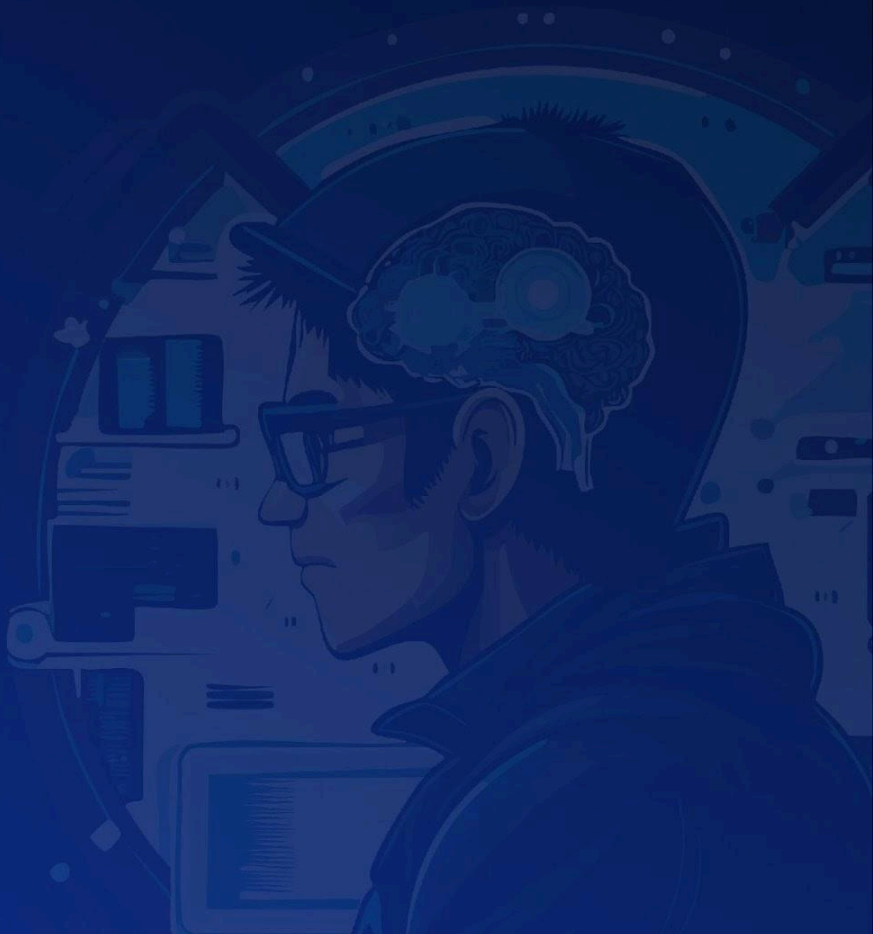
Anchored Vault

Security Review

Disclaimer

Security Review

Anchored Vault



Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

Anchored Vault

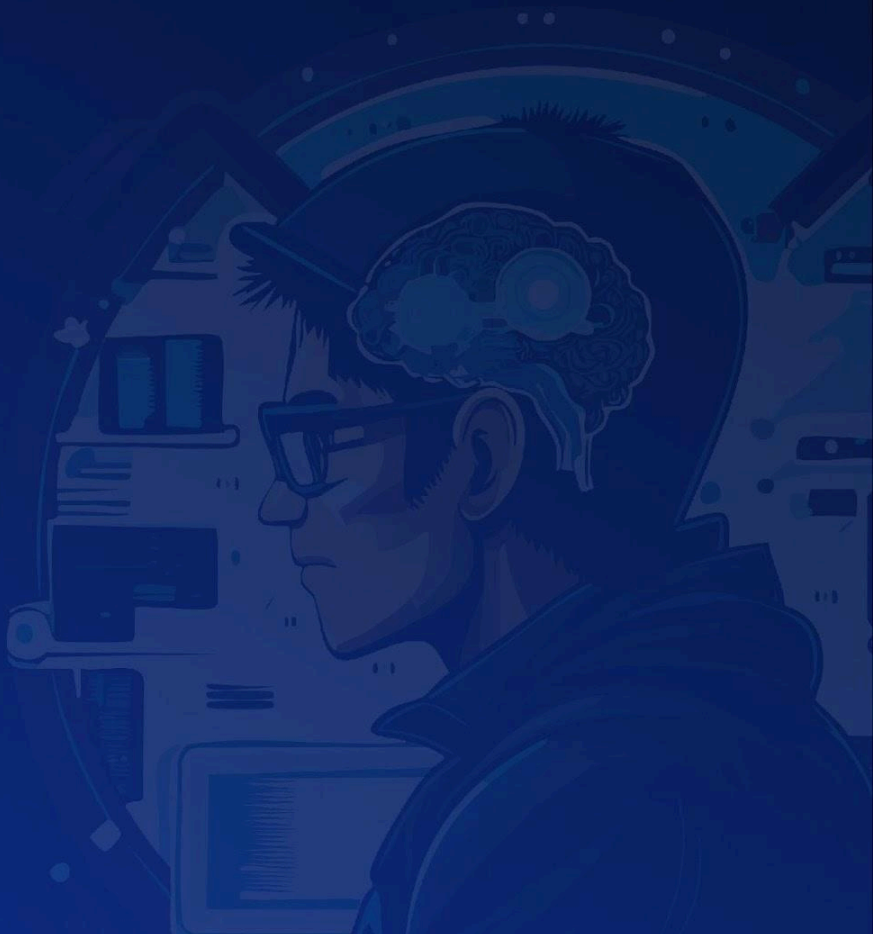
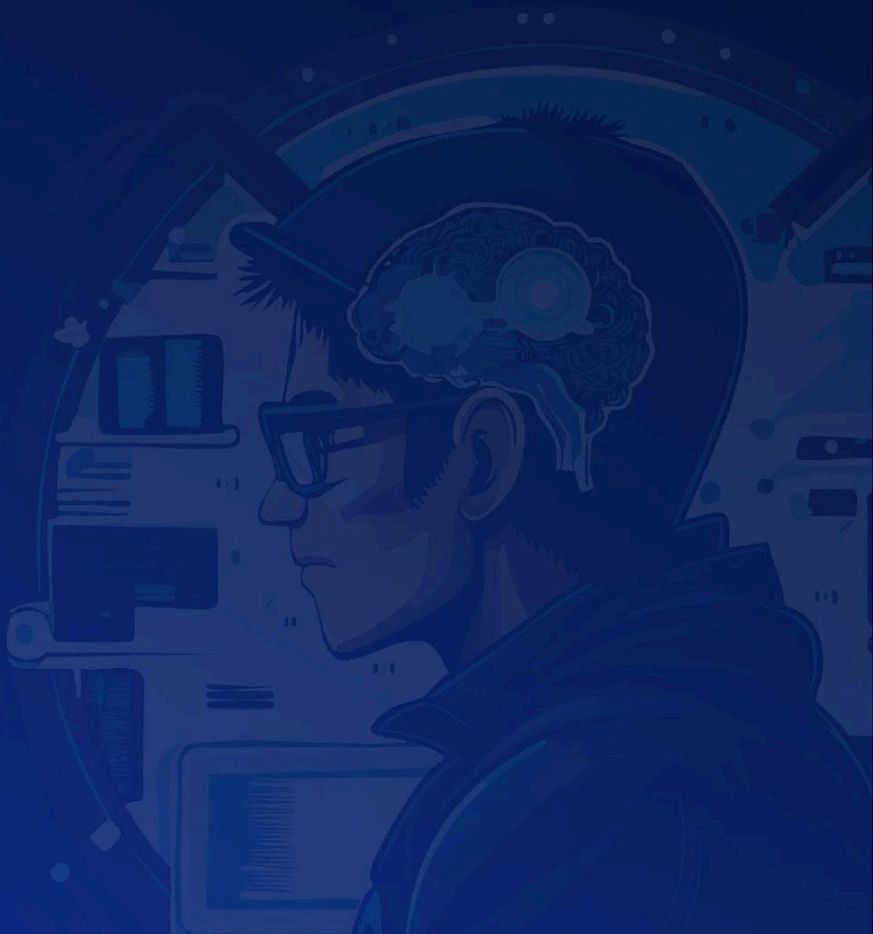


Table of Contents

Summary

Security Review

Anchored Vault



Summary

Three Sigma audited Anchored in a 2 person week engagement. The audit was conducted from 14/05/2026 to 21/05/2026.

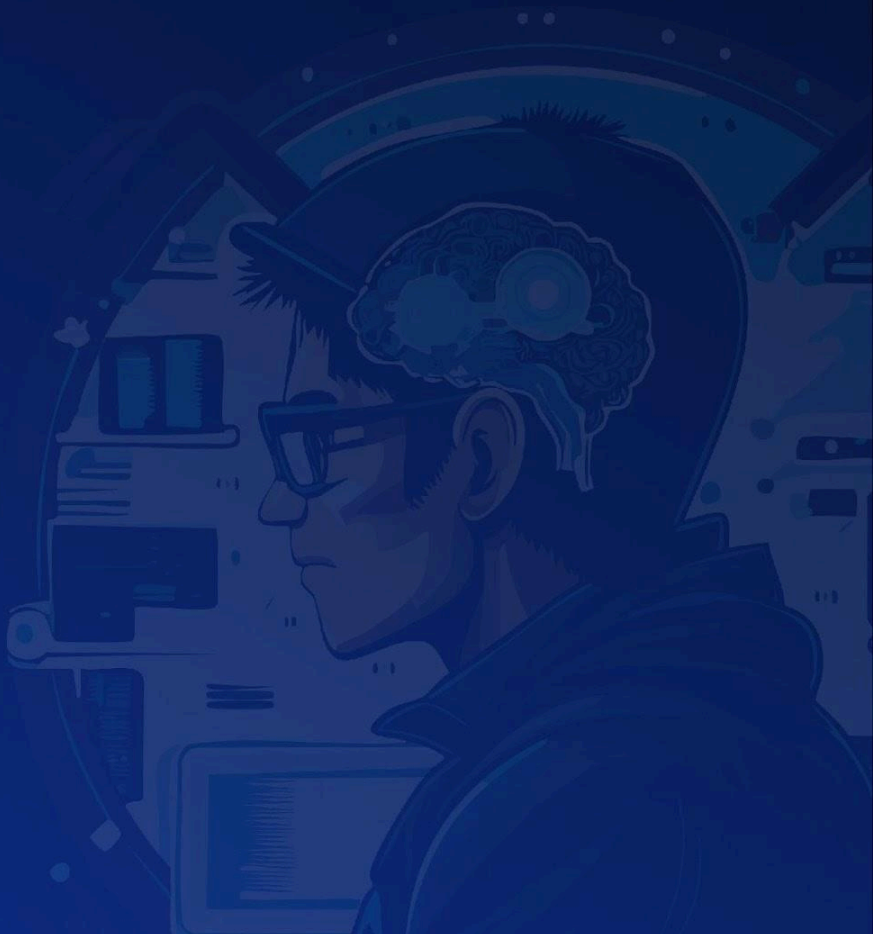
Protocol Description

Anchored Fund is a permissioned, on-chain tokenized fund protocol. Investors subscribe with whitelisted ERC20 assets and receive per-fund ERC20 share tokens; they later redeem those shares back into assets after a redemption notice period. Share pricing is driven by an off-chain NAV that an operator pushes on-chain as append-only records, which are then bound to individual requests at settlement time. Every investor-facing and settlement action is gated by a global compliance module enforcing KYC/approval together with denylist and sanctions screening.

Scope

Security Review

Anchored Vault



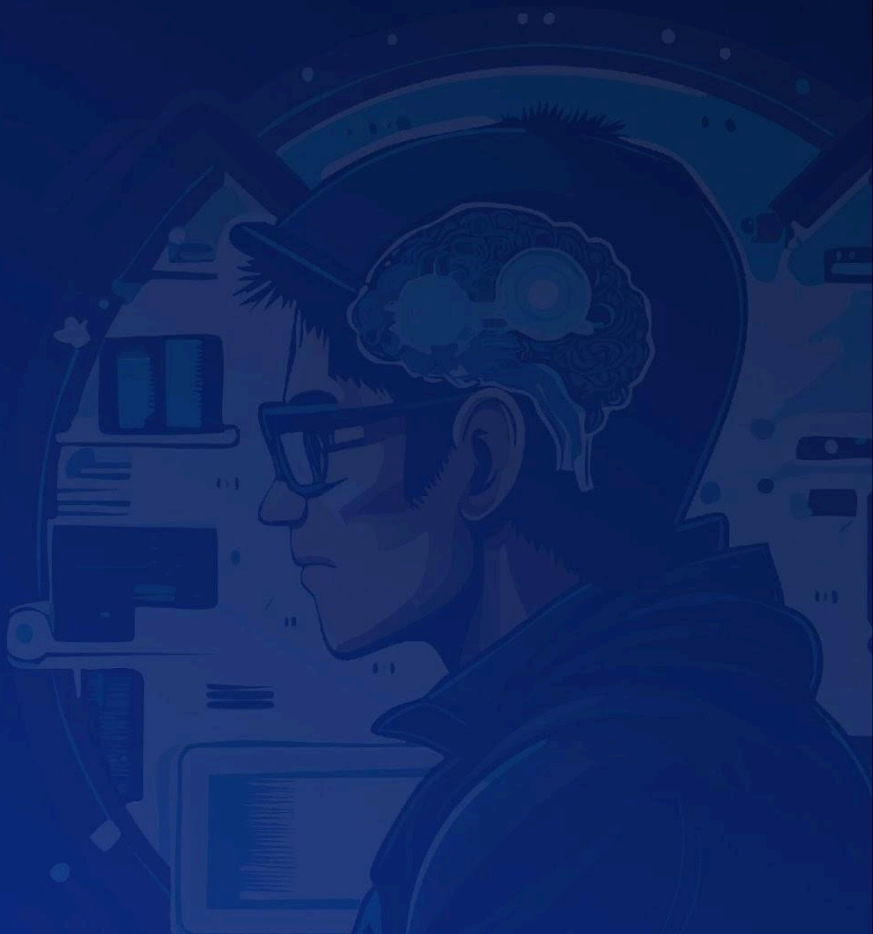
Scope

Filepath	nSLOC
contracts/fund/AncFund.sol	399
contracts/fund/AncFundController.sol	206
contracts/fund/AncFundToken.sol	93
contracts/fund/AncFundSettler.sol	82
contracts/fund/AncFundNavRecord.sol	62
contracts/libs/fund/FundTypes.sol	56
contracts/fund/AncFundRouter.sol	53
contracts/libs/fund/FundConfigLib.sol	47
contracts/libs/fund/AssetConfigLib.sol	31
contracts/libs/fund/TimeWindowLib.sol	27
contracts/libs/fund/RequestIdLib.sol	14
Total	**1070**

Methodology

Security Review

Anchored Vault



To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

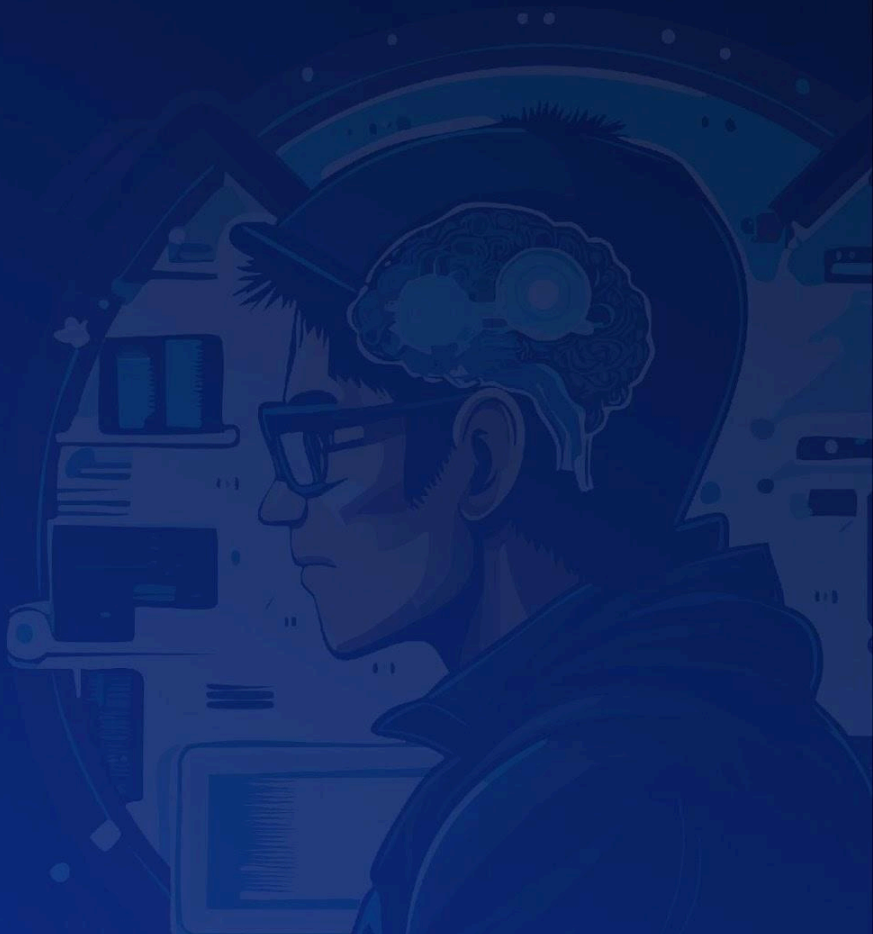
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Anchored Vault



Project Dashboard

Application Summary

Name	Anchored Fund
Repository	https://github.com/AnchoredLabs/anchored-contracts
Commit	8b1b89c
Language	Solidity
Platform	EVM

Engagement Summary

Timeline	14/05/2026 to 21/05/2026
№ of Auditors	2
Review Time	2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	2	1	1
Low	5	4	1
None	4	4	0

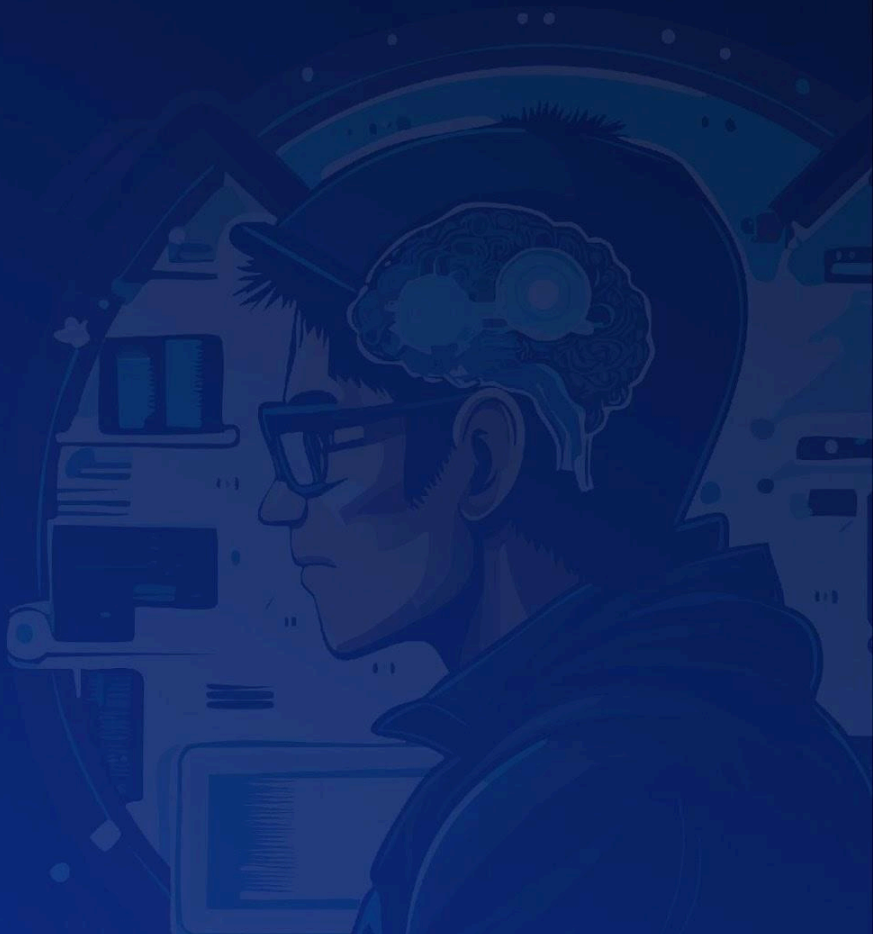
Category Breakdown

Suggestion	4
Documentation	0
Bug	6
Optimization	0
Good Code Practices	0

Risk Section

Security Review

Anchored Vault



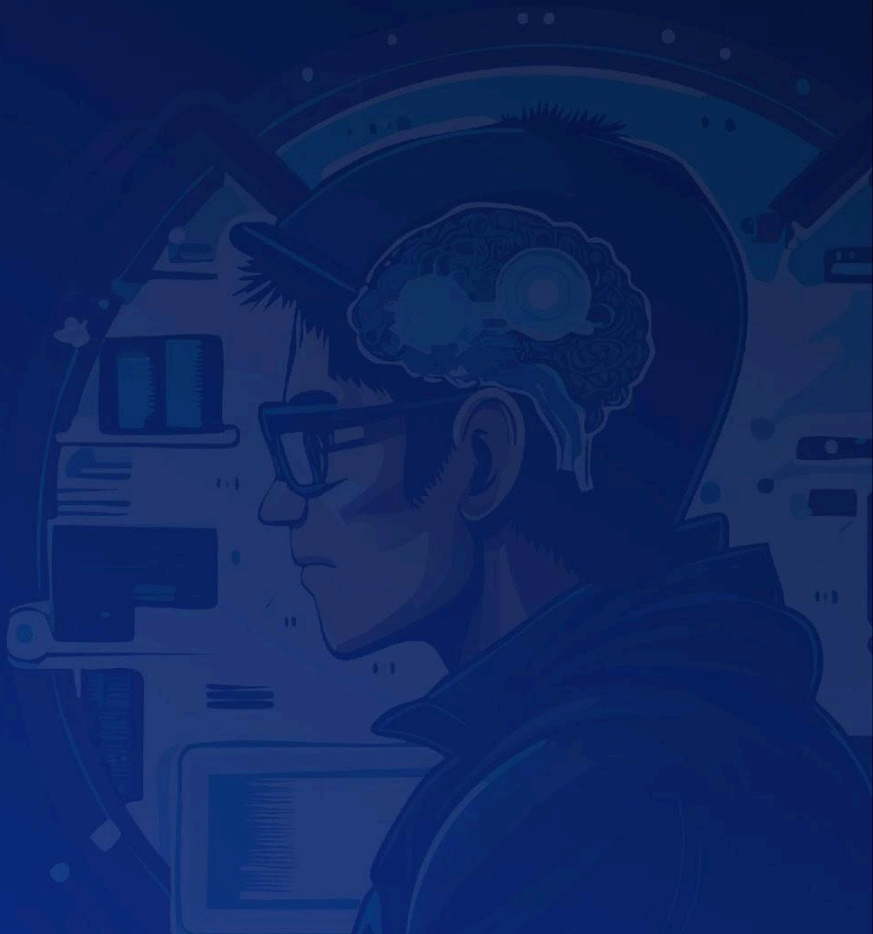
Risk Section

- **Redeemer-chosen payout asset may create rebalancing costs**: In AncFund::submitRedemptionRequest, redeemers select their payoutAsset, and the contract has no on-chain view of the fund's actual asset composition (assets leave the contract at collectFund), so it cannot weigh a request against available reserves. If a redeemer happens to choose an asset the fund is light on, the settler may need to source it on-market once the cancel window closes, and any slippage on that purchase is effectively shared by remaining holders. In normal conditions this is a minor rebalancing cost, but a redeemer could in principle pick the fund's scarcest asset to shift that cost onto others.
- **Upgradability Risk**: Administrators have the capability to update contract logic at any moment due to the project's upgradable design. Contract upgradability, while beneficial, also poses the risk of harmful modifications if administrative privileges or upgrade processes are compromised.

Findings

Security Review

Anchored Vault



Findings

3S-Fund-M01

AncFund::setPriceId does not validate that the nav record's asset matches the subscription's asset

Id	3S-Fund-M01
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Acknowledged

Description

AncFund::setPriceId binds a NAV record (**priceId**) to a pending subscription or redemption request. A fund can accept multiple asset types, and each subscription's **requestId** encodes the asset index via **RequestIdLib.pack**. However, **NavRecord** stores only **fundId** and has no **assetIndex** field. **AncFundNavRecord::recordNav** accepts a **fundId** but no asset identifier, so every record produced for a given fund is indistinguishable by asset.

Both **AncFund::_calculateSubscriptionSharesFromPriceId** and **AncFund::_calculateRedemptionAssetsFromPriceId** read **nav** from the chosen record and use it as the conversion ratio between shares and the request's asset. The subscription path computes shares as **scaledAssets * navScale / nav**, and the redemption path computes assets as **shares * nav / navScale**. The ratio **nav / navScale** is asset units per share, which is asset-specific because each accepted asset has its own market price. Correct settlement therefore requires a distinct NAV record per asset.

When the settler calls **setPriceId**, the function checks that **pricedFundId == fundId** but never verifies that the nav record corresponds to the correct asset. If a fund accepts USDT (index 0) and USDC (index 1), a USDT subscription can be assigned a USDC nav record. Settlement prices USDT shares using the USDC NAV. The investor receives the wrong share count.

Steps to Reproduce

1. Deploy a fund with two accepted assets: USDT at index 0 and USDC at index 1.

2. Alice calls **subscribe** with USDT; her **requestId** encodes **assetIndex = 0**.

3. The nav pusher calls **recordNav** twice for the same **fundId**, producing record **A** (intended for USDT) and record **B** (intended for USDC).

4. The settler calls **setPriceId** with Alice's **requestId** and the USDC record **B** as **priceId**.

5. The call succeeds because only **fundId** is validated. Alice's subscription is now priced at the USDC NAV.

6. **settleSubscription** mints shares to Alice using the wrong price.

Recommendation

Add **assetIndex** to **NavRecord** and pass it through **recordNav** and **getNavRecord**. In **setPriceId**, unpack the asset index from **requestId** and verify it matches the asset index stored in the nav record.

```
// contracts/libs/fund/FundTypes.sol
 struct NavRecord {
     uint16 fundId;            // slot 0, bytes 0..1
-    uint32 sourceUpdatedAt;    // slot 0, bytes 2..5
-    uint32 recordedAt;         // slot 0, bytes 6..9
-    uint128 nav;               // slot 0, bytes 10..25
+    uint16 assetIndex;         // slot 0, bytes 2..3
+    uint32 sourceUpdatedAt;    // slot 0, bytes 4..7
+    uint32 recordedAt;         // slot 0, bytes 8..11
+    uint128 nav;               // slot 0, bytes 12..27
 }
```

```
// contracts/fund/AncFundNavRecord.sol
-function recordNav(uint16 fundId, uint128 nav, uint32 sourceUpdatedAt)
+function recordNav(uint16 fundId, uint16 assetIndex, uint128 nav, uint32
sourceUpdatedAt)
    external
    override
    nonReentrant
    returns (uint64 navRecordId)
 {
    _requireNavPusher();
    if (fundId == 0) revert ZeroFundId();
    if (nav == 0) revert ZeroNav();
    if (sourceUpdatedAt == 0) revert ZeroSourceUpdatedAt();
```

```
    navRecordId = ++navRecordCounter;
    uint32 recordedAt = uint32(block.timestamp);
    NavRecord memory record =
-       NavRecord({ fundId: fundId, sourceUpdatedAt: sourceUpdatedAt, recordedAt:
recordedAt, nav: nav });
+       NavRecord({ fundId: fundId, assetIndex: assetIndex, sourceUpdatedAt:
sourceUpdatedAt, recordedAt: recordedAt, nav: nav });
    _navRecords[navRecordId] = record;
    emit NavRecordCreated(navRecordId, record);
 }
 function getNavRecord(uint64 navRecordId)
    external view override
-   returns (uint16 fundId, uint128 nav, uint32 sourceUpdatedAt, uint32 recordedAt)
+    returns (uint16 fundId, uint16 assetIndex, uint128 nav, uint32 sourceUpdatedAt,
uint32 recordedAt)
 {
    NavRecord storage record = _navRecords[navRecordId];
    if (record.recordedAt == 0) revert NavRecordNotFound();
-   return (record.fundId, record.nav, record.sourceUpdatedAt, record.recordedAt);
+    return (record.fundId, record.assetIndex, record.nav, record.sourceUpdatedAt,
record.recordedAt);
 }



// contracts/fund/AncFund.sol
-    (uint16 pricedFundId,,,) =
IAncFundNavRecord(NAV_RECORD).getNavRecord(priceId);
-    if (pricedFundId != fundId) revert InvalidPriceId();
-    (, uint16 packedFundId,,) = requestId.unpack();
-    if (packedFundId != fundId) revert RequestNotFound();
+    (uint16 pricedFundId, uint16 navAssetIndex,,,) =
IAncFundNavRecord(NAV_RECORD).getNavRecord(priceId);
+    if (pricedFundId != fundId) revert InvalidPriceId();
+    (, uint16 packedFundId, uint16 requestAssetIndex,) = requestId.unpack();
+    if (packedFundId != fundId) revert RequestNotFound();
+    if (navAssetIndex != requestAssetIndex) revert InvalidPriceId();
```

3S-Fund-M02

Missing NAV staleness validation when assigning a **priceId**

Id	3S-Fund-M02
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #8a026b0.

Description

AncFund::setPriceId is the gate where the settler attaches an off-chain NAV to a pending subscription or redemption request. NAV records live in **AncFundNavRecord**, where the **NAV_PUSH_ROLE** calls **recordNav(fundId, nav, sourceUpdatedAt)** to publish each new value. The registry stores four fields per record: the target **fundId**, the published **nav**, the **sourceUpdatedAt** timestamp that reports when the off-chain pipeline measured the NAV, and a **recordedAt** block timestamp captured at insertion. Settlement reads from the same registry. **AncFund::_calculateSubscriptionSharesFromPriceId** and **AncFund::_calculateRedemptionAssetsFromPriceId** both pull **nav** via **getNav(priceId)** and use it to mint shares for subscribers or pay assets to redeemers.

When **setPriceId** reads the record via **getNavRecord(priceId)**, it destructures the return tuple but takes only **pricedFundId**. The code discards the other three fields through blank identifiers. The single check it enforces is **pricedFundId == fundId**. The function never compares **sourceUpdatedAt** against **block.timestamp**, never compares **recordedAt**, and never bounds how old a record may be. A **priceId** whose underlying NAV was measured hours, days, or weeks before the call satisfies the check as long as the fund identifier matches. The settler also chooses the **priceId** argument freely; nothing forces the settler to attach the most recent record, so the settler can pick an outdated **priceId** deliberately or by accident.

The request lifecycle is one-way. **setPriceId** requires the request to be **Funded** (subscription) or **Pending** (redemption), then promotes it to **PriceSet**. **markSettleable** requires **PriceSet** and promotes to **Settleable**. **settleSubscription** and **settleRedemption** require **Settleable** and then delete the request. No transition routes back through **setPriceId**, so the recorded **priceId** is the final input to settlement and the **setPriceId** gate is the only point at which the contract can refuse a stale NAV.

A stale NAV produces directional value transfer. When the recorded NAV is higher than the current off-chain price, every redemption settled against it overpays the redeemer and

drains the fund's reserves. When the recorded NAV is lower than the current off-chain price, every subscription settled against it over-mints shares and dilutes the existing holders. The exposure scales with the gap between **sourceUpdatedAt** and **block.timestamp** at the moment of **setPriceId**. An attacker who controls timing, for example by waiting for a publishing outage or front-running a planned NAV refresh, selects the worst record still present in the registry.

Recommendation

Add a configurable maximum staleness window. Inside **setPriceId**, read **sourceUpdatedAt** from **getNavRecord** and revert when the NAV is older than the window.

Note: This control bounds NAV age at the moment the settler attaches the **priceId**. The one-way state machine offers no second checkpoint, so a NAV that is fresh at **setPriceId** may still go stale before **settleSubscription** or **settleRedemption** executes. Closing that residual gap cleanly would require restructuring the lifecycle to allow re-pricing after **PriceSet**. Until then, operators must keep the interval between **setPriceId** and settlement short enough that the staleness bound chosen here remains meaningful at settlement time.

```
// contracts/fund/AncFund.sol
+    uint32 public navMaxStaleness;
+
+    error StaleNavRecord();
+
+    function setNavMaxStaleness(uint32 newMaxStaleness) external
onlyRole(DEFAULT_ADMIN_ROLE) {
+        navMaxStaleness = newMaxStaleness;
+    }
     function setPriceId(...) external nonReentrant {
         ...
-        (uint16 pricedFundId,,,) =
IAncFundNavRecord(NAV_RECORD).getNavRecord(priceId);
+        (uint16 pricedFundId,, uint32 sourceUpdatedAt,) =
IAncFundNavRecord(NAV_RECORD).getNavRecord(priceId);
         if (pricedFundId != fundId) revert InvalidPriceId();
+        if (block.timestamp > uint256(sourceUpdatedAt) + navMaxStaleness) revert
StaleNavRecord();
         ...
     }
```

3S-Fund-L01

Upgradeable fund contracts inherit non-upgradeable OpenZeppelin parents, exposing proxies to storage layout collisions

Id	3S-Fund-L01
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Acknowledged

Description

The protocol deploys **AncFundToken**, **AncFundSettler**, **AncFundNavRecord**, **AncFundController**, and **AncFund** behind proxies and intends to upgrade their implementations over time. Each contract inherits from the non-upgradeable OpenZeppelin parents (**ERC20**, **AccessControlEnumerable**, **ReentrancyGuardTransient**) instead of the corresponding ***Upgradeable** variants.

Non-upgradeable OZ parents declare their persistent state directly in the inheritance-flattened layout, so every balance, allowance, and role-member mapping occupies a fixed slot determined by the order of parents and child variables. The upgradeable variants instead place that state inside an ERC-7201 namespaced struct at a hashed slot, fully decoupled from sibling and child layout. The current design therefore breaks under two routine upgrade workflows:

1. A future OpenZeppelin release that adds, reorders, or resizes a state variable in any inherited non-upgradeable contract (for example a new field added to **ERC20** or **AccessControl**) shifts every subsequent slot in the layout. Once the protocol upgrades the implementation to one compiled against the newer OZ version, the proxy reads existing balances, allowances, roles, NAV records, and config from the wrong slots. Every live fund's accounting becomes unrecoverable.

2. The protocol cannot insert a new parent contract into the inheritance chain of any of these implementations in a future version. Adding a parent pushes the existing parents' variables down, so a new **AncFundToken** implementation reads **ERC20** balances out of slots that previously held unrelated state.

The upgradeable variants (**ERC20Upgradeable**, **AccessControlEnumerableUpgradeable**, **ReentrancyGuardTransientUpgradeable**)

exist specifically to make both workflows safe by isolating parent state into deterministic namespaces.

Recommendation

Replace every non-upgradeable OZ parent in the five proxy-backed contracts with its ***Upgradeable** counterpart, and invoke the corresponding **__*_init** / **__*_init_unchained** calls from each **initialize** function. The change is shown below for **AncFundToken**; the same swap applies to **AncFundSettler**, **AncFundNavRecord**, **AncFundController**, and **AncFund**.

```
// contracts/fund/AncFundToken.sol
-import { AccessControlEnumerable } from
"@openzeppelin/contracts/access/extensions/AccessControlEnumerable.sol";
-import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
+import { AccessControlEnumerableUpgradeable } from
"@openzeppelin/contracts-upgradeable/access/extensions/AccessControlEnumerabl
eUpgradeable.sol";
+import { ERC20Upgradeable } from
"@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
-contract AncFundToken is IAncFundToken, Initializable, AccessControlEnumerable,
ERC20 {
+contract AncFundToken is IAncFundToken, Initializable,
AccessControlEnumerableUpgradeable, ERC20Upgradeable {
-    constructor(address compliance_, address fundController_) ERC20("", "") {
+    constructor(address compliance_, address fundController_) {
        if (compliance_ == address(0) || fundController_ == address(0)) revert
ZeroAddress();
        COMPLIANCE = compliance_;
        CONTROLLER = fundController_;
        _disableInitializers();
    }
    function initialize(string calldata tokenName_, string calldata tokenSymbol_)
external initializer {
+        __ERC20_init_unchained("", "");
+        __AccessControlEnumerable_init_unchained();
        _grantRole(DEFAULT_ADMIN_ROLE, CONTROLLER);
        _name = tokenName_;
        _symbol = tokenSymbol_;
    }
```

3S-Fund-L02

Compliance re-check on subscription settlement and redemption cancel paths strands investor assets when admin flips compliance mid-flow

Id	3S-Fund-L02
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #f0096dc.

Description

AncFundRouter::requestSubscription and **AncFundRouter::requestRedemption** already gate entry to the subscription and redemption flows on **_checkCompliance**, which calls **IAncCompliance.requireCompliant(investor)**. Once an investor passes that gate, gross asset is pulled into **FUND** for a subscription, or shares are locked under **lockedShares[fundId][investor]** for a redemption. The investor's position is then committed to protocol custody until the corresponding settle or cancel path closes the request.

The protocol re-evaluates investor compliance at two later points where the investor is unwinding or completing an already-committed position. The first is **AncFund::_settleSubscription**, which calls **_requireInvestorCompliant(investor)** immediately before minting shares. The second is **AncFundRouter::cancelRedemption**, which calls **_checkCompliance()** on the caller before delegating to **AncFund::cancelRedemption**. If **IAncCompliance** is mutated between submission and these exit points (revocation of approval, denylist addition, or any other compliance state change), the affected investor can no longer move forward through settlement nor unwind through cancel.

On the subscription side, this is especially damaging because the settler's **collectFund** (**AncFund::collectFund**) forwards the net asset to the off-protocol **outgoing** sink before **_settleSubscription** is invoked. By the time **_requireInvestorCompliant** reverts, the cash has already left **FUND** custody; no investor-callable refund path exists, so the cash is stranded at an off-protocol address pending out-of-band admin action. On the redemption side, the shares remain locked in **FUND** via **lockedShares[fundId][investor]**, and the investor is denied the cancel path that would otherwise return them; recovery depends on

the admin re-approving the investor before **cancelDeadlineAt**, after which **_settleRedemption** still rechecks compliance at **AncFund.sol:418**.

The shared root cause is that compliance is treated as a continuously-evaluated condition on exit paths rather than as an entitlement that is snapshotted at request submission. An investor whose status legitimately changes mid-flow should still be able to unwind a position that the protocol already accepted; instead, the unwind path is exactly where the re-check causes harm.

Recommendation

Treat the compliance check on **requestSubscription** and **requestRedemption** as the single entitlement gate for the lifetime of the request. Remove **_requireInvestorCompliant(investor)** from **AncFund::_settleSubscription** so a submitted subscription always settles into shares once the rest of the pipeline succeeds, and drop the **_checkCompliance()** call inside **AncFundRouter::cancelRedemption** so an investor can always retrieve their own locked shares before **cancelDeadlineAt**. Compliance enforcement on the entry side already prevents non-compliant principals from creating new exposure, while the exit-side re-checks only weaponize compliance changes against investors who are already inside the protocol.

```
    function _settleSubscription(uint16 fundId, bytes32 requestId) internal returns
(uint256 sharesMinted) {
        ...
        FundConfig memory fund =
IAncFundController(CONTROLLER).fundConfigs(fundId);
        if (!fund.isConfigured()) revert FundNotFound();
-       _requireInvestorCompliant(investor);
        IAncFundToken(fund.shareToken).mint(investor, sharesMinted);



    function cancelRedemption(uint16 fundId, bytes32 requestId) external
nonReentrant {
-       address investor = _checkCompliance();
-
-       IAncFund(FUND).cancelRedemption(fundId, investor, requestId);
+       IAncFund(FUND).cancelRedemption(fundId, _msgSender(), requestId);
    }
```

3S-Fund-L03

AncFundController exposes no forwarders for **AncFundToken::setName**, **AncFundToken::setSymbol**, or **AncFundToken::setTransferPaused**

Id	3S-Fund-L03
Classification	Low
Impact	Low
Likelihood	High
Category	
Status	Addressed in #47eb9b5.

Description

The **onlyFundController** modifier guards **AncFundToken::setName**, **AncFundToken::setSymbol**, and **AncFundToken::setTransferPaused**. It rejects every caller except the immutable **CONTROLLER** address. The constructor wires that address to the **AncFundController** deployment, so only the controller contract can reach these setters. **AncFundController** defines no function that calls any of the three.

A share token's name and symbol cannot change after **AncFundController::launchFund** writes them. No caller can flip the **transferPaused** switch on for incident response or back off afterwards. Any operational scenario that depends on halting transfers, recovering from a compromise, or rebranding the share token has no on-chain path.

Recommendation

Add three forwarders on **AncFundController**, each gated by **FUND_CONTROLLER_ROLE**, that call the matching setter on the fund's **shareToken**.

```
// contracts/fund/AncFundController.sol
+    function setFundTokenName(uint16 fundId, string calldata tokenName) external
nonReentrant {
+        _requireFundControllerRole();
+        IAncFundToken(_requireFund(fundId).shareToken).setName(tokenName);
+    }
+
+    function setFundTokenSymbol(uint16 fundId, string calldata tokenSymbol)
```

```
external nonReentrant {
+       _requireFundControllerRole();
+       IAncFundToken(_requireFund(fundId).shareToken).setSymbol(tokenSymbol);
+   }
+
+   function setFundTokenTransferPaused(uint16 fundId, bool paused) external
nonReentrant {
+       _requireFundControllerRole();
+
IAncFundToken(_requireFund(fundId).shareToken).setTransferPaused(paused);
+   }
```

3S-Fund-L04

Redemption requests submitted inside the active notice window receive a cancel deadline in the past

Id	3S-Fund-L04
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #36a0656.

Description

FundConfigLib::tryResolveRedemptionSchedule decides which notice window a new redemption request attaches to and what its **cancelDeadlineAt** becomes. The first branch handles **nowTs** before the current window, the second handles **nowTs** between the two windows, and the third branch fires when **nowTs** falls inside **[nextRedemptionNoticeWindow.start, nextRedemptionNoticeWindow.end)**. The third branch returns **(true, nextRedemptionNoticeWindow.end(), nextRedemptionNoticeWindow.start())**, so **cancelDeadlineAt = nextRedemptionNoticeWindow.start()**, which is already in the past relative to **nowTs**.

Two consequences follow. First, the investor has no time to cancel, because **AncFund::cancelRedemption** requires **block.timestamp <= cancelDeadlineAt**. Second, **AncFund::setPriceId** lets the settler price the request immediately in the same block, because its gate **block.timestamp <= cancelDeadlineAt** no longer holds. The settler relies on knowing every pending request before the notice window opens to compute the NAV. A request created mid-window violates that invariant and corrupts the cycle's accounting.

The branch only fires when the operator forgets to rotate the schedule after the previous cycle ends. **AncFund::submitRedemptionRequest** already reverts with **InvalidRedemptionNoticeWindows** when **tryResolveRedemptionSchedule** returns **found = false**. Removing the branch surfaces the stale configuration to the caller.

Steps to Reproduce

1. The operator configures **redemptionNoticeWindow = [10, 20]** and **nextRedemptionNoticeWindow = [30, 40]**.

2. Time advances to **block.timestamp = 35**, inside **nextRedemptionNoticeWindow**. The operator has not yet called **setRedemptionNoticeWindows** to rotate to a new pair.

3. An investor calls **AncFund::submitRedemptionRequest**.

4. **tryResolveRedemptionSchedule** returns **(true, 40, 30)**. The request stores **cancelDeadlineAt = 30**.

5. The investor cannot cancel because **block.timestamp = 35 > 30**.

6. The settler immediately calls **AncFund::setPriceId** for the new request; the **block.timestamp <= cancelDeadlineAt** check at **setPriceId** passes, and the request is priced against the in-flight NAV.

Recommendation

Drop the third branch so that **tryResolveRedemptionSchedule** returns **(false, 0, 0)** once **nowTs >= nextRedemptionNoticeWindow.start()**. The caller already converts a **false** result into a revert.

```
// contracts/libs/fund/FundConfigLib.sol
        if (nowTs < config.nextRedemptionNoticeWindow.start()) {
            return (true, config.nextRedemptionNoticeWindow.end(),
config.nextRedemptionNoticeWindow.start());
        }
-       if (nowTs < config.nextRedemptionNoticeWindow.end()) {
-           return (true, config.nextRedemptionNoticeWindow.end(),
config.nextRedemptionNoticeWindow.start());
-       }
        return (false, 0, 0);
```

3S-Fund-L05

Missing **transferPaused** check in **detectTransferRestriction**

Id	3S-Fund-L05
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #d20fc71.

Description

AncFundToken implements the ERC-1404 restricted-token interface so that off-chain consumers (front-ends, brokers, custodians, order management systems) can ask the token, before submitting a transfer, whether that transfer would be allowed. The view function **AncFundToken::detectTransferRestriction** is the canonical entry point for that query: it takes a **(from, to, amount)** triple and is expected to return the same restriction code that would be raised if a real transfer were attempted under the current state of the token. Consumers rely on this contract to decide whether to surface a transfer to the user, queue it, or pre-emptively reject it.

The token enforces two independent gates inside **AncFundToken::_update**. The first is **transferPaused**: when set, any non-mint/non-burn movement reverts with **TransferPaused** regardless of the compliance status of either party. The second is the compliance lookup performed by **_restrictionCode**, which queries **AncCompliance** for the sender and recipient. **detectTransferRestriction** delegates straight to **_restrictionCode** and therefore reports only on the second gate; it never inspects **transferPaused**. When the token is paused, the view continues to return **_RESTRICTION_ALLOWED** (zero) for any otherwise-compliant pair, while a real call to **transfer** or **transferFrom** reverts immediately at the **if (transferPaused) revert TransferPaused();** branch.

The consequence is a divergence between what off-chain consumers observe and what the chain enforces. A broker or front-end polling **detectTransferRestriction** to gate user actions will allow share transfers through its UI, only for those transfers to fail at submission. There is no separate restriction code surfaced for the paused state, so consumers cannot even disambiguate the failure category from **messageForTransferRestriction**. The standard's intent is precisely to avoid this kind of false-positive, so consumers can render an accurate reason to the user rather than relying on revert-message introspection. The same gap applies for any future automation (compliance reporting, transfer-attempt analytics) that treats the view as ground truth.

Recommendation

Introduce a fourth restriction code, for example **_RESTRICTION_PAUSED**, and check **transferPaused** at the top of **AncFundToken::detectTransferRestriction** (or inside a small wrapper used by both the view and **_update**) before delegating to the compliance lookup. Surface the same code from **AncFundToken::messageForTransferRestriction** so that off-chain consumers have a stable string for the paused condition. Keep the on-chain enforcement in **_update** unchanged so paused transfers continue to revert with the existing **TransferPaused** error; the change is purely additive to the view-side surface.

```
    uint8 private constant _RESTRICTION_ALLOWED = 0;
    uint8 private constant _RESTRICTION_RECIPIENT_NOT_APPROVED = 1;
    uint8 private constant _RESTRICTION_SENDER_NOT_APPROVED = 2;
+    uint8 private constant _RESTRICTION_PAUSED = 3;



    function detectTransferRestriction(address from, address to, uint256 amount)
        external
        view
        returns (uint8 restrictionCode)
    {
+        if (transferPaused) return _RESTRICTION_PAUSED;
        return _restrictionCode(from, to, amount);
    }
    function messageForTransferRestriction(uint8 restrictionCode) external pure
returns (string memory message) {
        if (restrictionCode == _RESTRICTION_ALLOWED) return "ALLOWED";
        if (restrictionCode == _RESTRICTION_RECIPIENT_NOT_APPROVED) return
"RECIPIENT_NOT_APPROVED";
        if (restrictionCode == _RESTRICTION_SENDER_NOT_APPROVED) return
"SENDER_NOT_APPROVED";
+        if (restrictionCode == _RESTRICTION_PAUSED) return "TRANSFER_PAUSED";
        return "UNKNOWN_RESTRICTION";
    }
```

3S-Fund-N01

Declared **Settled** and **Cancelled** request statuses are never written

Id	3S-Fund-N01
Classification	None
Category	Suggestion
Status	Addressed in #9d198cc.

Description

The **RequestStatus** enum in **contracts/libs/fund/FundTypes.sol:22-30** declares seven members: **None**, **Funded**, **Pending**, **PriceSet**, **Settleable**, **Settled**, **Cancelled**. The protocol writes the first five during the normal subscription and redemption lifecycle, but the two terminal members **Settled** and **Cancelled** are never assigned anywhere in the codebase.

Both completion paths instead clear the slot. **AncFund::_settleSubscription** (**AncFund.sol:401**), **AncFund::_settleRedemption** (**AncFund.sol:434**), and **AncFund::cancelRedemptionRequest** (**AncFund.sol:224**) all execute **delete** on the request, which resets the storage slot to its default zero value **RequestStatus.None**. A request that has successfully completed settlement or cancellation therefore reads back as **status == RequestStatus.None**, indistinguishable from a request id that was never created.

The on-chain surface is unaffected: every state-transition guard checks against the expected pre-transition status, and **None** is correctly treated as "not found". The drift is purely between the declared type and the realized behavior, and the consequence falls on off-chain consumers. Indexers, dashboards, and any client polling **subscriptionRequests[id].status** or **redemptionRequests[id].status** after a terminal action see **0** and cannot distinguish a settled request, a cancelled request, and a request id that never existed. A naive integration that follows the documented enum surfaces "request not found" for a user who in fact had their subscription successfully settled.

Recommendation

Realign the enum with the actual runtime behavior by removing **Settled** and **Cancelled** from **RequestStatus**, and add NatSpec on the enum (and on the **subscriptionRequests** / **redemptionRequests** mappings in **AncFund**) stating that a slot reading **RequestStatus.None** is either nonexistent or has reached a terminal state via **delete**, and that off-chain consumers must rely on the lifecycle events (**SubscriptionSettled**, **RedemptionSettled**, **RedemptionCancelled**) to distinguish the two.

```
enum RequestStatus {
    None,
    Funded,
    Pending,
    PriceSet,
-   Settleable,
-   Settled,
-   Cancelled
+    Settleable
}
```

3S-Fund-N02

Missing upper bound on **settleableAt** in **AncFund::markSettleable** lets an operator typo lock a request indefinitely

Id	3S-Fund-N02
Classification	None
Category	Suggestion
Status	Addressed in #5ef845d.

Description

AncFund::markSettleable is the second settlement step. The settler calls it after **setPriceId** to attach a **settleableAt** timestamp and advance the request from **PriceSet** to **Settleable**. **AncFund::_settleSubscription** and **AncFund::_settleRedemption** then revert until **block.timestamp >= request.settleableAt**. The function validates only that **settleableAt != 0** and, for redemption requests, that **settleableAt >= request.effectiveAt**. Subscription requests have no temporal floor, and neither path enforces an upper bound. A misplaced zero, a seconds-versus-milliseconds slip, or a copy-paste from the wrong column of an off-chain spreadsheet can therefore pin **settleableAt** decades into the future.

The request lifecycle offers no recovery. **markSettleable** requires **PriceSet**, so it cannot rewrite the timestamp once the request reaches **Settleable**. **AncFund::cancelRedemption** requires **Pending** and reverts on **Settleable**, and subscription requests have no cancel path at any stage. A stuck redemption pins the investor's shares in the **lockedShares** mapping. A stuck subscription is worse, because **AncFund::collectFund** has already transferred the investor's assets out of the contract before **markSettleable** runs; the investor paid in, and the contract owes shares that it will not mint until the runaway timestamp arrives.

Recommendation

Pass a maximum settle horizon as an immutable constructor argument. Inside **markSettleable**, reject any **settleableAt** that is not strictly in the future, and reject any value that exceeds **block.timestamp + MAX_SETTLEABLE_DAY**.

```
// contracts/fund/AncFund.sol
+    uint32 public immutable MAX_SETTLEABLE_DAY;
-    constructor(address router_, address navPriceRegistry_, address
fundController_) {
```

```
+    constructor(address router_, address navPriceRegistry_, address
fundController_, uint32 maxSettleableDay_) {
        if (router_ == address(0) || navPriceRegistry_ == address(0) || fundController_
== address(0)) {
            revert ZeroAddress();
        }
+        if (maxSettleableDay_ == 0) revert InvalidSettleableAt();
        ROUTER = router_;
        NAV_RECORD = navPriceRegistry_;
        CONTROLLER = fundController_;
+        MAX_SETTLEABLE_DAY = maxSettleableDay_;
        _disableInitializers();
    }
    function markSettleable(...) external nonReentrant {
        ...
        uint32 settleableAt = settleableAtList[i];
        if (settleableAt == 0) revert InvalidSettleableAt();
+        if (settleableAt <= block.timestamp) revert InvalidSettleableAt();
+        if (settleableAt >= block.timestamp + MAX_SETTLEABLE_DAY) revert
InvalidSettleableAt();
        ...
    }
```

3S-Fund-N03

FUND address subject to compliance checks can lead to DoS

Id	3S-Fund-N03
Classification	None
Category	Suggestion
Status	Addressed in #339392e.

Description

AncFundToken is the per-fund share token that gates every share transfer through the protocol's compliance layer. On any non-mint/non-burn movement, **AncFundToken::_update** calls **AncFundToken::_restrictionCode**, which queries **AncCompliance** for both the sender and the recipient and requires each of them to be approved, not denylisted, and not sanction-listed. Mints (**from == address(0)**) and burns (**to == address(0)**) are intentionally exempted from this check at **AncFundToken.sol:107**, but no equivalent exemption exists for the protocol's own **FUND** address.

The **FUND** proxy is, by design, a counterparty in two core investor flows. During redemption submission, **AncFundRouter::requestRedemption** performs **transferFrom(investor, FUND, shares)**, making **FUND** the recipient and therefore subjecting **FUND** to the recipient-side compliance check. During cancellation, **AncFund::cancelRedemption** performs **transfer(FUND, investor, shares)** to return the locked shares, making **FUND** the sender and subjecting it to the sender-side compliance check. In both cases, the operation succeeds only while **FUND** itself is considered compliant by **AncCompliance**.

FUND is a protocol-internal address and should never be treated as an end user, yet its compliance status is governed by the same role-controlled lists that apply to investors. A single holder of **APPROVAL_REMOVER_ROLE**, **DENYLIST_ADDER_ROLE**, or **SANCTION_ADDER_ROLE** on **AncCompliance** can, with one transaction, place **FUND** into a non-compliant state. This is not a hypothetical adversarial scenario: routine operational activity such as a scripted denylist sweep, a bulk import of a sanctions list, or an accidental approval revocation can include the **FUND** address by mistake, since nothing on the compliance contract distinguishes a protocol-internal address from a regular investor.

Because **FUND** is shared across every fund deployed through **AncFundController**, a single compliance flip on **FUND** simultaneously bricks redemption submissions and pending-redemption cancellations on every fund in the protocol. The settlement burn path keeps functioning (it routes through the **to == address(0)** bypass), which creates an additional asymmetry: already-priced redemptions can still be settled on the operator's schedule, while investors lose the ability to submit new redemptions or cancel existing ones during the same window. The invariant that should hold here is unconditional: **FUND** must

always be able to receive and send shares as part of normal protocol operation, regardless of the state of compliance lists, because **FUND** is the protocol itself, not a participant in it.

Recommendation

Exempt the **FUND** address from compliance lookups on a per-side basis: the sender-side and recipient-side checks should each be skipped only when that specific side is **FUND**, so the investor counterparty is still validated. Persist the **FUND** address on the token at construction or initialization as an immutable (it is already known to **AncFundController** at deploy time and passed into the token), then guard each compliance branch with a **FUND** check.

```
 function _restrictionCode(address from, address to, uint256 amount) internal view
returns (uint8) {
    amount; // silence the warning
    IAncCompliance compliance = IAncCompliance(COMPLIANCE);
-   if (!compliance.isApproved(from) || compliance.isDenylisted(from) ||
compliance.isSanctionListed(from)) {
+    if (from != FUND && (!compliance.isApproved(from) ||
compliance.isDenylisted(from) || compliance.isSanctionListed(from))) {
        return _RESTRICTION_SENDER_NOT_APPROVED;
    }
-   if (!compliance.isApproved(to) || compliance.isDenylisted(to) ||
compliance.isSanctionListed(to)) {
+    if (to != FUND && (!compliance.isApproved(to) || compliance.isDenylisted(to) ||
compliance.isSanctionListed(to))) {
        return _RESTRICTION_RECIPIENT_NOT_APPROVED;
    }
    return _RESTRICTION_ALLOWED;
 }
```

3S-Fund-N04

Non-atomic **MINTER_ROLE** provisioning in **launchFund**

Id	3S-Fund-N04
Classification	None
Category	Suggestion
Status	Addressed in #9c13ff5.

Description

AncFundController::launchFund is the entry point that creates a new fund. It deploys the per-fund share token as a **BeaconProxy** and registers a **FundConfig** for the new **fundId**, so that downstream functions (**setSubscriptionWindow**, **requestSubscription**, settlement) can address the fund. The freshly deployed share token holds **DEFAULT_ADMIN_ROLE** on the controller side but has zero holders of **MINTER_ROLE** and **BURNER_ROLE** at the end of the call: **launchFund** does not provision those roles on the new token. The operator is expected to follow up with a separate transaction, **AncFundController::grantFundTokenMinterRole(fundId, address(FUND))**, to give the protocol's **FUND** proxy permission to mint shares for that fund.

Nothing on the controller surface enforces this follow-up before the fund becomes usable. **setSubscriptionWindow** only validates the window itself and the fund's existence; it does not check that **MINTER_ROLE** has a holder, and there is no precondition on **requestSubscription** that the share token is ready to mint. If the follow-up **grantFundTokenMinterRole** call is forgotten, sent against the wrong **fundId**, or sent with a **minter** other than **address(FUND)**, the fund still appears fully open to investors. Subscription requests can be submitted through the router, **AncFundRouter::requestSubscription** pulls the gross asset into **FUND**, the settler proceeds through **setPriceId** and **markSettleable**, and **AncFund::collectFund** forwards the net asset to the off-protocol **outgoing** sink configured on the settler. Only when settlement reaches **AncFund::_settleSubscription** does the missing grant manifest: the call to **IAncFundToken.mint(investor, sharesMinted)** at **AncFund.sol:396** reverts with **AccessControlUnauthorizedAccount(FUND, MINTER_ROLE)**.

At that point the investor's cash has already left **FUND** custody and is sitting at the settler's external **outgoing** sink, while no investor-callable **cancelSubscription** path exists on the router or on **AncFund** to pull it back. The underlying issue is that role provisioning on the share token is detached from the fund's lifecycle. **launchFund** already knows the immutable **FUND** address (it is a constructor parameter on **AncFundController**) and already holds **DEFAULT_ADMIN_ROLE** on the newly minted token, so the grant could be performed atomically inside **launchFund**. Keeping it as a separate operator step turns a

single forgotten transaction into a path where investor cash exits the protocol perimeter before the fund is actually capable of settling.

Recommendation

Inside **AncFundController::launchFund**, after the **BeaconProxy** is deployed and the **FundConfig** is written, atomically grant **MINTER_ROLE** and **BURNER_ROLE** on the new share token to the immutable **FUND** address. This binds the role provisioning to fund creation, so a fund is either fully usable or does not exist at all, and no operator follow-up can be skipped or misdirected. The freeform **AncFundController::grantFundTokenMinterRole(uint16, address)** and **AncFundController::grantFundTokenBurnerRole(uint16, address)** wrappers can be removed from the controller surface once the atomic grant is in place.

```
    bytes memory tokenInitData = abi.encodeCall(IAncFundToken.initialize,
(fundName, fundSymbol));
    token = address(new BeaconProxy(BEACON, tokenInitData));
    if (_fundConfigs[fundId].shareToken != address(0)) revert FundAlreadyExists();
    _fundConfigs[fundId] = FundConfig({
        index: fundId,
        shareToken: token,
        subscriptionWindow: TimeWindowLib.pack(0, type(uint32).max),
        redemptionNoticeWindow: 0,
        nextRedemptionNoticeWindow: 0,
        feeBps: _DEFAULT_FEE_BPS
    });
+   IAncFundToken shareToken = IAncFundToken(token);
+   shareToken.grantRole(shareToken.MINTER_ROLE(), FUND);
+   shareToken.grantRole(shareToken.BURNER_ROLE(), FUND);
    emit FundRegistered(fundId, token);
    nextFundId = fundId + 1;
```