracker: issue #39 · **Re-review ID:** RR-04 · **Verified at:** `generation-d`

Severity: Low / Informational

Location: `src/PlakxioSpecialMarket.sol` — `purchaseBlocked`

Introduced by: the L-04 fix (#24), verified at `audit-v3-rereview-handoff`

What's wrong

The L-04 fix is correct — reading the stored terms is exactly what we asked for. It changed the function's signature to do it:

```
-function purchaseBlocked(address buyer, address holder) external view returns (bool blocked, address blockedBy)
+function purchaseBlocked(address buyer, address holder, uint256 tokenId) external view returns (bool blocked, address blockedBy)
```

There is no overload at the old selector, and this is a public read that a frontend or a monitor is the natural consumer of.

Why it matters

It fails safe — a caller on the old two-argument selector reverts rather than receiving a wrong answer, which is the right failure mode and why this is not higher. But it fails **silently at deploy**: nothing in the contract or the deploy flow surfaces that an off-chain consumer has just been broken, and a readiness check that quietly reverts reads the same as infrastructure trouble.

Flagging it because the re-review handoff's per-finding table records #24 as a contract fix without noting the interface change, so a reader reconciling the fix list would not see it.

Recommendation

No contract change needed if your own consumers are updated. Either:

- keep a two-argument overload that answers for the live terms, so old callers keep working and the answer is at worst conservative, or
- note the break explicitly in the release notes for the generation that ships it, and confirm the frontend and any monitor are on the three-argument form before it deploys

Worth a sweep for the same pattern: any other public view whose signature changed across `audit-v3..audit-v3-rereview` has the same property.

Remediation

Addressed by the agreed documentation route; no contract change, which is one of the two options this finding offered.

We ran the signature sweep independently and by a different method than the client's — comparing compiled ABI method signatures between the original audit baseline and the frozen generation-d commit, rather than comparing source declarations. It finds exactly five public signatures that changed shape with no overload at the old selector: the two-argument `purchaseBlocked` and `settlementBlocked`, and the previous `mint`, `mintBatch` and `mintSpecial` shapes. No further breaking signature was found. Everything else added in the generation is additive. Omitting overloads at the old selectors is consistent with the agreed route, because the retired two-argument views cannot give a token-specific answer reliably: a revert is the better failure than a confidently wrong one.

Release notes covering these changes are carried in the client's re-review addendum, and the signature sweep is recorded there as a standing item at each tag. The frozen code snapshot carries an earlier revision of that addendum; the completed deployment and evidence records are held at a later documentation commit, cited separately in §3.3. The comparison from the frozen code commit to that documentation commit shows no change under `src/`, `script/` or `test/`, so the two references describe the same reviewed code.

The client's statement that its own application and edge functions are migrated to the three-argument forms is a client assertion; it is outside this repository and was not independently verified.

[L-11] Nothing enforces the 72-hour activation delay at a production deploy: an env default, a 60s floor, and a self-skipping assertion

Fixed

Tracker: issue #40 · **Re-review ID:** RR-05 · **Verified at:** `generation-d`

Severity: Low

Location: `script/DeployBase.s.sol` `_activationDelayParam`; `test/DeployDefaults.t.sol:90`; `src/PlakxioBadge.sol` `MIN_SETTLEMENT_ACTIVATION_DELAY`

Verified at: `audit-v3-rereview-handoff`, plus the live Base Sepolia deployment

Raised after verifying the deployed bytecode. Your §9 discloses the testnet divergence fully and this is **not** a challenge to that disclosure — the question is narrower: what would stop a *mainnet* deploy from taking a short delay.

What we checked

The live deployment reads `settlementActivationDelay() = 120`, against `MIN_SETTLEMENT_ACTIVATION_DELAY() = 60`. Both expected per §9. Tracing what defends the 72-hour figure:

1. **The deploy script defaults to it but does not require it.** `vm.envOr("SETTLEMENT_ACTIVATION_DELAY", uint256(72 hours))` — omitting the variable yields 72 hours, which is the right direction for the default. Setting it to `60` is accepted on any network, including a production `NETWORK_KEY`.
2. **The contract floor is 60 seconds**, so the code refuses nothing in that range (see #3 — the 1-hour commit `7d74f21` is still unmerged).
3. **The test that asserts the default silences itself.** `test_settlementActivationDelayDefaultsToSeventyTwoHours` is guarded by `vm.skip(bytes(vm.envOr(name, string(""))).length != 0)`, so it does not run whenever the variable is exported. §9 says "We check that they actually ran", and it *is* running — our own run reports 0 skipped across 753 tests. But that is a procedural check: nothing **fails** if the assertion stops running, and a CI runner or developer shell that exports the variable turns the guarantee off silently.

So the 72-hour protection is a convention held by a default, a comment, and a test that can opt out of itself. Nothing in the contracts or the deploy path enforces it, and it has never been exercised at that value on any network.

Why it matters

Normally this would be a documentation-grade observation. It is Low rather than Informational because of what #37 did to the surrounding controls: with Special exempted from the holder's refusal, this delay is the **only contract-level protection** for the highest-value class, and your §10(j) names it as the compensating control. A compensating control whose value is set by an environment variable, floored at 60 seconds, and asserted by a self-skipping test is weaker than the sentence in §10(j) implies.

Recommendation

You already have everything needed, and it is a better fix than raising the constant alone:

Gate the floor by network. `script/network.sh` already maps `NETWORK_KEY` to one chain id and already refuses a mainnet key on a testnet RPC. Extend that: a production `NETWORK_KEY` requires `SETTLEMENT_ACTIVATION_DELAY >= 72 hours` (and the admin transfer delay >= 7 days); testnet keys may go short, as they do now. Then the divergence in §9 becomes enforced rather than conventional, and the shell harnesses that set 60 seconds keep working unchanged.

Secondary: make the skip visible rather than silent — assert on the *default path* in a harness that controls its own environment (`vm.setEnv` to clear, or a parameterised read) so the assertion cannot be turned off by the ambient shell. Note `vm.setEnv` is process-global and races under parallel tests, so prefer the parameterised read.

Related: #3 (the 60-second floor, and the unmerged 1-hour commit), #37 (why this delay is now load-bearing).

Remediation
Fixed, in both halves.

The contract half is the construction floor described under M-12, so no deployment on any network can build a badge below it whatever its environment says.

The script half. The production gate is keyed to a production network key only, accepts plain decimal seconds exclusively, strips leading zeros — the padded-value bypass the client's own adversarial round found — treats values above eighteen significant digits as above any minimum, judges the process environment before the `.env` file, and runs before any RPC read. The minima are 259,200 seconds of activation delay and 604,800 seconds of administrator transfer delay. Before promotion, the wrappers additionally read both delays back *from the chain* and refuse to promote on a failed probe rather than reasoning from one — the correct direction for a control of this kind. One shared reader now serves each delay across the

deployment and upgrade scripts, each with a parameterised parse, so the default is asserted without mutating process environment.

The self-silencing assertion this finding was named for is gone: the defaults test no longer skips itself, and the suite reported zero skipped tests. While fixing it the client found and closed a further latent defect — both upgrade scripts still defaulted the administrator transfer delay to two days after the base deployment moved to seven — which the shared reader closes.

Two things to state plainly. The 72-hour production value has still never been exercised at that value on any network; under our agreement of 13 October the rehearsal is post-review evidence and not a new precondition. And a narrower instance of the same self-skipping pattern survives in `AdminRequired.t.sol`, where the assertion that the production variable names are the ones read silences itself if a developer exports one of them locally. The substantive assertions for that finding do not skip, so this does not disturb the verified fix here; the same parameterised reader already built is the remedy.

7.3 Informational Severity Findings

[I-01] StressLive reports success when every settlement item fails

Fixed

Tracker: issue #35 · **Verified at:** `audit-v3-rereview-handoff`

Description

In `script/StressLive.s.sol#L118-L148`, the script counts successful batch items into `ok`, but every aggregate expectation is scaled by `ok` and every per-item assertion skips failed entries. It never requires any settlement to succeed before printing `all invariants held` and returning normally.

When all items fail, every observed balance delta and every expected delta is zero, so all assertions pass vacuously. A live reproduction produced:

```
settled 0 of 3
settlement USDC at rest (MUST be 0) 0
treasury delta 0
expected       0
all invariants held
ONCHAIN EXECUTION COMPLETE & SUCCESSFUL.
```

No order settled, no badge moved, and no seller or treasury payment occurred. This is an assurance defect rather than a contract vulnerability: no checked CI or release gate consumes the script, but its false-green result can mislead an operator running the advertised live stress check.

Recommendations

Require at least one successful item immediately after counting the results:

```
require(ok > 0, "STRESS FAILED: zero settlements succeeded");
```

This preserves intentional partial-success testing while rejecting the vacuous all-fail case. If the script is intended to prove complete deployment readiness, require `ok == batchSize` instead, or enforce an explicitly documented minimum success threshold.

Remediation

Fixed. `_requireAllSettled` requires every item in the batch to have settled, names each item that did not, and refuses an empty batch as proving nothing. A run in which every settlement failed can no longer report success.

8. Evidence and Limitations

The latest pass assessed seven findings and the sprint carryover, together with the additional access-control and retirement changes, at `generation-d` (`bcf18bb29265`). Seventeen earlier finding dispositions remain attributed to `audit-v3-rereview-handoff` in Appendix B; they were not re-verified in this pass. On-chain results were read from Base Sepolia (chain id 84532) at block **48,407,956**, through two independent providers (`sepolia.base.org` and `base-sepolia-rpc.publicnode.com`), which agreed on every value read, including the deployed contracts' code hash. All chain access was read-only: no transaction was sent, no key was used, and no deployed state was changed. Simulated calls were executed as `eth_call` and alter nothing.

8.1 Deployed bytecode against the frozen source

We compiled the frozen source and compared it to the deployed runtime bytecode, masking immutables at their compiler-recorded offsets:

Contract	Address	Runtime length	Immutable bytes masked	Executable code identical
`PlakxioBadge`	`0xBE47dc08d018F2f9a7667Af767b2B029c9e016cF`	18,494	288	Yes
`PlakxioSettlement`	`0x687c4568Be0514Aaa975DB91F018E7518fc1860C`	19,888	928	Yes
`PlakxioSpecialMarket`	`0x9E583F48107f6C9538c31EB7F5eeC079F767c4C6`	12,042	1,216	Yes

The only residual differences lie inside each contract's 53-byte trailing compiler metadata region, which varies with source paths and does not affect execution.

8.2 Access-control enumeration and the retirement checks

The generation adds enumerable role membership to the badge and both markets, chained under the existing two-step administrator rules. We verified the chaining by execution rather than inference: a role grant the administrator rules refuse never enters the member set, the two-step handover moves the set without leaving a stale member, and a 256-run randomised test found no case where enumerated membership and the authoritative role check disagree. That last property is the one the retirement pipeline now depends on, which is why it was tested rather than assumed.

The privilege surface is unchanged. The extension adds exactly three read-only functions per contract and removes nothing; no new state-changing function was introduced, and the administrator rules retain the last word on every grant, revocation and renunciation.

The retirement checks now treat an unreadable log source as a finding rather than a note on a contract that cannot enumerate its role holders, read role membership from state instead of from a provider's log index, cross-check the enumerated set against the authoritative role check and treat disagreement as a finding, and exit non-zero when a run absorbed warnings. Each of these was a specific request from the re-review and each is implemented.

8.3 Tests and checks actually run

Check	Result
Build at the frozen commit	Clean
Client test suite, default profile	825 passed, 0 failed, 0 skipped
Our own independent suite (12 tests, including a 256-run fuzz)	12 passed, 0 failed
Deployed-bytecode comparison, three contracts	Executable code identical
Live reads and simulated gate calls, two providers	All values agreed
Independent compiled-ABI signature sweep	Exactly five breaking signatures

8.4 Limitations

These are stated so the conclusion is not read more widely than it was established.

1. **Base mainnet is not deployed and was not reviewed.** Nothing here is a mainnet approval.
2. **Two extended retirement harnesses did not complete.** `retire_superseded.sh` stopped at a 60-minute limit after 160 passing assertions with no failures, at case 22 of its sequence; `post_retire_check.sh` stopped at a 40-minute limit having produced no output, for a cause we did not determine. Neither is counted as a pass anywhere in this report, and the retirement dispositions in §8.2 rest on cited code inspection instead.
3. **Seventeen of the twenty-four findings were verified at `audit-v3-rereview-handoff`, not at the frozen commit.** They are recorded as such in Appendix B. This pass re-verified seven findings plus the sprint carryover at `generation-d`, together with the access-control and retirement changes the generation introduced. It was not a re-audit of the whole tree.

4. **The client's mutation-testing result was not reproduced** by this review.
5. **Client assertions not independently verified**, and labelled as assertions wherever they appear: the application and edge-function migration to the new signatures; client-facing refusal copy and its rendered behaviour; off-chain alerting cadence, including the external-monitor gap figures that the six-hour floor is argued against; administrator key custody and threshold policy; and continuous-integration state.
6. **Passing a supplied test suite is not a security proof** and is not treated as one here. The dispositions rest on code reading, our own tests, and on-chain reproduction.
7. **The 72-hour production activation delay has never been exercised at that value on any network.** Under the agreement of 13 October it is post-review rehearsal evidence, not a precondition of this report.

Appendix A — August Sprint Carryover Re-verified

The Founder Security Sprint of 25–26 August 2026 was a separate, earlier engagement phase with its own finding identifiers, and its sixteen findings were reported and closed in the sprint report. They are **not** part of the 24 findings counted in this document.

One of them was re-opened for a specific condition and is therefore re-verified here: the sprint's **H-01**, whose remaining condition — the interim 60-second activation-delay floor — was the item this review was asked to treat first. It is reported in this appendix, under the sprint's own identifier, so it is not confused with the full-audit identifier space.

A.1 [Sprint H-01] Any whitelisted operator can take any holder's badges, and the holder's revocation is unreadable

Original severity: High (adjusted one tier down at the time for its fully-trusted administrator precondition). **Tracker:** issue #3. **Raised against:** `audit-v1.1`.

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.

PoC

- **Apply / run:** add as `test/poc/OperatorGrantConfiscation.t.sol`, then `forge test --match-path "test/poc/OperatorGrantConfiscation.t.sol" -vv`
- **Asserts:** the harm — the holder's balance goes to zero across three classes, the grantee's goes to one, and the grantee transferred no value. A second test asserts the holder has no self-rescue route.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test} from "forge-std/Test.sol";
import {PlakxioBadge} from "../../src/PlakxioBadge.sol";

contract OperatorGrantConfiscationTest is Test {
    PlakxioBadge internal badge;
    address internal admin = makeAddr("admin");
    address internal minter = makeAddr("minter");
    address internal victim = makeAddr("victim");
    address internal grantee = makeAddr("grantee"); // plain EOA, no code

    uint256 internal constant NORMAL_ID = 7;
    uint256 internal constant PREMIUM_ID = 1_000_000_007;
    uint256 internal constant SPECIAL_ID = 2_000_000_007;

    function setUp() public {
        badge = new PlakxioBadge(admin, "uri");
        bytes32 minterRole = badge.MINTER_ROLE();
        vm.prank(admin);
        badge.grantRole(minterRole, minter);
        vm.startPrank(minter);
        badge.mint(victim, NORMAL_ID, 1, "");
        badge.mint(victim, PREMIUM_ID, 1, "");
        badge.mintSpecial(victim, SPECIAL_ID, 99);
        vm.stopPrank();
    }

    function test_H01_whitelistedEoaTakesEveryBadgeWithoutSignatureOrPayment()
public {
        vm.prank(victim);
        badge.setApprovalForAll(grantee, false);
        assertTrue(badge.isApprovedForAll(victim, grantee) == false, "not yet
whitelisted");

        uint256 maskAll = badge.CLASS_MASK_ALL();
        vm.prank(admin);
        badge.setSettlementContract(grantee, true, maskAll);

        assertTrue(badge.isApprovedForAll(victim, grantee), "revocation ignored");

        uint256[] memory ids = new uint256[](3);
        uint256[] memory amts = new uint256[](3);
        ids[0] = NORMAL_ID; ids[1] = PREMIUM_ID; ids[2] = SPECIAL_ID;
        amts[0] = 1; amts[1] = 1; amts[2] = 1;
```

```
        vm.prank(grantee);
        badge.safeBatchTransferFrom(victim, grantee, ids, amts, "");

        assertEq(badge.balanceOf(victim, NORMAL_ID), 0, "normal taken");
        assertEq(badge.balanceOf(victim, PREMIUM_ID), 0, "premium taken");
        assertEq(badge.balanceOf(victim, SPECIAL_ID), 0, "special taken");
        assertEq(badge.balanceOf(grantee, NORMAL_ID), 1);
        assertEq(badge.balanceOf(grantee, PREMIUM_ID), 1);
        assertEq(badge.balanceOf(grantee, SPECIAL_ID), 1);
        assertEq(victim.balance, 0, "victim received no native payment");
    }

    function test_H01_holderCannotSelfRescue() public {
        vm.prank(victim);

vm.expectRevert(abi.encodeWithSelector(PlakxioBadge.NotSettlementContract.selector,
victim));
        badge.safeTransferFrom(victim, address(0xdead), NORMAL_ID, 1, "");
    }
}
```

- **Observed output:**

```
Ran 2 tests for
test/poc/OperatorGrantConfiscation.t.sol:OperatorGrantConfiscationTest
[PASS] test_H01_holderCannotSelfRescue() (gas: 14526)
[PASS] test_H01_whitelistedEoaTakesEveryBadgeWithoutSignatureOrPayment() (gas:
165672)
Suite result: ok. 2 passed; 0 failed; 0 skipped
```

- **Baseline:** the existing suite is unaffected — `forge test --no-match-path "test/poc/*"` reports `182 tests passed, 0 failed, 0 skipped`.
- **Fix verified:** changing the `||` at L259 to `&&` — the change described under Recommendations as Option A — makes the PoC fail at `revocation ignored`, i.e. the holder's revocation is honoured and the transfer no longer authorises. The PoC is pinned to that line.

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

Fixed for the activation-delay floor condition.

The badge now enforces a six-hour minimum (21,600 seconds) and a 365-day maximum activation delay, refused at construction. Our own boundary tests confirmed rejection at 21,599 and acceptance at 21,600 with the expected named errors, and both providers returned the 21,600 minimum and configured delay from the deployed badge. The prior 60-second-floor concern is resolved in the reviewed implementation. Production deployment of 72 hours is enforced separately and is reported under L-11.

This does not introduce mandatory consent. A holder who has never opted out remains exposed to a wrong whitelist grant, and the delayed, fully-trusted administrator model is unchanged — the administrator transfer is two-step and delayed, which bounds but does not remove that exposure. The claimed off-chain alerting cadence that the six-hour figure is argued against is client- supplied and was not independently verified.

Appendix B — Identifier Mapping and Status Ledger

Every finding identifier in the engagement is accounted for below, so that this document can be reconciled against the tracker without inference. Report identifiers from the 18 September full audit are **unchanged**; the five findings raised in the October re-review were appended within their severity rather than renumbered over the originals.

B.1 This report's findings — identifier mapping and status ledger

The *verified at* column records the commit at which each status was established by Zealynx. It is not derived from a client label: a label states what the client believes was fixed, and this column states where we confirmed it.

Report ID	Tracker issue	Re-review ID	Raised in	Status	Verified at
M-01	#17	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-02	#18	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-03	#21	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-04	#22	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-05	#26	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-06	#27	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-07	#28	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-08	#29	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-09	#30	—	Full audit	Fixed	`audit-v3-rereview-handoff`

Report ID	Tracker issue	Re-review ID	Raised in	Status	Verified at
M-10	#31	—	Full audit	Fixed	`audit-v3-rereview-handoff`
M-11	#36	RR-01	Re-review	Fixed · residual	`generation-d`
M-12	#37	RR-02	Re-review	Partly addressed	`generation-d`
L-01	#19	—	Full audit	Fixed · residual	`generation-d`
L-02	#20	—	Full audit	Fixed	`audit-v3-rereview-handoff`
L-03	#23	—	Full audit	Fixed	`audit-v3-rereview-handoff`
L-04	#24	—	Full audit	Fixed	`audit-v3-rereview-handoff`
L-05	#25	—	Full audit	Fixed	`audit-v3-rereview-handoff`
L-06	#32	—	Full audit	Fixed	`generation-d`
L-07	#33	—	Full audit	Fixed	`audit-v3-rereview-handoff`
L-08	#34	—	Full audit	Fixed	`audit-v3-rereview-handoff`
L-09	#38	RR-03	Re-review	Fixed	`generation-d`
L-10	#39	RR-04	Re-review	Addressed · docs	`generation-d`
L-11	#40	RR-05	Re-review	Fixed	`generation-d`

Report ID	Tracker issue	Re-review ID	Raised in	Status	Verified at
I-01	#35	—	Full audit	Fixed	`audit-v3-rereview-handoff`

Status vocabulary. *Fixed* — the remediation holds at the stated commit. *Fixed · residual* — the remediation holds and a named, bounded residual is retained and documented. *Partly addressed* — the stated condition is implemented, and a named exposure is retained by design. *Addressed · docs* — closed by the agreed documentation route with no code change.

B.2 Earlier engagement phase — August Founder Security Sprint

These sixteen findings belong to the sprint phase and its own identifier space. They are outside this report's 24 and are listed only so the boundary is explicit.

Tracker issue	Sprint ID	Disposition in this document
#1	L-01	Reported and closed in the sprint report
#2	L-02	Reported and closed in the sprint report
#3	H-01	Re-verified in this review — Appendix A
#4	H-02	Reported and closed in the sprint report
#5	M-01	Reported and closed in the sprint report
#6	M-02	Reported and closed in the sprint report
#7	M-03	Reported and closed in the sprint report
#8	M-04	Reported and closed in the sprint report
#9	M-05	Reported and closed in the sprint report
#10	M-06	Reported and closed in the sprint report
#11	L-03	Reported and closed in the sprint report
#12	L-04	Reported and closed in the sprint report
#13	L-05	Reported and closed in the sprint report
#14	L-06	Reported and closed in the sprint report

Tracker issue	Sprint ID	Disposition in this document
#15	L-07	Reported and closed in the sprint report
#16	L-08	Reported and closed in the sprint report

B.3 Coverage statement

The tracker holds 40 issues: 16 from the August sprint (B.2), 19 from the 18 September full audit, and 5 raised in the October re-review. This report's §7 carries the 19 + 5 = 24, and Appendix A carries the one sprint finding re-verified in this pass. No tracker issue in the reviewed scope is unaccounted for.

Tracker open/closed state is administrative and separate from the technical dispositions recorded here. It is not evidence either way about the statuses recorded above.