ZEALYNX

Web3 Security & Smart Contract Development

Dripster

Smart Contract Audit

28 April 2026

Zealynx

contact@zealynx.io

Carlos (Bloqarl)	@TheBlockChainer
Bobby_Petrov	@0x_bob_0x

Contents

1. About Zealynx

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

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

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

2. Disclaimer

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

3. Overview

3.1 Project Summary

Zealynx Security conducted a security audit of the Dripster Leveraged Prediction Vault — a custodial EVM smart-contract suite enabling leveraged positions on Polymarket Conditional Token Framework markets. The audit covered the core vault contract and six supporting libraries.

3.2 About Dripster

Dripster (Dimes.fi) is a leveraged prediction-market vault on Polygon. Users deposit USDC; an ML-driven backend opens leveraged positions on Polymarket using protocol capital alongside the user's collateral. The vault custodies USDC and conditional tokens through a fourteen-state position lifecycle, with backend-signed EIP-712 payloads gating value-changing operations and timelock-gated control of upgrades, capital, and admin rotation.

3.3 Audit Scope

The code under review is composed of seven Solidity contracts totaling **~1,825 nSLOC**: `LeveragedPredictionVaultV1.sol`, `PositionLib.sol`, `SignatureLib.sol`, `CloseLib.sol`, `Events.sol`, `Types.sol`, and `Errors.sol`.

Repository: https://github.com/bloom-art/dripster-lend
Branch: `evm-only-for-audit`
Commit: `4b5dd35ed922f34c8c589241000b6e37a4c35d68`

Out of scope: frontend/UI, ML models, third-party dependencies, off-chain infrastructure, and the API/backend (audited separately).

4. Audit Methodology

Approach

Zealynx applies a multi-layer audit methodology that combines manual review with multiple automated verification techniques, scaled to the protocol's risk surface and time budget. Each layer catches a different class of bug; running them in concert provides depth that no single technique can match.

Manual review

- Line-by-line code reading of every contract in scope.
- End-to-end scenario walkthroughs of every documented user flow, traced from on-chain initiation through finalization.
- Adversarial deep-dives on each high-risk attack surface (signatures, math/boundaries, state machine, capital accounting, upgrade path).
- Application of external audit heuristic checklists and pattern-matching against historical findings from comparable protocols.

Automated and formal verification

- Static analysis (Slither + custom rules) integrated into CI.
- Custom Foundry invariant fuzzing, with each invariant calibrated against a deliberately-seeded bug before its passing result is trusted.
- Mutation testing (Trail of Bits *mewt*) to verify the existing test suite actually catches behavioral changes in the contract.
- Stateful protocol-level fuzzing (Medusa) for long-sequence violations across millions of method invocations.
- Symbolic execution (Kontrol) to formally prove core mathematical identities for all inputs.

Throughout the audit we prioritize the following aspects to uphold excellence:

1. Code Quality: We diligently evaluate the overall quality of the code, aiming to identify potential vulnerabilities or weaknesses.

2. Best Practices: Our assessments emphasize adherence to established best practices, ensuring 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.

4. Defense in Depth: Beyond a single review pass, we apply independent verification layers and require each layer to demonstrate sensitivity to a deliberately-introduced bug before trusting its passing results.

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

The audit applied a multi-layer methodology: manual review, Krait automated pre-audit, six custom Foundry invariants (each calibrated against a deliberately-seeded bug), mutation testing (Trail of Bits *mewt*, 99% high-severity catch rate), stateful fuzzing (Medusa, 4.5M invocations), and symbolic execution (Kontrol, 15 proofs passed).

6.1 The Key Findings

Emergency mode blocks escape-hatch refund paths, trapping user funds: The `whenNotEmergencyOnly` modifier is applied to `cancelPosition`, `revertOpen`, and `finalizeClose` — the very functions that exist to refund funds caught in transitional states during incidents. While the team accepts the asymmetry as intended for capital-touching paths, we maintained that `cancelPosition` (which has no protocol capital exposure) should be reachable during outages; the team agreed to add a self-cancel path gated behind the existing `serverHalted` flag, preserving the conservative posture during normal operation while giving users a non-custodial exit during a declared incident.

Storage slot annotations off, creating a V2 upgrade hazard: All 28 inline slot annotations in the contract assumed OpenZeppelin v4 sequential base storage rather than v5 ERC-7201 namespaced storage. A V2 author trusting the comments could place a new variable at "slot 0", overwriting `globalAdmin` after `upgradeToAndCall` and corrupting governance.

Incomplete `rescueERC1155` guard allows draining tokens from active positions: The original guard used `tokenTotalBorrowedUsdcUnits[tokenId] > 0` as a proxy for "active position holds this tokenId". After a force-unwind drains borrowed capital to zero while the position retains its conditional tokens, the proxy reads zero and `globalAdmin` can rescue tokens out from under the position — bricking every subsequent close, settle, or liquidate. The same surface was independently

flagged by the Krait pre-audit from the inverse angle, confirming the rescue guard as the correct remediation site.

6.2 Architectural Security Observations

During the assessment, we observed several architectural decisions and controls that contribute to Dripster's security posture:

Graduated Four-Tier Operational Controls: Pause mechanisms are tiered (`newPositionsPaused` → `userActionsPaused` → `emergencyOnlyMode` → `serverHalted`), allowing scoped incident response rather than a binary kill switch.

Asymmetric Admin Authority with Timelock Gating: Four-role separation plus a `TimelockController` on every high-risk operation. `emergencyAdmin` can pause but only `globalAdmin` can unpause — preventing a hot key from re-enabling the protocol after a halt.

Two-Step Backend-Coordinated Lifecycle: Position transitions split into on-chain initiation, off-chain Polymarket execution (Fill-Or-Kill, no resting-order windows), and on-chain finalization, with explicit `*Pending` states.

EIP-712 Typed-Data Signatures with Chain-Bound Domain: Position payloads are signed via EIP-712 with the OpenZeppelin v5 namespaced domain separator — wallet-side transparency plus cross-chain replay prevention via `chainid()` and `address(this)`.

Concurrent Unwind+Liquidate State Isolation: Separate `pendingTokenUnits` and `pendingUnwindTokenUnits` fields permit a force-unwind and liquidation to interleave without data corruption; terminal `finalizeForceUnwind` from `Liquidated` avoids double-decrementing capital tracking.

Contract-Wide Invariant Enforcement: `_assertPositionInvariants` runs at every state-mutating exit, locking in core position identities.

6.3 Security Strengths Observed

Based on our assessment, Dripster demonstrates the following security strengths:

Iteratively-Hardened Codebase: Extensive `AUDIT-FIX` annotations reflect multiple prior security review iterations. The Krait pre-audit (1 Low fixed, 4 Informational acknowledged) predated manual review, showing automated tooling is part of the development loop.

Deferred Fee Accrual Eliminates Create→Open Race: Origination fees are reserved as `pendingOriginationFeesUsdcUnits` at `createPosition` and only realized into `accumulatedFeesUsdcUnits` at `finalizeOpen`, with refund on partial fills.

Modern Solidity Practices: Custom errors, `SafeERC20`, `nonReentrant` on every state-mutating entry point, strict CEI ordering, and OpenZeppelin v5 upgradeable patterns with `_disableInitializers()` in the implementation constructor.

Defense-in-Depth on Backend Trust: Soft sanity checks (`isWithinBounds` at 5×/10× tolerance) catch inconsistent submissions; off-chain `VaultEventAlertService` monitors every finalize event with automatic circuit-breaker tripping at 5,000 USDC bad debt.

High Test-Suite Sensitivity: Mutation testing yielded a 99% high-severity catch rate. Surviving mutants were classified as equivalent or covered by the calibrated invariant layer added during this audit.

Summary of Findings:

Vulnerability	Severity	Status
[M-1] Emergency mode blocks escape-hatch refund paths, trapping user funds	Medium	Acknowledged
[M-2] Incorrect storage slot annotations create high-risk reference hazard for V2 upgrade	Medium	Fixed
[L-1] cancelPosition() escape hatch is inaccessible to the user whose funds are locked	Low	Fixed
[L-2] USDC sent directly to the vault is permanently irrecoverable	Low	Acknowledged
[L-3] createPosition accepts user funds without checking capital availability or limits	Low	Acknowledged
[L-4] Missing revert mechanism for pending states allows permanent position lockup on backend failure	Low	Acknowledged
[L-5] Incomplete rescueERC1155 guard allows extraction of active position tokens after full deleverage	Low	Fixed
[L-6] Force unwind unconditionally resets position state to Opened, discarding user's prior CloseRequested intent	Low	Fixed
[L-7] finalizeLiquidate leaves actualNotionalUsdcUnits stale post-liquidation, creating a latent fee-computation bug and inconsistent off-chain reads	Low	Fixed
[I-1] cancelDelaySeconds changes apply retroactively to existing positions	Info	Fixed

Vulnerability	Severity	Status
[I-2] No update path creates a chicken-and-egg risk with Timelock address being immutable	Info	Fixed
[I-3] Coarse-grained emergency mode forces all position exits through liquidation path, causing users to pay unwarranted liquidation fees	Info	Acknowledged
[I-4] Missing post-unwind leverage validation allows ineffective force unwinds to complete silently	Info	Fixed
[I-5] Invariant suite handler uses legacy EIP-191 signatures, silently reverts every position creation, yields zero security signal	Info	Fixed
[I-6] EIP-191 personal-sign instead of EIP-712 typed data reduces wallet transparency	Info	Fixed
[I-7] No on-chain cumulative bad-debt tracking	Info	Acknowledged
[I-8] Partial fill can result in leverage below MIN_LEVERAGE_BPS	Info	Acknowledged

7. Audit Findings

7.1 Medium Severity Findings

[M-01] Emergency mode blocks escape-hatch refund paths, trapping user funds

Affected files

LeveragedPredictionVaultV1.sol#L533
LeveragedPredictionVaultV1.sol#L608
LeveragedPredictionVaultV1.sol#L717
LeveragedPredictionVaultV1.sol#L781
LeveragedPredictionVaultV1.sol#L850
LeveragedPredictionVaultV1.sol#L882

Description

The protocol implements a multi-tiered pause system where `emergencyOnlyMode` acts as a circuit breaker for exceptional situations. The design correctly allows risk-reduction operations (`forceUnwind`, `liquidatePosition`, `settlePosition` and their finalize counterparts) to bypass emergency mode so the protocol can continue shedding risk during incidents.

However, the `whenNotEmergencyOnly` modifier is also applied to `cancelPosition`, `revertOpen`, `openPosition`, `finalizeOpen`, `closePosition`, and `finalizeClose` — the very functions that serve as escape hatches for positions caught mid-lifecycle.

This creates a contradiction: `cancelPosition` was specifically implemented as an escape hatch to prevent permanent fund locks when the admin is delayed (up to

`cancelDelaySeconds`). `revertOpen` is the designated failure path to refund users when a Polymarket exchange order fails. During an active emergency — such as a backend crash, database outage, or exchange API failure — the system is *most likely* to suffer from the exact delays and failed orders that these functions were built to resolve.

Vulnerable Scenario:
1. User calls `createPosition`, transferring collateral + origination fee to the vault. Position enters `Created` state.
2. An incident occurs (backend crash, exchange outage) and `emergencyAdmin` (hot wallet) activates `emergencyOnlyMode`.
3. The user's position is stuck in `Created` — `cancelPosition` is blocked by `whenNotEmergencyOnly`, so neither the user nor the admin can refund the escrowed funds.
4. Similarly, positions in `OpenPending` cannot be refunded via `revertOpen`, and positions in `ClosePending` cannot be completed via `finalizeClose`.
5. The only recovery is for `globalAdmin` (2-of-3 cold-wallet multisig) to disable emergency mode entirely via `setEmergencyOnlyMode(false)` — which could take hours to coordinate and forces a binary choice between maintaining emergency posture and returning user funds.

The asymmetry is illogical: if the protocol trusts admins to finalize liquidations, force unwinds, and settlements during emergency mode, there is no security benefit to preventing those same admins from issuing simple refunds for stuck positions.

Impact

User USDC (collateral + origination fee) is locked in `Created`, `OpenPending`, or `ClosePending` states with no on-chain recovery path while emergency mode is active. Recovery requires the cold-wallet multisig to fully disable the protocol's

emergency defenses — defeating the purpose of the circuit breaker during the incident window when it is needed most.

Recommendations

Exempt the refund/recovery functions from the `whenNotEmergencyOnly` modifier while keeping it on position-initiation functions:

> Remove `whenNotEmergencyOnly` from `cancelPosition`, `revertOpen`, and `finalizeClose` — these are fund-recovery paths that pose no new risk during emergencies (they only return escrowed funds or complete already-initiated operations).
>
> Keep `whenNotEmergencyOnly` on `createPosition`, `openPosition`, and `closePosition` — these initiate new lifecycle transitions that should remain blocked during emergencies.

Alternatively, create dedicated emergency-exempt recovery functions (`emergencyCancelPosition`, `emergencyRevertOpen`) that bypass the modifier while preserving the existing function signatures for non-emergency use.

Status

Acknowledged. Team agreed to add a self-cancel path on `cancelPosition` gated by the existing `serverHalted` flag, with tests covering (a) self-cancel reverts when the flag is unset, (b) self-cancel succeeds and refunds collateral + origination fee when the flag is set, and (c) the flag itself is admin-gated. The remaining emergency-mode asymmetry on `revertOpen` and `finalizeClose` is accepted by design.

[M-02] Incorrect storage slot annotations create high-risk reference hazard for V2 upgrade

Affected files

LeveragedPredictionVaultV1.sol#L69-L122

Description

The storage layout section contains inline comments documenting the EVM storage slot for every state variable. All 28 slot annotations are systematically incorrect — every comment is off by exactly 4 slots.

The comments were written assuming OpenZeppelin v4 behavior, where `UUPSUpgradeable`, `ReentrancyGuardUpgradeable`, `PausableUpgradeable`, and `EIP712Upgradeable` each reserved sequential slots (0–3). OZ v5 replaced this with ERC-7201 namespaced storage, where each base contract stores its state at a deterministic hash-derived location rather than in sequential slots. This means the contract's own variables begin at slot 0, not slot 4.

```
// Note: UUPSUpgradeable uses slots 0-1, ReentrancyGuard
slot 2, Pausable slot 3
// ↑ INCORRECT: OZ v5 base contracts use ERC-7201 hashed
storage, not slots 0-3

// Admin addresses (slots 4-7)
address public globalAdmin;    // slot 4  ← actual: slot
0
address public emergencyAdmin; // slot 5  ← actual: slot
1
address public appAdmin1;      // slot 6  ← actual: slot
```

```
2
address public appAdmin2;      // slot 7  ← actual: slot
3

// External contracts (slots 8-10)
address public usdcToken;           // slot 8   ← actual:
slot 4
address public conditionalTokens;   // slot 9   ← actual:
slot 5
address public capitalPool;         // slot 10  ← actual:
slot 6

// ... (pattern continues for all 28 annotated variables)

// Storage gap for upgrades (slots 32-65)
uint256[34] private _gap;           // ← gap actually
starts at slot 24, ends at slot 57
```

There is no runtime impact on V1. The actual storage layout is correct, the gap is correctly sized, and no contract logic depends on slot number comments. The risk is entirely in the upgrade path.

Vulnerable Scenario:

1. A developer begins work on V2 and reads the storage layout comments to understand which slots are occupied.
2. The comments say slots 0–3 are reserved for OZ base contracts — the developer believes these are "OZ-reserved" and safe to use for V2 variables.
3. The developer places a new V2 variable at slot 0 (thinking it's free) — but `globalAdmin` is already there.
4. After a UUPS `upgradeToAndCall`, the new variable overwrites `globalAdmin` in the proxy's storage, corrupting protocol governance and potentially granting control to an attacker or permanently disabling admin access.

PoC (verified with `forge test`):

```
function test_globalAdmin_at_slot_0_not_slot_4() public
view {
    // Read slot 0 directly from the proxy
    bytes32 slot0Value = vm.load(address(vault),
bytes32(uint256(0)));
    address globalAdminFromSlot0 =
address(uint160(uint256(slot0Value)));
    // globalAdmin IS at slot 0, not slot 4 as comments
claim
    assertEq(globalAdminFromSlot0, globalAdmin);
}

function test_verify_sequential_admin_slots_from_zero()
public view {
    bytes32 slot0 = vm.load(address(vault),
bytes32(uint256(0)));
    bytes32 slot1 = vm.load(address(vault),
bytes32(uint256(1)));
    bytes32 slot2 = vm.load(address(vault),
bytes32(uint256(2)));
    bytes32 slot3 = vm.load(address(vault),
bytes32(uint256(3)));

    assertEq(address(uint160(uint256(slot0))),
globalAdmin);      // slot 0, not 4
    assertEq(address(uint160(uint256(slot1))),
emergencyAdmin);   // slot 1, not 5
    assertEq(address(uint160(uint256(slot2))),
appAdmin1);        // slot 2, not 6
```

```
    assertEq(address(uint160(uint256(slot3))),
appAdmin2);        // slot 3, not 7
}
```

Impact

A V2 developer relying on these comments to plan new storage additions could overwrite the four admin address slots (slots 0–3), corrupting `globalAdmin`, `emergencyAdmin`, `appAdmin1`, or `appAdmin2` in a live upgrade. The `_gap` range annotation (`slots 32–65`) is also incorrect (actual: `slots 24–57`), potentially misleading V2 capacity planning.

Recommendations

Update the storage section header comment and all 28 inline slot annotations to reflect the actual OZ v5 ERC-7201 layout (apply −4 correction to all slot numbers).

Add `forge inspect LeveragedPredictionVaultV1 storage-layout` to the upgrade checklist as a mandatory pre-deployment verification step to prevent future comment drift.

Status

Fixed. Slot annotations corrected; OpenZeppelin Foundry-upgrades plugin validates layout in CI.

7.2 Low Severity Findings

[L-01] cancelPosition() escape hatch is inaccessible to the user whose funds are locked

Affected files

LeveragedPredictionVaultV1.sol#L779-L816

Description

`cancelPosition()` at L779 was designed as a safety mechanism to prevent user funds from being permanently locked when the admin fails to call `openPosition()` after a user creates a position. However, the function uses the `onlyAppAdminOrAbove` modifier — meaning only the admin can call it. The user who owns the position and whose money is locked cannot call it themselves.

```
function cancelPosition(
    bytes32 positionKey
) external nonReentrant onlyAppAdminOrAbove
whenServerNotHalted whenNotEmergencyOnly {
//                      ^^^^^^^^^^^^^^^^^^^
//                      User cannot call this
```

Additionally, the function is blocked by `whenServerNotHalted` and `whenNotEmergencyOnly`, so the escape hatch is also disabled during the exact emergency scenarios where it's most needed.

Recommendations

Allow the position owner to call `cancelPosition` after the delay, while preserving admin access:

```
function cancelPosition(
    bytes32 positionKey
) external nonReentrant whenServerNotHalted
whenNotEmergencyOnly {
    Types.PositionVault storage position =
_requirePositionExists(positionKey);
    _requirePositionState(positionKey,
Types.PositionState.Created);

    // Allow position owner (after delay) OR admin (after
delay)
    if (msg.sender != position.owner) {
        _checkAppAdminOrAbove();
    }

    uint256 cancelAfter = uint256(position.createdAt) +
cancelDelaySeconds;
    if (block.timestamp < cancelAfter) {
        revert CancelDelayNotElapsed(positionKey,
position.createdAt, cancelAfter, block.timestamp);
    }
    // ... rest unchanged
}
```

Status

Fixed. Position owner can now self-cancel after `cancelDelaySeconds`; admin path preserved.

[L-02] USDC sent directly to the vault is permanently irrecoverable

Affected files

LeveragedPredictionVaultV1.sol#L1616-L1623

Description

The `rescueERC20` function explicitly blocks USDC recovery with `CannotRescueOperationalToken` at L1620. This is intentional — the contract holds operational USDC and a naive rescue function could drain protocol balances. However, no alternative rescue mechanism exists, meaning any USDC that enters the contract outside the tracked accounting paths is permanently stuck.

This can happen through a user accidentally sending USDC to the contract address instead of calling `createPosition`, or a frontend/integration bug routing funds to the wrong address.

The contract tracks USDC across five buckets: `capitalPoolBalanceUsdcUnits`, `accumulatedFeesUsdcUnits`, `pendingOriginationFeesUsdcUnits`, per-position `collateralBalanceUsdcUnits` (Created state only), and per-position `pendingRefundUsdcUnits`. Any USDC outside these buckets is inaccessible — but two of these buckets (`collateralBalanceUsdcUnits` and `pendingRefundUsdcUnits`) exist only at the position level with no global accumulator, so the contract cannot compute its own tracked total on-chain to distinguish stuck funds from operational funds.

Recommendations

Add two global accumulator state variables to track the per-position USDC buckets:

```
uint256 public totalCollateralBalanceUsdcUnits;
uint256 public totalPendingRefundUsdcUnits;
```

Maintain them at all mutation points:

`totalCollateralBalanceUsdcUnits` (3 sites): increment at `createPosition`, decrement at `openPosition` (L575), decrement at `cancelPosition`.

`totalPendingRefundUsdcUnits` (3 sites): increment at `finalizeOpen` (partial fill refund), increment at `finalizeForceUnwind` (non-terminal path), decrement at `CloseLib.finalizePositionClose` (when `pendingRefundUsdcUnits` is zeroed).

Then add a timelock-gated rescue that can only withdraw the provably untracked excess:

```
function rescueUSDC(uint256 amountUsdcUnits) external
nonReentrant onlyTimelock {
    uint256 trackedTotal = capitalPoolBalanceUsdcUnits
        + accumulatedFeesUsdcUnits
        + pendingOriginationFeesUsdcUnits
        + totalCollateralBalanceUsdcUnits
        + totalPendingRefundUsdcUnits;

    uint256 actualBalance =
```

```
IERC20(usdcToken).balanceOf(address(this));
    uint256 excess = actualBalance - trackedTotal; //
reverts on underflow

    if (amountUsdcUnits > excess) revert
InsufficientExcess();
    IERC20(usdcToken).safeTransfer(globalAdmin,
amountUsdcUnits);
}
```

This preserves the original safety concern — admin cannot touch operational USDC — while making accidentally transferred USDC recoverable.

Status

Acknowledged. USDC rescue not added on contract-size grounds; existing rescue paths hardened to `onlyTimelock`.

[L-03] createPosition accepts user funds without checking capital availability or limits

Affected files

LeveragedPredictionVaultV1.sol#L428-L522

Description

When a user calls `createPosition()`, their USDC (collateral + origination fee) is transferred into the contract at L503 without any check on whether the protocol can fund the requested leverage. Both the capital pool balance check (`capitalPoolBalanceUsdcUnits < protocolCapital` at L542) and the four capital limit checks (`validateCapitalLimits` at L547 — per-user, per-market, per-token, and global caps) only occur later when the admin calls `openPosition()` in a separate transaction that the user has no visibility into.

If any of these checks fail, `openPosition` reverts on the admin side. The user receives no on-chain feedback — their funds remain locked in `Created` state. Recovery requires a manual `cancelPosition` call (currently admin-only per L-01) after `cancelDelaySeconds` elapses (default 10 minutes).

The capital pool is funded manually via `addCapital()` by the globalAdmin (2-of-3 cold wallet multisig). There is no automated replenishment, no minimum reserve requirement, and no utilization circuit breaker. During demand surges the pool can drain to zero through normal usage, causing all new position opens to fail simultaneously while multiple users have funds locked.

Recommendations

Add capital availability and limit checks to `createPosition` before transferring user funds:

```
// In createPosition(), before the safeTransferFrom:
uint256 protocolCapitalUsdcUnits = notionalUsdcUnits -
collateralUsdcUnits;
if (capitalPoolBalanceUsdcUnits <
protocolCapitalUsdcUnits) {
    revert InsufficientCapital();
}
PositionLib.validateCapitalLimits(
    userTotalBorrowedUsdcUnits,
    marketTotalBorrowedUsdcUnits,
    tokenTotalBorrowedUsdcUnits,
    totalProtocolBorrowedUsdcUnits,
    maxUserCapitalUsdcUnits,
    maxMarketCapitalUsdcUnits,
    maxTokenCapitalUsdcUnits,
    maxTotalCapitalUsdcUnits,
    msg.sender,
    marketId,
    tokenId,
    protocolCapitalUsdcUnits
);
```

This does not replace the checks in `openPosition` — both are needed since pool balance and limits can change between calls. The early checks prevent the common case where a user locks funds into a position that is already unfundable at creation time.

Status

Acknowledged. Backend pre-flight is the primary gate; recovery available via L-01 user self-cancel.

[L-04] Missing revert mechanism for pending states allows permanent position lockup on backend failure

Affected files

LeveragedPredictionVaultV1.sol#L965-L996 (settlePosition → SettlePending)
LeveragedPredictionVaultV1.sol#L843-L870 (closePosition → ClosePending)
LeveragedPredictionVaultV1.sol#L1133-L1170 (liquidatePosition → LiquidatePending)

Description

When `settlePosition` is called, the position transitions to `SettlePending` and all conditional tokens are transferred to the admin for off-chain redemption (L985). If `finalizeSettle` is never called — due to backend crash, key rotation invalidating the pre-signed signature, or operational error — the position is permanently stuck.

No other function accepts `SettlePending` state. There is no `revertSettle`, no timeout, and no user-initiated recovery path. The admin holds the tokens (or has already redeemed them for USDC off-chain), and the user has no on-chain recourse.

This contrasts with `OpenPending`, which has `revertOpen` (L714) as an escape hatch — itself added as a prior audit fix (MEDIUM-3) to address the same class of issue.

The same structural problem applies to `ClosePending` and `LiquidatePending`: both transfer tokens to the admin and rely exclusively on the corresponding `finalize*` function to resolve the position. If finalization never occurs, the position is stuck and user funds are locked.

Recommendations

Add a `revert*` escape hatch for each pending state, following the `revertOpen` precedent:

- `revertSettle`: returns USDC (or re-credits the position) after a configurable timeout, callable by admin or position owner.
- `revertClose` / `revertLiquidate`: similar recovery mechanisms returning tokens to the vault and reverting state to the prior position state.

Each should include a timeout to prevent premature reversion while giving the backend reasonable time to complete the off-chain operation.

Status

Acknowledged. Deferred to a future V2 with partial-fill-aware reverse transitions.

[L-05] Incomplete rescueERC1155 guard allows extraction of active position tokens after full deleverage

Affected files

LeveragedPredictionVaultV1.sol#L1635

Description

The `rescueERC1155` function guards against rescuing active position tokens by checking `tokenTotalBorrowedUsdcUnits[tokenId] > 0`. This proxy variable tracks protocol-borrowed capital per tokenId, not whether active positions still hold tokens of that type.

```
function rescueERC1155(address token, uint256 tokenId,
uint256 amountUnits)
    external nonReentrant onlyGlobalAdmin
{
    if (token == conditionalTokens &&
tokenTotalBorrowedUsdcUnits[tokenId] > 0) {
        revert CannotRescueOperationalToken();
    }
    IERC1155(token).safeTransferFrom(address(this),
globalAdmin, tokenId, amountUnits, "");
}
```

After a position is fully deleveraged via force unwind (all borrowed capital repaid, `borrowedUsdcUnits` reduced to 0), `tokenTotalBorrowedUsdcUnits[tokenId]` reaches zero — even though the position still holds ERC1155 tokens in `positionTokenUnits` and remains in

`Opened` state. A `rescueERC1155` call for that tokenId succeeds, draining the vault's token balance. All subsequent exit paths (`closePosition`, `settlePosition`, `liquidatePosition`) revert permanently because the vault cannot transfer tokens it no longer holds. The user's accumulated `pendingRefundUsdcUnits` from prior unwind cycles are also permanently locked.

Recommendations

Track a separate `tokenTotalPositionUnits` counter that is incremented at `finalizeOpen` and decremented only at terminal finalization, and use it as the rescue guard:

```
if (token == conditionalTokens &&
tokenTotalPositionUnits[tokenId] > 0) {
    revert CannotRescueOperationalToken();
}
```

Status

Fixed. New `tokenTotalPositionUnits` counter replaces the borrowed-capital proxy in the rescue guard. Closes the related Krait pre-audit observation on the same surface.

[L-06] Force unwind unconditionally resets position state to Opened, discarding user's prior CloseRequested intent

Affected files

LeveragedPredictionVaultV1.sol#L1487

Description

When `finalizeForceUnwind` completes a non-terminal unwind, it unconditionally resets the position state to `Opened`:

```
// LeveragedPredictionVaultV1.sol:L1487-1488
positionStates[positionKey] =
    currentState == Types.PositionState.UnwindPending ?
Types.PositionState.Opened : currentState;
```

At finalization time, `currentState` is always `UnwindPending` (set by `forceUnwind`), so the ternary always evaluates to `Opened`. If the position was in `CloseRequested` before the unwind was initiated, that state is silently discarded — there is no mechanism to preserve or restore it through the unwind cycle.

The user must discover off-chain that their close request was dropped and re-submit `requestClose`. In a multi-unwind scenario, this can happen repeatedly — each unwind cycle erases the close intent, requiring the user to re-signal after every cycle with no on-chain notification.

Recommendations

Store the pre-unwind state before transitioning to `UnwindPending` and restore it after finalization. For example, add a `preUnwindState` field to the position struct:

```
// In forceUnwind:
position.preUnwindState = currentState; // save
CloseRequested or Opened

// In finalizeForceUnwind:
positionStates[positionKey] =
    currentState == Types.PositionState.UnwindPending ?
position.preUnwindState : currentState;
```

This ensures a user's close intent survives unwind cycles without requiring them to re-signal.

Status

Fixed. New `preUnwindState` field records pre-unwind state; non-terminal `finalizeForceUnwind` restores it.

[L-07] finalizeLiquidate leaves actualNotionalUsdcUnits stale post-liquidation, creating a latent fee-computation bug and inconsistent off-chain reads

Affected files

LeveragedPredictionVaultV1.sol#L1248-L1254

Description

`finalizeLiquidate` defensively writes `borrowedUsdcUnits`, `pendingLifetimeFeeUsdcUnits`, and `feeAccrualStartAt` after the liquidation completes, so that a delayed `finalizeForceUnwind` arriving from `Liquidated` state (the concurrent unwind+liquidate race) can correctly compute its remaining work. However, it does NOT update `actualNotionalUsdcUnits`. The stored notional retains its pre-liquidation value even though the position's effective post-liquidation notional is now `collateralUsdcUnits + borrowedUsdcUnits` (which is smaller after the residual borrow reduction at L1248).

```
// LeveragedPredictionVaultV1.sol:L1248-1254
position.borrowedUsdcUnits -=
dist.protocolCapitalReturnUsdcUnits;
position.pendingLifetimeFeeUsdcUnits =
collectedLifetimeFeeUsdcUnits <=
totalLifetimeFeesOwedUsdcUnits
    ? totalLifetimeFeesOwedUsdcUnits -
collectedLifetimeFeeUsdcUnits
    : 0;
position.feeAccrualStartAt = uint64(block.timestamp);
// Missing: position.actualNotionalUsdcUnits =
```

```
position.collateralUsdcUnits +
position.borrowedUsdcUnits;
```

The contract is currently safe on-chain only by accident: the only consumer that would use the stale notional for fee math (`finalizeForceUnwind` at L1407–L1410) short-circuits to `pendingLifetimeFeeUsdcUnits` via the `isTerminal` branch at L1402 before reaching the stale-read path. The safety property is not "the value is correct" — it isn't — but rather "we currently never look at it in the wrong state." Any future refactor of the `isTerminal` branch in `finalizeForceUnwind`, or any new on-chain consumer of `actualNotionalUsdcUnits` on a `Liquidated` position, would manifest the latent bug as incorrect fee accrual against the user.

The off-chain impact is more immediate. `actualNotionalUsdcUnits` is a `public` storage variable with an auto-generated getter, almost certainly consumed by partner frontends, indexers, dashboards, monitoring, and protocol-TVL attestation. Those consumers have no way to know the field silently goes stale the moment a position transitions to `Liquidated`, and would display or aggregate pre-liquidation notionals for liquidated positions — inflating per-position figures and any aggregates derived from them. The same staleness would also propagate into a future V1→V2 storage migration if the field is copied verbatim into the new schema.

Recommendations

Add one line in `finalizeLiquidate` after L1248 so the stored notional stays consistent with the position's post-liquidation state:

```
position.borrowedUsdcUnits -=
dist.protocolCapitalReturnUsdcUnits;
```

```
+ position.actualNotionalUsdcUnits =
position.collateralUsdcUnits +
position.borrowedUsdcUnits;
```

This eliminates the latent on-chain fee-computation bug, removes the implicit dependency on `finalizeForceUnwind`'s `isTerminal` branch, and ensures all on-chain and off-chain readers of `actualNotionalUsdcUnits` see a value that reflects the position's current state.

Status

Fixed. `_resyncDerivedFields` recomputes `actualNotionalUsdcUnits` and `leverageBps` after the borrowed decrement; identity locked in by `_assertPositionInvariants`.

7.3 Informational Severity Findings

[I-01] cancelDelaySeconds changes apply retroactively to existing positions

Description

`updateCancelDelay()` at L329 modifies the global `cancelDelaySeconds`. The cancel check at L787 reads this global value at cancel-time:

```
uint256 cancelAfter = uint256(position.createdAt) +
cancelDelaySeconds;
```

If a user creates a position when the delay is 10 minutes and the admin later changes it to 1 hour, that position's lock-up is retroactively extended. The position was created under one set of rules and cancelled under another.

This works in both directions — increasing the delay locks existing `Created` positions longer than expected, while decreasing it to 0 makes all `Created` positions immediately cancellable.

The impact is bounded by `MAX_CANCEL_DELAY_SECONDS` (1 hour) and the short window positions typically spend in `Created` state. However, this becomes more relevant if `cancelPosition` is made user-callable (per L-01), since the delay would then function as a user-facing guarantee that can be retroactively changed.

Recommendations

Snapshot the applicable delay per-position at creation time. Add a field to `PositionVault`:

```
uint32 appliedCancelDelaySeconds;
```

Set it in `storeNewPosition`, and read it in `cancelPosition` instead of the global:

```
uint256 cancelAfter = uint256(position.createdAt) +
uint256(position.appliedCancelDelaySeconds);
```

Status

Hardened. `updateCancelDelay` moved to `onlyTimelock` so retroactive changes traverse the timelock delay.

[I-02] No update path creates a chicken-and-egg risk with Timelock address being immutable

Description

`setTimelock()` at L254 uses `reinitializer(2)` and checks `timelock != address(0)`, making it a one-shot setter. Once set, there is no `updateTimelock` function. The timelock address is permanent for the life of V1.

The timelock gates three critical operations via `onlyTimelock`: `updateAdmins` (L276), `withdrawCapital` (L361), and `_authorizeUpgrade` (L1685). If the timelock contract becomes inoperable (contract bug, governance locked, keys lost), the vault is permanently frozen:

- No upgrades possible
- No admin address changes possible
- No capital withdrawal possible
- The globalAdmin retains pause/unpause, addCapital, updateLimits, and updateFeeAdmin — but can never recover the capital pool

Every other privileged address in the contract (globalAdmin, emergencyAdmin, appAdmin1/2, feeAdmin, feeReceiver) is changeable via `updateAdmins`. The timelock is the only permanently immutable address.

The timelock is likely an OpenZeppelin `TimelockController` with its own role management, so it is not entirely ungovernable. However, from the vault's

perspective, it is forever bound to one address with no on-chain migration path.

Recommendations

Add a timelock-gated `updateTimelock` function so the existing timelock can authorize its own replacement:

```
function updateTimelock(address newTimelock) external
nonReentrant onlyTimelock {
    if (newTimelock == address(0)) revert ZeroAddress();
    address oldTimelock = timelock;
    timelock = newTimelock;
    emit Events.TimelockUpdated(oldTimelock, newTimelock,
block.timestamp);
}
```

Alternatively, if the team considers this an accepted architectural constraint, document it explicitly so future operators are aware of the permanence.

Status

Fixed. `updateTimelock` added, gated `onlyTimelock`; preserves the on-chain delay model for any rotation.

[I-03] Coarse-grained emergency mode forces all position exits through liquidation path, causing users to pay unwarranted liquidation fees

Affected files

LeveragedPredictionVaultV1.sol#L850 (closePosition — blocked by whenNotEmergencyOnly)
LeveragedPredictionVaultV1.sol#L882 (finalizeClose — blocked by whenNotEmergencyOnly)
LeveragedPredictionVaultV1.sol#L825 (requestClose — blocked by whenNotEmergencyOnly)

Description

During emergency mode, `closePosition`, `finalizeClose`, and `requestClose` are all blocked by `whenNotEmergencyOnly`. However, `liquidatePosition` and `finalizeLiquidate` remain available. This means the only exit path for positions during an emergency (besides settlement, which requires market resolution) is liquidation — which charges `liquidationFeeBps` on the full notional (up to 20% max). The normal close path charges no liquidation fee.

The emergency admin (hot wallet) can enable emergency mode, but only the global admin (cold multisig) can disable it. If the multisig is slow to coordinate, the emergency window could extend for hours or days. Every position exited during that window pays a liquidation fee that would not apply under normal close operations, even if the position is perfectly healthy.

Recommendations

Consider a more granular emergency pause architecture — for example, a separate modifier for withdrawal/close operations distinct from `whenNotEmergencyOnly`. This would allow the admin to selectively block the close path only when the emergency specifically concerns close logic, while keeping it available when the emergency relates to other operations (e.g., position opening, market interactions). The existing three-tier pause structure (`newPositionsPaused` → `userActionsPaused` → `serverHalted`) plus `emergencyOnlyMode` could be extended with a withdrawal-specific tier.

Status

Acknowledged. Emergency mode is the "risk-reduction only" tier by design; `settlePosition` and `forceUnwind` remain available with no liquidation fee.

[I-04] Missing post-unwind leverage validation allows ineffective force unwinds to complete silently

Description

In `finalizeForceUnwind`, after the waterfall runs the actual new leverage is computed at L1464:

```
computedLeverageBps = uint32((computedNotionalUsdcUnits * Types.BPS_BASE) / collateralUsdcUnits);
```

However, there is no check that `computedLeverageBps` is at or below the `targetLeverageBps` specified in `forceUnwind`. The result is stored and the function completes regardless of whether the unwind achieved its deleveraging goal.

This is possible because `tokenUnitsToSell` and `targetLeverageBps` are validated independently in `forceUnwind` — the contract never checks that they are consistent with each other. An admin could pass `targetLeverage = 2x` but sell too few tokens, producing a waterfall that barely returns any capital. The position returns to `Opened` at nearly the same leverage as before, and the contract emits events as if the unwind succeeded.

Other finalize functions in the contract validate computed results against expectations (e.g., `isWithinBounds` sanity checks on proceeds vs notional). The absence of a similar post-hoc check here is an inconsistency in the contract's defense-in-depth posture.

Recommendations

Add a post-unwind sanity check after the leverage is computed at L1464:

```
require(computedLeverageBps <= storedTargetLeverageBps +
TOLERANCE, "Unwind did not achieve target leverage");
```

Additionally, as a backend verification item: validate that the computed `tokenUnitsToSell` is consistent with `targetLeverageBps` before submitting the `forceUnwind` transaction.

Status

Mitigated off-chain. Backend `VaultEventAlertService.checkFinalizeEvent` raises a Sentry alert on leverage drift beyond tolerance.

[I-05] Invariant suite handler uses legacy EIP-191 signatures, silently reverts every position creation, yields zero security signal

Description

The Foundry invariant suite at `evm_protocol/test/invariant/VaultInvariant.t.sol` declares ten `invariant_*` properties exercised through `PositionHandler.sol`. The suite reports green on every campaign run, but produces no security signal because its handler builds signatures with legacy EIP-191 packed format while the contract requires EIP-712 typed data:

```
// PositionHandler._signCreatePosition (lines 470-491) —
legacy EIP-191 packed:
bytes32 messageHash = keccak256(abi.encodePacked(
    "CREATE_POSITION", positionSeed, user, marketId, ...
));
bytes32 ethSignedHash = keccak256(abi.encodePacked(
    "\x19Ethereum Signed Message:\n32", messageHash
));

// SignatureLib.verifyCreatePositionSignature requires
EIP-712:
bytes32 structHash =
keccak256(abi.encode(CREATE_POSITION_TYPEHASH, ...));
bytes32 digest = keccak256(abi.encodePacked(
    "\x19\x01", domainSeparator, structHash
));
```

Different digests → recovered signer ≠ `appAdmin1` / `appAdmin2` → `InvalidSignature` revert. The revert is swallowed by the `try/catch` block in

`PositionHandler.createPosition` (line 130), so the fuzzer treats the call as non-reverting and continues. `createdPositionKeys` stays empty across every run, and every position-iterating invariant is vacuously satisfied. Only invariants that check immutable state (`AdminAddressesNeverZero`, `ExternalContractsNeverZero`, `VersionIsOne`) or pause-toggle ghost state carry real signal.

The correct EIP-712 signing helpers (`_signCreatePositionWithKey`, `_signFinalizeSettleWithKey`) already exist in `test/helpers/SignatureHelpers.sol` and are used by every passing unit test. The handler simply does not inherit them.

Recommendations

Have `PositionHandler` inherit `SignatureHelpers` (which extends `Test`) instead of importing `Test` directly, override `_getVaultAddress()` to return `address(vault)`, delete the local packed-signature helpers, and route signing through the inherited EIP-712 functions. Add a campaign-level liveness assertion (`afterInvariant`: `positionCount > 0`) so a regression that re-introduces silent signature failures fails a hard check instead of being absorbed by the catch block.

Status

Fixed. Invariant handler now inherits `SignatureHelpers` and signs via the EIP-712 helpers; campaign-level liveness assertion guards against silent regression.

[I-06] EIP-191 personal-sign instead of EIP-712 typed data reduces wallet transparency

Description

At the time of the Krait pre-audit (commit `5eae950`, 2026-04-07), `SignatureLib` used EIP-191 (`toEthSignedMessageHash`) with custom packed encoding and inline assembly. While functionally secure (`chainid()` and `address(this)` were already included for replay protection), EIP-191 displays an opaque hash in user wallets rather than human-readable structured data, and prevents users from verifying what they are signing.

Recommendations

Migrate signature verification to EIP-712 typed data. Both `CreatePosition` and `FinalizeSettle` payloads have well-defined structured fields suitable for `EIP712Domain` + struct-hash encoding.

Status

Fixed. `SignatureLib` migrated to EIP-712 typed data before the manual audit began.

[I-07] No on-chain cumulative bad-debt tracking

Description

`badDebtUsdcUnits` is computed in `CloseLib.calculateCloseDistribution` and emitted in `PositionClosed` / `PositionSettled` / `PositionLiquidated` / `PositionForceUnwound` events, but never accumulated in a storage variable. Determining cumulative protocol bad-debt exposure for governance, capital-pool sizing, or solvency assessment requires off-chain event indexing rather than a single on-chain read.

Recommendations

Add a `totalBadDebtUsdcUnits` storage variable, incremented in `finalizeClose`, `finalizeSettle`, `finalizeLiquidate`, and the terminal branch of `finalizeForceUnwind`, to enable on-chain solvency queries.

Status

Acknowledged. On-chain bad-debt accumulator deferred on contract-size grounds; off-chain indexing of finalize events covers the same data.

[I-08] Partial fill can result in leverage below MIN_LEVERAGE_BPS

Description

When `finalizeOpen` processes a partial fill, leverage is recomputed from `actualNotionalUsdcUnits`:

```
position.leverageBps = uint32(
    (actualNotionalUsdcUnits * Types.BPS_BASE) /
position.collateralUsdcUnits
);
```

The guard at line 643 enforces `actualNotionalUsdcUnits >= collateralUsdcUnits` (≥1x leverage) but does not enforce `MIN_LEVERAGE_BPS` (1.5x). A position created at 2× leverage with a 60% partial fill would resolve at 1.2× — below the protocol's stated minimum.

All downstream functions (close, settle, liquidate, force unwind) handle sub-1.5× leverage correctly. The alternative — rejecting partial fills below `MIN_LEVERAGE_BPS` — would force `revertOpen`, returning the user to the start of the lifecycle for what is otherwise a valid position. The cost-benefit favors accepting the sub-minimum leverage.

Recommendations

Document the partial-fill leverage relaxation in `finalizeOpen`'s NatSpec so the inconsistency between the `MIN_LEVERAGE_BPS` constant (applied at `createPosition`) and the post-fill realized leverage is visible to future maintainers.

Status

Acknowledged as documented behavior. No code change.