Dre Mortgage Security Review

Auditors

0xRajeev, Lead Security Researcher

R0bert, Lead Security Researcher

Chinmay Farkya, Security Researcher

Kamensec, Associate Security Researcher

Report prepared by: Lucas Goiriz

March 5, 2026

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Dre Mortgage according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 6 days in total, Dre Labs engaged with Spearbit to review the dreusd protocol. In this period of time a total of **73** issues were found.

Summary

Project Name	Dre Labs
Repository	dreusd
Commit	49e08974
Type of Project	DeFi, Stablecoin
Audit Timeline	Feb 8th to Feb 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	13	11	2
Low Risk	25	19	6
Gas Optimizations	3	2	1
Informational	30	24	6
Total	**73**	**58**	**15**

The Spearbit team reviewed Dre Labs's dreusd holistically on commit hash 3c0e338a and concluded that all findings were addressed and no new vulnerabilities were identified.

4.1 Scope

The security review had the following components in scope for dreusd on commit hash 49e08974:

```
contracts
├── dreAaveAdapter.sol
├── dreRewardsDistributor.sol
├── dreUSD.sol
├── dreUSDManager.sol
├── dreUSDOracle.sol
├── dreUSDs.sol
├── dreWithdrawalNFT.sol
├── governance
│   └── dreTimelockController.sol
└── ovault
    ├── dreOVaultComposer.sol
    ├── dreShareOFT.sol
    └── dreShareOFTAdapter.sol
```

5 Findings

5.1 High Risk

5.1.1 Fiat mint struct hash clobbers memory

Severity: High Risk

Context: dreUSDManager.sol#L932-L950

Description: The fiat mint flow in `dreUSDManager` computes a struct hash in `_computeFiatMintStructHash` using inline assembly that writes to fixed memory locations including `0x40` and `0x60`. In Solidity, memory slot `0x40` stores the free memory pointer and `0x60` is reserved as the zero slot. Overwriting these reserved slots can corrupt subsequent memory allocation and ABI encoding performed by the compiler.

This struct hash is computed inside `_mintFromFiatUsd` (used by `mintFromUsd` and `mintRewards`) immediately before calling `ECDSA.recover(ethSignedHash,custodianSig)`. Because `ECDSA.recover` takes `bytes memory`, passing the `bytes calldata custodianSig` forces a calldata-to-memory copy that uses the free memory pointer stored at `0x40`. Since `_computeFiatMintStructHash` overwrites `0x40` with `usdAmount`, the compiler-generated memory copy can attempt to allocate/copy at a very large offset, triggering excessive memory expansion and out-of-gas, or otherwise producing unreliable signature verification behavior.

This failure mode does not require an astronomically large `usdAmount`. `usdAmount` is expressed in USD with 2 decimals, so $10,000.00 corresponds to `usdAmount = 1_000_000`. Since the clobbered free memory pointer is interpreted as a byte offset, even moderate fiat mints can force the EVM to expand memory into the megabytes. Because memory expansion cost grows roughly quadratically with the highest accessed memory word, realistic `usdAmount` values can make fiat minting become unexecutable under typical transaction/block gas limits. As a rough reference point, expanding memory to around 1,000,000 bytes costs on the order of 2,000,000 gas, while expanding to around 4,000,000 bytes costs on the order of 31,000,000 gas, which can exceed common gas ceilings.

```
// dreUSDManager
bytes32 structHash = _computeFiatMintStructHash(m.mintRef, m.receiver, m.usdAmount,
↪  m.validUntil, m.chainId);
bytes32 ethSignedHash = MessageHashUtils.toEthSignedMessageHash(structHash);
signer = ECDSA.recover(ethSignedHash, custodianSig);
```

```
// dreUSDManager
function _computeFiatMintStructHash(bytes32 a, address b, uint256 c, uint256 d, uint256 e)
↪  internal pure returns (bytes32 hashedVal) {
    assembly {
        mstore(0x00, a)
        mstore(0x20, b)
        mstore(0x40, c) // clobbers free memory pointer
        mstore(0x60, d) // clobbers zero slot
        mstore(0x80, e)
        hashedVal := keccak256(0x00, 0xa0)
    }
}
```

The impact is that keeper-driven fiat minting and reward minting can revert or behave unpredictably depending on the fiat mint parameters, turning a core operational pathway into a denial of service and increasing the chance of production incidents.

Find the proof of concept at gist 0d18a224.

Recommendation: In order to address this, do not write to reserved memory slots in inline assembly. The simplest fix is to replace the assembly hashing with a safe Solidity expression like `keccak256(abi.encode(mintRef,receiver,usdAmount,validUntil,chainId))` (which matches the current word-aligned encoding). If assembly is preferred, write to a buffer starting at the free memory pointer (for example, `let ptr := mload(0x40)`) and avoid modifying `0x40` and `0x60` so downstream memory allocations remain well-defined.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified. `dreUSDManager._computeFiatMintStructHash` was changed from inline assembly (`mstore(0x40), mstore(0x60)`) to safe `keccak256(abi.encode(...))`.

5.1.2 Rewards can become unaccounted and stuck

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: This issue was introduced in the commit e81e17b - chore: anti share inflation attack, during the fix period.

The vault accounting model in `dreUSDs` intentionally avoids using raw token balance and instead derives ERC4626 accounting from `_virtualBalance` plus distributor-reported vesting. This means direct token transfers into the vault are not automatically reflected in `totalAssets()`.

```
function totalAssets() public view override returns (uint256) {
    uint256 vested = rewardsDistributor != address(0)
        ? IdreRewardsDistributor(rewardsDistributor).vestedAmount()
        : 0;
    return _virtualBalance + vested;
}
```

The vault attempts to synchronize `_virtualBalance` only when its own deposit and withdraw hooks execute, by calling `claimVested()` and adding the returned amount. This sync does not happen for transfers that occur outside those hooks.

```
function _claimVestedRewards() internal returns (uint256 claimed) {
    if (rewardsDistributor == address(0)) return 0;
    return IdreRewardsDistributor(rewardsDistributor).claimVested();
}

function _deposit(address caller, address receiver, uint256 assets, uint256 shares) internal
↪   override whenNotPaused {
    _virtualBalance += _claimVestedRewards();
    _virtualBalance += assets;
    super._deposit(caller, receiver, assets, shares);
}

function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal override whenNotPaused {
    _virtualBalance += _claimVestedRewards();
    _virtualBalance -= assets;
    super._withdraw(caller, receiver, owner, assets, shares);
}
```

In `dreRewardsDistributor`, `claimVested()` is callable by anyone, and `addRewards()` also flushes vested rewards through `_claimVested()`. Both paths transfer `dreUSD` directly into the vault.

```
function addRewards() external onlyRole(MODERATOR_ROLE) whenNotPaused {
    _claimVested();
//...
}

function claimVested() external whenNotPaused returns (uint256 claimed) {
    return _claimVested();
}
```

```
function _claimVested() internal returns (uint256 vested) {
    (vested, newClaimTimestamp) = _computeVestedAmount();
    if (vested > 0) {
        IERC20(dreUSD).safeTransfer(vault, vested);
        rewards = rewards - vested;
        cTs = newClaimTimestamp;
    }
}
```

When either of those distributor paths runs without passing through vault deposit or withdraw hooks, vault token balance increases but `_virtualBalance` does not. At the same time, distributor `vestedAmount()` decreases after claim. The net effect is a persistent accounting divergence where real assets held by the vault become invisible to ERC4626 conversion math. This can cause rewards to become practically unclaimable through normal vault accounting and can produce materially incorrect share/asset conversions over time.

Recommendation: Unify reward settlement so every transfer of rewards into the vault also updates vault accounting in the same call path. One robust approach is to restrict distributor claims to the vault and force all claiming through a vault function that increments `_virtualBalance` from the returned amount.

```
// in distributor
function claimVested() external onlyVault whenNotPaused returns (uint256 claimed) {
    return _claimVested();
}

// in vault
function claimRewards() public {
    uint256 claimed = IdreRewardsDistributor(rewardsDistributor).claimVested();
    _virtualBalance += claimed;
}
```

`addRewards()` should not directly flush vested rewards unless it routes through the same vault accounting entrypoint.

Dre Labs: Fixed in commit 66f6d063.

Spearbit: Fix verified.

5.2 Medium Risk

5.2.1 Stale cTs causes instant reward vesting

Severity: Medium Risk

Context: dreRewardsDistributor.sol#L115-L119

Description: The rewards distributor uses a linear vesting schedule defined by a start timestamp (`cTs`) and an end timestamp (`eTs`). In `addRewards()`, when there are no in-progress rewards (rewards == 0) and new rewards are added, the contract sets rewards and `eTs` but does not reset `cTs` to the current time. As a result, if the contract has been idle since initialization or since the prior vesting stream completed, `cTs` can be far in the past.

```
// dreRewardsDistributor
if (rewards == 0) {
    rewards = newRewards;
    eTs = block.timestamp + vestPeriod;
}
```

Because vested rewards are computed as a fraction of (now - `cTs`) over (`eTs` - `cTs`), a stale `cTs` makes a non-trivial portion of the newly added rewards immediately appear vested right after `addRewards()` runs, even though the intent is to start vesting from the top-up time. A user can deposit into the vault shortly before the moderator

tops up rewards and capture value as if they had been staked during the idle period. The impact is incorrect and unfair reward distribution driven by predictable timing.

Recommendation: When starting a new vesting stream (rewards == 0 and `newRewards` > 0), reset the vesting start to the current timestamp and set the end from that same start time, for example:

```
cTs = block.timestamp;
eTs = block.timestamp + vestPeriod;
rewards = newRewards;
```

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `f6f3f36`, `dreRewardsDistributor.addRewards()` was updated so that when `rewards == 0`, it now sets `cTs = block.timestamp` before setting `eTs`.
2. In final commit `3c0e338`, that same logic is still present (`rewards = newRewards; cTs = block.timestamp; eTs = block.timestamp + VEST_PERIOD;`), so stale `cTs` is no longer carried into a new stream.
3. A regression test was added in the same fix (`test_AddRewards_WhenIdle_ResetsCTs_SoVestingStartsFromNow`) and is present in `3c0e338`.

5.2.2 VestPeriod under 1 day can brick addRewards

Severity: Medium Risk

Context: dreRewardsDistributor.sol#L125-L128

Description: In `addRewards()`, the contract compares a computed `newVestPeriod` against an upper and lower bound. The lower-bound check is written as a comparison against (`vestPeriod - 1 days`). If `vestPeriod` is configured to a value below 1 day, this subtraction underflows and reverts in Solidity 0.8+, causing `addRewards()` to unexpectedly fail in the rewards > 0 code path whenever that branch evaluates the lower-bound expression.

```
// dreRewardsDistributor
if (newVestPeriod > vestPeriod || newVestPeriod < (vestPeriod - 1 days)) {
    cTs = block.timestamp;
    eTs = block.timestamp + vestPeriod;
}
```

This creates an avoidable operational failure mode where reward top-ups can revert depending on configuration, effectively bricking reward distribution until the configuration is corrected.

Recommendation: Either enforce `vestPeriod` >= 1 days when updating `vestPeriod`, or restructure the lower-bound logic to avoid an underflow when `vestPeriod` is less than 1 day. For example, compute a safe lower bound and compare against it:

```
uint256 lowerBound = vestPeriod > 1 days ? vestPeriod - 1 days : 0;
if (newVestPeriod > vestPeriod || newVestPeriod < lowerBound) {
    cTs = block.timestamp;
    eTs = block.timestamp + vestPeriod;
}
```

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `f404a87`, `dreRewardsDistributor` changed vesting period from mutable storage to a fixed constant (`VEST_PERIOD = 7 days`) and removed the `setVestPeriod` entrypoint.
2. In final commit `3c0e338`, the same fixed-period design remains (`VEST_PERIOD` only, no configurable vest period), so `VEST_PERIOD - 1 days` cannot underflow from misconfiguration.

3. The interface and tests in `3c0e338` were aligned with this change (no `setVestPeriod` / `ZeroVestPeriod`, and tests use `VEST_PERIOD`).

5.2.3 Express limit decrease can break payback

Severity: Medium Risk

Context: dreUSDManager.sol#L711-L717

Description: The express withdrawal subsystem tracks two related values: `expressWithdrawalMaxLimit` (a configured maximum) and `expressWithdrawalAvailable` (current headroom). `updateExpressWithdrawal(maxLimit, feeBps, feeRecipient)` allows changing `expressWithdrawalMaxLimit`, but it does not reconcile `expressWithdrawalAvailable` against the new limit. If `expressWithdrawalMaxLimit` is reduced below `expressWithdrawalAvailable`, the invariant `expressWithdrawalAvailable <= expressWithdrawalMaxLimit` is broken.

Later, when paying back express filler debt, `_paybackExpressFiller` computes headroom as `expressWithdrawalMaxLimit - expressWithdrawalAvailable`. If `expressWithdrawalAvailable` is greater than `expressWithdrawalMaxLimit`, this subtraction underflows and reverts, which can block `payExpressDebt()` and prevent operators from restoring express capacity via payback. Separately, `requestExpressWithdrawal()` caps express routing using `expressWithdrawalAvailable`, so lowering `expressWithdrawalMaxLimit` does not immediately constrain express requests unless `expressWithdrawalAvailable` is also reduced. The impact is an operational denial of service and inconsistent enforcement of the intended express withdrawal limit.

```
// dreUSDManager
function updateExpressWithdrawal(uint256 maxLimit, uint256 feeBps, address feeRecipient)
↪  external {
//...
    expressWithdrawalMaxLimit = maxLimit;
//...
}

function _paybackExpressFiller(uint256 amount) internal {
//...
    uint256 limitHeadroom = expressWithdrawalMaxLimit - expressWithdrawalAvailable;
//...
}
```

Recommendation: Consider maintaining an explicit invariant that `expressWithdrawalAvailable` is always less than or equal to `expressWithdrawalMaxLimit`. When updating the express parameters, clamp `expressWithdrawalAvailable` down to the new `maxLimit`. Also defensively compute headroom to avoid underflow if configuration is ever inconsistent. One concrete approach is:

```
expressWithdrawalMaxLimit = maxLimit;
if (expressWithdrawalAvailable > maxLimit) {
    expressWithdrawalAvailable = maxLimit;
}
```

Additionally, compute headroom with a conditional expression so misconfiguration cannot brick payback. For example:

```
uint256 limitHeadroom =
    expressWithdrawalMaxLimit > expressWithdrawalAvailable
        ? expressWithdrawalMaxLimit - expressWithdrawalAvailable
        : 0;
```

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `4db6017`, `updateExpressWithdrawal` now clamps `expressWithdrawalAvailable` to `maxLimit` when reducing limits, enforcing `expressWithdrawalAvailable <= expressWithdrawalMaxLimit`.
2. In final commit `3c0e338`, `_paybackExpressFiller` computes headroom defensively using a conditional expression, preventing underflow when state is inconsistent.
3. Regression coverage is present in `3c0e338`, including tests for clamping available on limit decrease (`test_UpdateExpressWithdrawal_ClampsAvailableToMaxLimit`) and payback/headroom behavior (`test_PayExpressDebt_LimitHeadroom`).

5.2.4 Composer refund can fail without `msg.value`

Severity: Medium Risk

Context: dreOVaultComposer.sol#L22

Description: The hub `dreOVaultComposer` inherits `VaultComposerSync` and is intended to automatically refund users on compose execution failures. In `lzCompose`, the composer calls `handleCompose` and, on any error other than `InsufficientMsgValue`, attempts to refund the received OFT balance back to the source chain via `_refund`.

However, the refund send is funded solely by the compose call's `msg.value`. `_refund` always routes through `_sendRemote`, which forwards the full `_msgValue` as the `IOFT.send` fee payment. If the compose call is executed with `msg.value == 0` (or otherwise underfunded for the refund message fee), the refund attempt can revert, causing `lzCompose` to revert and preventing the intended automatic refund. The bridged tokens remain credited to the composer contract until someone manually retries the compose execution through the endpoint with sufficient `msg.value` to either complete the vault operation or successfully execute the refund send.

This is especially easy to hit in flows where the "second hop" is local to the hub (when `SendParam.dstEid` equals the local `VAULT_EID`), because `_sendLocal` ignores `msg.value` and does not refund it. Callers will naturally choose `msg.value == 0` to avoid permanently locking native tokens on successful local sends, but that same choice makes the failure path brittle because refunds still require a non-zero message fee.

```
// VaultComposerSync.sol
try this.handleCompose{ value: msg.value }(_composeSender, composeFrom, composeMsg, amount) {
    emit Sent(_guid);
} catch (bytes memory _err) {
    if (bytes4(_err) == InsufficientMsgValue.selector) {
        assembly {
            revert(add(32, _err), mload(_err))
        }
    }
    _refund(_composeSender, _message, amount, tx.origin, msg.value);
    emit Refunded(_guid);
}

function _refund(address _oft, bytes calldata _message, uint256 _amount, address
↪   _refundAddress, uint256 _msgValue) internal virtual {
    SendParam memory refundSendParam;
    refundSendParam.dstEid = OFTComposeMsgCodec.srcEid(_message);
    refundSendParam.to = OFTComposeMsgCodec.composeFrom(_message);
    refundSendParam.amountLD = _amount;
    _sendRemote(_oft, refundSendParam, _refundAddress, _msgValue);
}
```

The impact is user funds being stranded on the hub until an operator or user retries compose execution with sufficient `msg.value` and an inability to simultaneously

1. Safely overfund compose execution and...
2. Avoid permanently locking native tokens on successful local-second-hop executions.

Recommendation: Consider making compose refund behavior robust against misfunding by ensuring there is a safe way to overfund compose execution without permanently locking native tokens on success. One concrete approach is to override `_sendLocal` in `dreOVaultComposer` to refund any non-zero `_msgValue` to `_refundAddress` after performing the local `ERC20` transfer, so callers can provide enough `msg.value` to cover worst-case refund fees without risking permanent native-token lockup on successful local sends.

Additionally, consider using the existing `minMsgValue` parameter to enforce a minimum `msg.value` that is sufficient for the expected refund and/or second-hop send fee for the pathway, so underfunded compose executions fail early with `InsufficientMsgValue` and can be retried deterministically with the correct `msg.value`.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `8b65641`, `dreOVaultComposer` overrides `_sendLocal` to refund non-zero `_msgValue` after successful local sends.
2. In final commit `3c0e338`, this `_sendLocal` refund behavior remains, with fallback from `_refundAddress` to `msg.sender` and explicit revert if both refund attempts fail.
3. Regression tests in `3c0e338` (`test/ovault/dreOVaultComposer.t.sol`) cover local-send refund paths, including direct refund, alternate refund recipient, fallback-to-sender, and double-reject revert cases.

5.2.5 Sanctions checks are inconsistent across flows

Severity: Medium Risk

Context: dreUSDManager.sol#L420

Description: The protocol appears to treat `dreUSD.validateAddress()` as the canonical compliance gate (sanctions + freeze), but enforcement is fragmented across `dreUSDManager`, `dreWithdrawalNFT`, and `dreUSDs`.

In withdrawal fill paths (`fillWithdrawal` and `fillExpressWithdrawals`), the manager reads the current NFT owner and only checks sanctions via `sanctionsList.isSanctioned(currentOwner)`. This omits freeze checks and bypasses the manager's own `_validateAddress` helper. At the same time, `dreWithdrawalNFT` does not enforce compliance in transfer hooks (`_update`), so both standard withdrawal NFTs (`withdrawalNFT`) and express withdrawal NFTs (`expressWithdrawalNFT`) can be moved to another address after request creation. Combined behavior creates a policy gap where compliance behavior differs by path and by timing of transfer.

`mintFrom` also lacks `_validateAddress(from)`, so payer-side validation is not consistently applied before pulling stablecoins. Additionally, `_mintFromFiatUsd` only checks `m.receiver != address(0)` and does not call manager-level `_validateAddress(m.receiver)`, and `dreUSDs._withdraw` does not explicitly validate the withdrawal `receiver` before dispatching assets. Some of these paths may currently rely on downstream token hooks, but the policy is not enforced at one clear layer and can drift over upgrades or refactors.

```
// dreUSDManager.fillWithdrawal / fillExpressWithdrawals
address currentOwner = IERC721(withdrawalNFT).ownerOf(tokenId);
address expressOwner = IERC721(expressWithdrawalNFT).ownerOf(tokenId);
address list = IdreUSD(dreUSD).sanctionsList();
if (list != address(0) && ISanctionsList(list).isSanctioned(currentOwner)) {
    revert SanctionedAddress(currentOwner);
}
if (list != address(0) && ISanctionsList(list).isSanctioned(expressOwner)) {
    revert SanctionedAddress(expressOwner);
}

// dreUSDManager.mintFrom
_executePermit(from, asset, amountIn, permitSig);
uint256 dreUSDAmount = _transferAndMint(asset, from, amountIn, minAmountOut, receiver);

// dreUSDManager._mintFromFiatUsd
```

```
if (m.receiver == address(0)) revert ZeroAddress();
IdreUSD(dreUSD).mint(m.receiver, dreUSDAmount);
```

The impact is mostly an inconsistent compliance behavior across mint, transfer and withdrawal flows.

Recommendation: Use one compliance primitive and apply it uniformly before value movement in all user-facing flows.

Replace sanctions-only checks in withdrawal fills with `_validateAddress(currentOwner)` so sanctions and freeze are both enforced.

Add explicit manager/vault checks where missing: `_validateAddress(from)` in `mintFrom`, `_validateAddress(m.receiver)` in `_mintFromFiatUsd`, and `_validateAddress(receiver)` in `dreUSDs._withdraw`.

For withdrawal claim NFTs, enforce address validation on transfer by adding a compliance check in `dreWithdrawalNFT` transfer hooks (for non-mint/non-burn `from/to`) or explicitly making these NFTs non-transferable if that better matches product intent.

If any exceptions are intentional (for example allowing a frozen but non-sanctioned withdrawal recipient), document the policy explicitly and keep naming consistent so ”validation” means the same rule set everywhere.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `43cd857`, sanctions-only checks in `fillWithdrawal` and `fillExpressWithdrawals` were replaced with `_validateAddress(currentOwner)`, enforcing both freeze and sanctions via the canonical primitive.
2. In fix commit `baeade1`, manager entrypoints were hardened with missing validations, including `_validateAddress(from)` and `_validateAddress(receiver)` in `mintFrom`.
3. In final commit `3c0e338`, withdrawal NFTs enforce compliance on transfer/mint/burn paths via `dreWithdrawalNFT._update` address validation, removing the transfer-policy gap described in the finding.
4. The remaining paths noted in the finding (`_mintFromFiatUsd` receiver and `dreUSDs._withdraw` receiver) are enforced by `dreUSD`'s canonical `_update` validation during mint/transfer, so the effective compliance rule set is consistent even where checks are downstream rather than explicit pre-checks.

5.2.6 Cross-Chain token freeze can strand funds

Severity: Medium Risk

Context: dreUSD.sol#L146-L156

Description: When a dreUSD holder initiates a LayerZero cross-chain transfer, the operation is two-phase: tokens are burned on the source-chain via `_debit()`, and then minted on the destination-chain via `_credit()`. Both phases pass through `_update()` in `dreUSD.sol`, which calls `_validateAddress()`.

If the recipient address gets frozen or sanctioned between the source-chain burn and the destination-chain mint, the `_credit()` call reverts on the destination-chain. The tokens have already been destroyed on the source-chain but cannot be created on the destination. They are stranded in LayerZero's message queue with no automatic resolution, i.e. funds are in limbo.

Recommendation: Consider overriding LZ `_credit()` and have it directly call `ERC20Upgradeable._update(...)` for bypassing address validation in the overridden `_update(...)` during crediting. The frozen address then receives the tokens but is unable to move them because any subsequent `transfer()` or `send()` still calls `_validateAddress()`, which should revert. The funds are effectively quarantined in place rather than stranded in the LZ messaging layer.

Dre Labs: Fixed in commit 7fba78b3.

Spearbit: Fixed by overriding LZ `_credit()` in `dreUSD` and `dreShareOFT`. For failing transactions in `dreShareOFTAdapter`, funds are transferred to a multisig for later manual transfers to receivers.

5.2.7 Missing oracle price deviation check exposes protocol to depeg losses

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `dreUSDOracle` contract has had its price deviation checks intentionally removed, noting that callers should rely on `minAmountOut` slippage parameters instead. Slippage protection and oracle deviation bounds serve fundamentally different purposes and protect different parties. Slippage checks protect the *user* from receiving too little, whereas oracle deviation checks protect the *protocol* from disbursing too much. Without deviation bounds, a stablecoin depeg event - which has occurred multiple times historically for USDC - allows users to extract value from the treasury/vault with no protocol-level circuit breaker.

- User-side Positive Slippage: Case 1

  `_mintDreUSD` calls `oracle.getUsdValue(asset, amountIn)` to determine how much dreUSD to mint for a stablecoin deposit. If the oracle reports a price above $1.00, the USD value is over-estimated and the user receives more dreUSD than the deposit is worth. The `minAmountOut` check in `_transferAndMint` does not catch this because the user *benefits* and they receive *more* than their minimum, not less.

  Result: The treasury is left under-collateralised between USDC:DreUSD thus the assumed peg of 1 usd to 1 dreUSD fails.

- User-side Positive Slippage: Case 2

  `requestWithdrawal` and `requestExpressWithdrawal` call `oracle.getTokenAmount(usdc, dreUSDAmount)` to convert dreUSD to USDC. The formula `tokenAmount = usdAmount / price` (line 163) means a lower reported price yields more USDC per dreUSD burned. At a $0.87 price, each $1 of dreUSD redeems for ~1.149 USDC. Again, the `minUsdcAmount` slippage check does not trigger because the user receives more than their floor.

  Result: In this case the dreUSD vault is distributing the correct amount of USDC since the depeg only affects the USD:USDC conversion. However, the vault now burns more USDC than it holds, which means the vault may not have enough collateral to satisfy full vault withdrawals.

`minAmountOut` is a user-set, unidirectional floor it only reverts when the user would receive too little. It has no ability to revert when the protocol gives too much. A deviation check is a protocol-set, bidirectional bound that halts operations whenever the oracle price deviates beyond an acceptable range in either direction. These are complementary protections at different trust boundaries, not substitutes.

Worth mentioning that USDC depeg events are not a theoretical attack vector, this has happened several times in the past. For reference:

1. 2024: Flash Crash on Binance caused USDC to drop to $0.74 before quickly returning to $1.
2. March 10-11, 2023: USDC fell below 87 cents after Circle revealed it has nearly 8% of its $40 billion in reserves tied up at the collapsed lender Silicon Valley Bank.
3. 2022: USDC fell just below $0.99 because of the collapse of crypto hedge fund Three Arrows Capital.
4. October 2018: Tether FUD had USDC around $0.97.

Recommendation: Re-introduce a configurable per-token deviation threshold in `dreUSDOracle` (e.g., 2-5% from $1.00). When the Chainlink price falls outside this range, the oracle should revert, acting as a protocol-level circuit breaker independent of user-set slippage parameters.

Dre Labs: Fixed in PR 10.

Spearbit: Fixed by implementing default deviation checks on `getPriceDecimals()` function.

5.2.8 Users may receive lower than expected express withdrawal amounts due to unbounded fees applied after slippage check

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: DreUSD allows users to request an express withdrawal, which has a 6 hour fill target but with a 50 bps fee in USDC. This is a faster alternative to the standard withdrawal, which has no fees but may take more than 7 days.

However, there are two issues with its current implementation:

1. `updateExpressWithdrawal(...)` allows the protocol `WITHDRAWAL_CONFIG_ROLE` to set an unbounded `expressWithdrawalFeeBps` because the only sanity check applied is `feeBps > 10_000` to cap it to 100%. This allows the protocol to set an unreasonably high fee, even 100%.
2. `requestExpressWithdrawal(...)` applies a slippage check on `totalUsdcAmount` (as returned by the Chainlink oracle for the USDC value of the user's `dreUSDAmount`) against `minUsdcAmount` provided by the user, but *before* deducting `feeUsdc` in `_queueExpressWithdrawal(...)`.

These two issues may cause users to receive much lower than their expected `minUsdcAmount` USDC when express withdrawals are filled later by the protocol, which is effectively unexpected loss of user funds towards the protocol.

Recommendation: Consider:

1. Applying a reasonably upper bound guardrail to `expressWithdrawalFeeBps` in `updateExpressWithdrawal(...)`.
2. Applying the slippage check after deducting fees in `requestExpressWithdrawal(...)`.

Dre Labs: Fixed in commit 9100dc22.

Spearbit: Fixed by enforcing a `MAX_EXPRESS_WITHDRAWAL_FEE_BPS = 500` (5%) in `updateExpressWithdrawal(...)` but retaining the slippage check as-is.

5.2.9 Missing L2 sequencer uptime check may cause loss of funds due to stale prices

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The hub chain for DreUSD deployment is Base, which is a L2 chain based on Optimism. Base operates with a single centralized sequencer that has had two downtime incidents so far in September 2023 (~43 minutes) and August 2025 (~33 minutes). When a L2 sequencer goes down, advanced users can interact directly through L1 rollup contracts.

However, `dreUSDOracle` does not check if the L2 sequencer is up/down as recommended by Chainlink at their documentation's L2 Sequencer Feeds page. A staleness check using `updatedAt` is not sufficient on L2s and does not replace the need for a sequencer uptime check. When the sequencer is down:

1. The price feed on L2 stops updating, i.e. `updatedAt` timestamp freezes.
2. Sophisticated users can still submit transactions via L1.
3. These L1-submitted transactions execute on L2 before the sequencer's backlog when it comes back up.

Without this check, the protocol may use stale data from the oracle feeds for such L1-submitted transactions leading to loss of funds for users/protocol during minting or withdrawals. This could be actively exploited by advanced users via any price movement that occurs during the downtime or when the sequencer has just recovered.

Recommendation: Consider integrating Chainlink's suggested ”Sequencer Uptime Data Feed” to:

1. Detect sequencer downtime in real time.
2. Implement a grace period to prevent mass mints/withdrawals.

3. Temporarily pause operations during sequencer failures.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

5.2.10 Insufficient guardrails for `stalenessThresholds` may cause loss of funds due to stale prices

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: `dreUSDOracle` implements `stalenessThresholds` as a mapping of token address to staleness threshold in seconds. These are set by the `MODERATOR_ROLE` in `setOracle(...)`. They are used to check for stale oracle prices in `getUsdValue(...)` and `getTokenAmount(...)` by the reverting check `block.timestamp - updatedAt > threshold`.

However, the only guardrail implemented in `setOracle(...)` is a zero check `if (stalenessThreshold == 0) revert InvalidStalenessThreshold()`. This zero check alone is insufficient guardrails for `stalenessThreshold` and can allow stale values to be used:

1. A moderator could accidentally set a threshold, for e.g. `type(uint256).max`, effectively disabling staleness protection and causing loss of funds to users/protocol due to stale prices used during mints/withdrawals.
2. A moderator could accidentally set a very low threshold always causing `StaleOracleData` reverts and disrupting protocol operations.

Recommendation: Consider:

1. Configuring Chainlink chain+token specific heartbeat values to be used as the upper bound here. A reasonable value for many stablecoins on Base could be 86400 seconds given this is the time-based trigger for USDC/USDT/USDE/USDS/GHO and USD pairs on Base, for e.g. see USDC/USD on Base.
2. Implementing a reasonable lower bound for `stalenessThreshold`.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

5.2.11 Centralization risks can cause significant loss/lock of funds

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Critical aspects of the protocol are controlled by privileged actors who can cause loss/lock of user/protocol funds if they are malicious/compromised or misconfigured. Some key flows/impacts include:

1. DreUSD `GUARDIAN_ROLE` can freeze any arbitrary account.
2. DreUSD `DEFAULT_ADMIN_ROLE` can set any arbitrary `sanctionsList` to sanction any arbitrary account.
3. DreUSD `MANAGER_ROLE` can mint/burn arbitrary amounts of DreUSD to arbitrary addresses.
4. DreUSD `UPGRADER_ROLE` can upgrade the proxy to any arbitrary implementation. This is true for all protocol contracts that are upgradeable.
5. DreUSDManager `MODERATOR_ROLE` can set arbitrary custodian vaults, custodians (that are authorized to mint DreUSD against fiat), `dailyFiatMintCapUsd`, `dreRewardsDistributor` and `allowed` stablecoin tokens.
6. DreUSDManager `WITHDRAWAL_CONFIG_ROLE` can set arbitrary `expressWithdrawalMaxLimit`, `expressWithdrawalFeeBps`, `withdrawalWaitingTime` and `withdrawalVaultAdapter`.
7. DreUSDManager `KEEPER_ROLE` can arbitrarily mint DreUSD and rewards with the assumption that an equivalent Fiat USD has been deposited but without any proof-of-reserves.

8. DreUSDManager `EXPRESS_OPERATOR_ROLE` can include/exclude arbitrary Express Withdrawal NFTs to fill/deny requested withdrawals.
9. DreUSDManager `TREASURY_ROLE` can include/exclude arbitrary Withdrawal NFTs to fill/deny requested withdrawals.
10. DreUSDManager `PAUSE_ROLE` can arbitrarily pause all mints and withdrawal requests.
11. DreUSDOracle `MODERATOR_ROLE` can set arbitrary `oracles` and `stalenessThresholds`.
12. DreUSDs `PAUSE_ROLE` can arbitrarily pause all vault deposits and withdrawals.
13. DreWithdrawalNFT `MINTER_ROLE` and `BURNER_ROLE` can mint/burn arbitrary withdrawal NTFs.
14. DreAaveAdapter `WITHDRAWER_ROLE` can withdraw arbitrary amounts from the Aave vault meant for filling standard/long queue positions.
15. The `DEFAULT_ADMIN_ROLE` across all contracts has full administrative control to arbitrary grant roles.

Recommendation: Consider:

1. Enforcing separation of privilege across roles where they are backed by different accounts/addresses to minimize single points of failure.
2. Enforcing highest levels of operational security appropriately to accounts/roles by using hardware wallets, multisigs with reasonably high signature thresholds and other infrastructure security measures to avoid, minimize, mitigate, compartmentalize and contain risks.
3. Enforcing appropriate timelocks to privileged actions affecting protocol/users.
4. Limiting the impact of malicious/compromised or misconfigured roles by enforcing in-protocol guardrails.
5. Documenting all the roles and their responsibilities in `ROLES.md`.
6. Warning users appropriately about relevant risks.
7. Preparing an incident response playbook for exploit scenarios.
8. Practicing war game exercises for incident response preparedness.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.2.12 Unfillable express NFTs can lock capacity

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Express requests reduce `expressWithdrawalAvailable` immediately and burn user `dreUSD`, but there is no cancellation or forced-resolution path if the position becomes permanently unfillable. Filling requires owner compliance checks and the NFT transfer and burn path also enforces owner compliance. If a position owner is frozen or sanctioned after requesting express withdrawal, the position can become non-transferable and non-fillable indefinitely while still consuming express capacity.

```
// request path
expressWithdrawalAvailable -= usdcAmount;

// fill path
address currentOwner = IERC721(expressWithdrawalNFT).ownerOf(tokenId);
_validateAddress(currentOwner);

// NFT update path
if (owner != address(0)) {
    _validateAddress(owner);
}
```

This can permanently leak express capacity and lock user funds for affected positions because the burned `dreUSD` has already left user balance and no unwind mechanism exists in the manager.

Recommendation: Add an explicit resolution path for stale or unfillable express positions. The resolution flow should burn the position, restore express availability, and apply policy-defined settlement for the underlying value. If compliance policy forbids returning value to the owner, route settlement to a designated custodial address while preserving protocol accounting.

Dre Labs: Acknowledged, this is solved by a b2b contract between us and our partner.

Spearbit: Acknowledged.

5.2.13 Express limit update desyncs capacity

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The function `updateExpressWithdrawal` updates `expressWithdrawalMaxLimit` and only clamps `expressWithdrawalAvailable` downward when it is above the new maximum. This creates inconsistent capacity semantics because the function does not preserve outstanding utilization when limits are changed. When the limit is increased, available capacity does not increase accordingly even though the configured maximum is higher. When the limit is decreased below already committed utilization, the new maximum is no longer a true cap on effective outstanding exposure.

```
function updateExpressWithdrawal(
    uint256 maxLimit,
    uint256 feeBps,
    address feeRecipient
) external onlyRole(WITHDRAWAL_CONFIG_ROLE) {
//...
    uint256 oldLimit = expressWithdrawalMaxLimit;
    expressWithdrawalMaxLimit = maxLimit;
    if (expressWithdrawalAvailable > maxLimit) {
        expressWithdrawalAvailable = maxLimit;
    }
//...
}
```

This desynchronization also propagates into repayment logic because `payExpressDebt` computes `limitHeadroom` from `expressWithdrawalMaxLimit - expressWithdrawalAvailable`. Once the relation between outstanding utilization and available capacity is broken, repayment gating can become unintuitive and fragile operationally.

```
uint256 limitHeadroom =
    expressWithdrawalMaxLimit > expressWithdrawalAvailable
        ? expressWithdrawalMaxLimit - expressWithdrawalAvailable
        : 0;
```

Recommendation: Preserve utilization explicitly during limit updates. Compute `outstanding` from pre-update state, require the new limit to be compatible with that outstanding amount, and derive a consistent post-update available value from the new limit and preserved utilization.

```
uint256 oldLimit = expressWithdrawalMaxLimit;
uint256 oldAvailable = expressWithdrawalAvailable;
uint256 outstanding = oldLimit > oldAvailable ? oldLimit - oldAvailable : 0;
require(maxLimit >= outstanding, "new limit below outstanding");
expressWithdrawalMaxLimit = maxLimit;
expressWithdrawalAvailable = maxLimit - outstanding;
```

Dre Labs: Fixed in commit 681a6008.

Spearbit: Fix verified.

5.3 Low Risk

5.3.1 Express `minUsdcAmount` uses uncapped quote

Severity: Low Risk

Context: dreUSDManager.sol#L454-L476

Description: `requestExpressWithdrawal(dreUSDAmount,minUsdcAmount,deadline)` uses `minUsdcAmount` as a slippage check against the oracle quote for the full requested `dreUSDAmount`, but the function can later cap the express amount to the global `expressWithdrawalAvailable` and burn only a proportional amount of `dreUSD`. As a result, the check does not guarantee a minimum `USDC` amount for the express withdrawal position that is actually created.

In practice, a caller can pass a `minUsdcAmount` that is satisfied by the uncapped oracle quote, yet still receive a much smaller express withdrawal because the global express capacity is lower. This is surprising for integrators that interpret `minUsdcAmount` as "minimum USDC I will get out of this call". The minted express withdrawal `NFT` stores the net `USDC` amount after applying `expressWithdrawalFeeBps`, which further reduces what the user ultimately receives.

```
// dreUSDManager
uint256 totalUsdcAmount = IDreUSDOracle(oracle).getTokenAmount(usdc, dreUSDAmount);
if (totalUsdcAmount < minUsdcAmount) revert SlippageExceeded(minUsdcAmount, totalUsdcAmount);

uint256 dreUSDNeeded = dreUSDAmount;
uint256 expressUsdcAmount = totalUsdcAmount;
if (expressUsdcAmount > expressWithdrawalAvailable) {
    expressUsdcAmount = expressWithdrawalAvailable;
    dreUSDNeeded = dreUSDAmount * expressUsdcAmount / totalUsdcAmount;
}

IdreUSD(dreUSD).burn(msg.sender, dreUSDNeeded);
expressTokenId = _queueExpressWithdrawal(msg.sender, expressUsdcAmount);
```

```
// dreUSDManager
uint256 feeUsdc = (usdcAmount * expressWithdrawalFeeBps) / 10_000;
uint256 userReceivesUsdc = usdcAmount - feeUsdc;
tokenId = IWithdrawalNFT(expressWithdrawalNFT).mint(user, userReceivesUsdc);
```

Recommendation: Decide and enforce the intended meaning of `minUsdcAmount`. If it is intended to be a minimum amount the user will receive for this specific express withdrawal request, move the check to after capping (and after fee calculation) and compare against the actual amount stored in the express withdrawal position. If it is intended to be a minimum quote for the full requested `dreUSDAmount`, rename it to reflect that meaning and document that the function may partially fill the request based on global capacity and fees.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `a256b85`, the express-withdrawal slippage check was moved off the uncapped oracle quote path and into the post-cap/post-fee flow.
2. In final commit `3c0e338`, requestExpressWithdrawal routes to `_queueExpressWithdrawal(..., minUsdcAmount)`, which checks `userReceivesUsdc` against `minUsdcAmount` and reverts with `SlippageExceeded` if `userReceivesUsdc` is lower.
3. Regression tests for both capped-limit and post-fee slippage behavior are present in 3c0e338 (including `test_RequestExpressWithdrawal_RevertIf_SlippageExceeded_AtExpressLimit`).

5.3.2 Fiat mint sig not purpose bound

Severity: Low Risk

Context: dreUSDManager.sol#L392-L410

Description: Fiat mints are authorized by a custodian signature over the `FiatMint` fields (`mintRef`, `receiver`, `usdAmount`, `validUntil`, `chainId`). Both `mintFromUsd(m, sig)` and `mintRewards(m, sig)` accept the same signed payload and both call `_mintFromFiatUsd(m, sig)` to verify the signature, mark `mintRef` as used, and mint `dreUSD` to `m.receiver`. The difference is that `mintRewards` additionally calls `dreRewardsDistributor.addRewards()`, while `mintFromUsd` has no such side effect.

As a result, the signature is not bound to the intended on-chain action. Any keeper can take a signature that was operationally intended to fund rewards (for example, with `m.receiver` set to the rewards distributor) and submit it via `mintFromUsd` instead of `mintRewards`. This will successfully mint the tokens but will not start or update the vesting schedule, and the `mintRef` will be consumed so the same signed mint cannot later be replayed through `mintRewards` to trigger `addRewards()`.

A realistic exploitation scenario is a compromised or malicious keeper key that processes a "rewards top-up" mintRef using `mintFromUsd`. The new `dreUSD` ends up sitting in the rewards distributor contract balance, but it is not incorporated into `rewards` until a moderator later calls `addRewards()`. If operators do not notice promptly, users may observe a period of missing or delayed rewards despite funds being minted on-chain, which can trigger user complaints and operational incident response. Even if operators do notice, a malicious keeper can intentionally choose this path to decouple minting from vesting, creating an avoidable "stuck rewards until manual intervention" failure mode.

The same lack of an explicit action discriminator also reduces future safety: if additional custodian-signed actions are added later and reuse the same message fields and verification path, the system can reintroduce cross-function confusion unless each message is explicitly bound to a single purpose.

```
function mintFromUsd(FiatMint calldata m, bytes calldata custodianSig) external
↪  onlyRole(KEEPER_ROLE) {
    _mintFromFiatUsd(m, custodianSig);
}

function mintRewards(FiatMint calldata m, bytes calldata custodianSig) external
↪  onlyRole(KEEPER_ROLE) {
    if (m.receiver != dreRewardsDistributor) revert InvalidReceiver(m.receiver);
    _mintFromFiatUsd(m, custodianSig);
    IdreRewardsDistributor(dreRewardsDistributor).addRewards();
}
```

The impact is primarily an operational and accounting footgun: rewards can be minted into the distributor without starting vesting, requiring privileged follow-up to avoid "silent stuck rewards".

Recommendation: Consider binding custodian signatures to a specific purpose by including an explicit action discriminator in the signed data (for example, a `uint8 action` where `0 = fiatMint` and `1 = rewardsMint`) and have each entry point enforce the expected action before minting.

If changing the signing format is not feasible, enforce the separation at the contract level by preventing `mintFromUsd` from minting to the rewards distributor (revert when `m.receiver == dreRewardsDistributor`), so the only way to mint to the distributor is through `mintRewards`, which guarantees `addRewards()` is called.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `a256b85`, `mintFromUsd` added a receiver guard that reverts when `m.receiver == dreRewardsDistributor`.
2. In final commit `3c0e338`, that guard remains in place (`if (m.receiver == dreRewardsDistributor) revert InvalidReceiver(m.receiver);`), while mintRewards still requires `m.receiver == dreRewardsDistributor` and calls `addRewards()`.

3. A regression test for this separation is present in 3c0e338 (`test_MintFromUsd_RevertIf_ReceiverIsDreRewardsDistributor`).

5.3.3 Compose flow can get stuck on freeze/sanctions/pause

Severity: Low Risk

Context: dreOVaultComposer.sol#L11

Description: Composed second-hop execution and refunds depend on token transfers and vault operations that may enforce sanctions, freeze rules and pausing. If an address becomes frozen or sanctioned, or if the vault is paused, the hub-side compose execution can revert. The refund path can also fail if the refund recipient is blocked under the same rules. This can leave user funds credited to the composer or otherwise stranded until policy state changes or an operator intervenes.

One realistic failure mode is a spoke-to-hub "deposit assets, receive shares" flow using compose. A user sends `dreUSD` from a spoke chain to the hub `dreOVaultComposer` with a non-empty `composeMsg` that instructs the composer to call `VAULT.deposit(...)` and then transfer `dreUSDs` shares to the intended recipient. If, between the source send and the hub compose execution, the system is paused for incident response or the recipient becomes frozen or sanctioned, the composed execution can revert (either because the vault is paused or because the share transfer/mint is blocked by compliance checks).

When `handleCompose()` reverts, `lzCompose()` catches the error and attempts to refund back to the source chain. That refund is also implemented as an OFT send, so it depends on successful token crediting and any transfer hooks. If the refund recipient is blocked (or becomes blocked before the refund is executed), the refund attempt can revert as well, leaving the bridged amount stuck until an operator unpauses or unblocks addresses and retries execution.

```
// VaultComposerSync
try this.handleCompose{ value: msg.value }(_composeSender, composeFrom, composeMsg, amount) {
    emit Sent(_guid);
} catch (bytes memory _err) {
    if (bytes4(_err) == InsufficientMsgValue.selector) {
        assembly {
            revert(add(32, _err), mload(_err))
        }
    }
    _refund(_composeSender, _message, amount, tx.origin, msg.value);
    emit Refunded(_guid);
}
```

```
function _update(address from, address to, uint256 value) internal override {
    if (from != address(0)) _validateAddress(from);
    if (to != address(0)) _validateAddress(to);
    super._update(from, to, value);
}
```

Recommendation: Consider documenting an explicit runbook for stuck compose or refund flows and monitor composer balances and pending messages. If stranded funds are unacceptable, consider adding controlled recovery mechanisms (for example, a compliance escrow route) or additional pre-checks to prevent initiating transfers to blocked recipients.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

1. The provided fix commit `429d494` only changes `dreRewardsDistributor` naming (`vestPeriod` → `VEST_PERIOD`) and related tests/interfaces; it does not modify compose/refund flow contracts.
2. In `3c0e338`, compose/refund logic still relies on `VaultComposerSync.lzCompose()` try/catch with `_refund(...)` in the catch path, without a protective fallback if `_refund` itself fails.

3. In 3c0e338, dreUSDs still enforces whenNotPaused and sanctions/freeze validation in vault operations and share transfers (_deposit, _withdraw, _update), so compose execution can still revert during pause/compliance events.
4. In 3c0e338, dreOVaultComposer does not add a stuck-funds recovery path for failed token refunds under blocked-recipient conditions.

5.3.4 Express debt can accrue without payback addr

Severity: Low Risk

Context: dreUSDManager.sol#L581-L582

Description: Express withdrawal fills accrue expressFillerDebt that is expected to be repaid later by the Treasury via payExpressDebt(amount). However, the payback path requires expressPaybackAddress to be configured, while the express request and fill paths do not. Since expressPaybackAddress is not set in initialize, it is possible to enable and use express withdrawals (by setting only expressFeeRecipient) and begin accruing express debt even though the system cannot repay it.

This is a potential configuration flaw. If an operator fills express withdrawals while expressPaybackAddress is unset, the system will record debt but payExpressDebt will revert and expressWithdrawalAvailable cannot be restored via payback. This can wedge express capacity and leave the express filler unable to be repaid through the intended mechanism.

```
// dreUSDManager
function fillExpressWithdrawals(uint256[] calldata tokenIds) external
↪  onlyRole(EXPRESS_OPERATOR_ROLE) returns (...) {
//...
    expressFillerDebt += totalRequired;
//...
}
```

```
// dreUSDManager
function _paybackExpressFiller(uint256 amount) internal {
    if (amount == 0) revert ZeroAmount();
    if (expressPaybackAddress == address(0)) revert NoPaybackAddressSet();
//...
    IERC20(usdc).safeTransferFrom(msg.sender, expressPaybackAddress, amount);
//...
}
```

Recommendation: Require expressPaybackAddress to be configured before allowing express withdrawals to be used. A concrete approach is to add an expressPaybackAddress != address(0) check in the express request/fill path (for example in _queueExpressWithdrawal or fillExpressWithdrawals) so debt cannot accrue unless payback is possible. Alternatively, set expressPaybackAddress during initialize and make it part of the same configuration step as updateExpressWithdrawal.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit 2522177, dreUSDManager.initialize was changed to require _expressPaybackAddress as an initializer parameter and to revert on zero address.
2. In final commit 3c0e338, this requirement remains in place (if (_expressPaybackAddress == address(0)) revert ZeroAddress();) and expressPaybackAddress is set during initialization.
3. Regression tests in 3c0e338 cover this initialization invariant, including test_Initialize_RevertIf_ExpressPaybackAddressIsZeroAddress.

5.3.5 Composer clears OFT minAmountLD to zero

Severity: Low Risk

Context: dreOVaultComposer.sol#L11

Description: In `_depositAndSend()`, the composer uses `SendParam.minAmountLD` to enforce slippage on the vault conversion, then overwrites `minAmountLD` with zero before performing the OFT send. This disables OFT-level minimum-amount checks for the second hop. For small amounts, dust removal or rounding can result in a second-hop send of zero while leaving residual tokens trapped in the composer, and users cannot express a minimum bridged amount requirement for the OFT hop.

```
_assertSlippage(shareAmountReceived, _sendParam.minAmountLD);
_sendParam.amountLD = shareAmountReceived;
_sendParam.minAmountLD = 0;
_send(SHARE_OFT, _sendParam, _refundAddress, _msgValue);
```

Recommendation: Separate vault-conversion slippage from OFT slippage. For example, keep a dedicated `minVaultOut` for the vault conversion and preserve the user's `minAmountLD` for the OFT send, or enforce that amounts are above any OFT dust threshold before sending. Consider adding a controlled sweep mechanism for accumulating dust if it can occur.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `ee476f3`, `dreOVaultComposer` overrides `_depositAndSend` and `_redeemAndSend` to preserve user-provided `minAmountLD` for OFT send after vault slippage checks, instead of zeroing it out.
2. In final commit `3c0e338`, the composer still sets `_sendParam.minAmountLD = minAmountLD` (and no longer resets it to `0`) in both deposit and redeem paths.
3. Regression tests are present in `3c0e338` at `test/ovault/dreOVaultComposer.t.sol`, including `test_depositAndSend_PreservesMinAmountLD` and `test_depositAndSend_MinAmountLD_NotZero`.

5.3.6 AaveAdapter init does not validate aToken

Severity: Low Risk

Context: dreAaveAdapter.sol#L82-L84

Description: `dreAaveAdapter.initialize()` derives `aUsdc` by calling `getReserveData(usdc)` on the configured pool and storing `reserveData.aTokenAddress` without validation. If the pool does not support the configured asset or is misconfigured, this derived address can be `address(0)` (or otherwise not a valid token contract). Later calls that assume `aUsdc` is an ERC20, such as `getAvailableBalance()` and `withdraw()`, can revert or behave unexpectedly, potentially causing a denial of service for vault-based withdrawals.

```
IAaveV3Pool.ReserveData memory reserveData = IAaveV3Pool(_aavePool).getReserveData(_usdc);
aUsdc = reserveData.aTokenAddress;
```

Recommendation: Validate the derived `aToken` during initialization. At minimum, require `reserveData.aTokenAddress != address(0)`. For fail-fast hardening, also require `reserveData.aTokenAddress.code.length > 0`.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `f4285d0`, `dreAaveAdapter.initialize()` added validation that the derived `reserveData.aTokenAddress` is non-zero and has deployed code, reverting with `InvalidATokenAddress` otherwise.
2. In final commit `3c0e338`, these checks remain in place before `aUsdc` is assigned.
3. Regression tests are present in `3c0e338` (`test_Initialize_RevertIf_ZeroATokenAddress` and `test_Initialize_RevertIf_ATokenAddressHasNoCode`) and assert `InvalidATokenAddress` reverts.

5.3.7 WithdrawalNFT burn has unbounded loop

Severity: Low Risk

Context: dreWithdrawalNFT.sol#L97-L100

Description: When burning a `tokenId` far ahead of the current queue front, the burn logic iterates from `lastBurnedTokenId + 1` up to `tokenId - 1` to mark each intermediate id as skipped. This loop grows linearly with the size of the gap. At sufficiently large queue depths, burning an out-of-order id can exceed the block gas limit and revert.

```
// dreWithdrawalNFT
for (uint256 id = lastBurnedTokenId + 1; id < tokenId; id++) {
    _skippedTokenIds.add(id);
}
```

This creates a gas-based denial of service risk for operational flows that attempt out-of-order fills, and it can effectively force strict in-order processing once the queue becomes large.

Recommendation: Avoid O(n) bookkeeping for skipped ids. Track skipped ranges compactly (for example by storing intervals), cap the maximum out-of-order delta allowed per burn, or remove the skipped set and rely on off-chain enumeration of pending positions.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `54f3275`, `dreWithdrawalNFT.burn()` removed the gap-walking loop over `lastBurnedTokenId + 1 .. tokenId - 1` and the `_skippedTokenIds` set bookkeeping.
2. In final commit `3c0e338`, `burn()` updates `lastBurnedTokenId` in $\mathcal{O}(1)$ (`if (tokenId > lastBurnedTokenId) lastBurnedTokenId = tokenId;`) with no unbounded iteration.
3. Related interface/tests in `3c0e338` were updated to the new queue model (`getPositions`, `getTokensByIndexes`, and `test_GetPendingRange_OutOfOrderBurns`), confirming the loop-based skipped-ID mechanism is no longer used.

5.3.8 USDT enabling fee-on-transfer will lead to protocol loss

Severity: Low Risk

Context: dreUSDManager.sol#L28, dreUSDManager.sol#L832-L847

Description: DreUSD supports minting of dreUSD 1:1 with allowed stablecoins of USDC and USDT. However, USDT has a fee-on-transfer toggle that is currently disabled, but can be enabled anytime in future. But, `_transferAndMint(...)` assumes that the entire `amountIn` is transferred by the user into its custodian vault and mints an equivalent dreUSD. So if the actual USDT tokens received are `amountIn` minus fee, then internal accounting inflates balances leading to undercollateralization and eventual protocol loss.

Recommendation: Consider adding a before-after differential balance check to account only for received tokens.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

5.3.9 CEI violation in `_queueExpressWithdrawal` fee bypass via `_safeMint` reentrancy

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `expressWithdrawalFees[tokenId]` is set after the external call to `IWithdrawalNFT.mint()`, which uses `_safeMint` and triggers `onERC721Received` on the recipient. During this callback the fee mapping is still zero. A recipient contract that also holds `EXPRESS_OPERATOR_ROLE` could call `fillExpressWithdrawals`

(which lacks `nonReentrant`) to self-fill with zero fee. Practically unlikely since `EXPRESS_OPERATOR_ROLE` is a privileged role, and express fill time is intended to be 6 hours (though not enforced on-chain).

Recommendation: Set `expressWithdrawalFees[tokenId]` before the mint call, or add `nonReentrant` to `fillExpressWithdrawals`.

Spearbit: Fixed. The nonReentrant modifier prevents the attack. The underlying CEI ordering is mitigated by the re-entrancy guard.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified. The `nonReentrant` modifier prevents the attack. The underlying CEI ordering is mitigated by the re-entrancy guard.

5.3.10 Spoke Chain Share Token Missing Sanctions Enforcement

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `dreShareOFT` on spoke chains inherits from plain `OFT` without overriding `_update()` to enforce sanctions or freeze checks. On the hub chain, `dreUSDs._update()` validates both sender and recipient against the dreUSD sanctions and freeze list on every transfer, mint, and burn. The spoke chain representation has no equivalent check, allowing sanctioned addresses to freely transfer share tokens on spoke chains.

A sanctioned user who bridged shares to a spoke chain before being added to the sanctions list retains full transfer capability on that chain. They can transfer shares to a clean address on the spoke, which can then bridge back to the hub via the `dreShareOFTAdapter`, effectively circumventing the hub chain's compliance enforcement.

Recommendation: Override `_update()` in `dreShareOFT` to query a spoke-chain sanctions oracle or mirror the hub chain's sanctions list, ensuring consistent compliance enforcement across all chains.

Dre Labs: Fixed in PR 10.

Spearbit: Fixed. Both the spoke chain _update and _credit were overridden with the recommendation.

5.3.11 Dust Reward Addition Unfairly Resets Vesting Schedule, Delaying Existing Unvested Rewards

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `dreRewardsDistributor`, calling `addRewards()` with a trivially small amount of new dreUSD (as low as 1 wei) can trigger a full vesting schedule reset when more than 1 day has elapsed in the current period. This causes all remaining unvested rewards to be redistributed over a fresh 7-day window, delaying their availability to vault stakers. The issue requires `MODERATOR_ROLE` to exploit, but represents an unintended side effect of the extension math where dust additions produce outsized schedule changes.

When `addRewards()` is called, it first invokes `_claimVested()` which advances `cTs` to `block.timestamp` if any vested amount is claimable. After claiming, the function computes how much time the new rewards should add to the schedule at the current vesting rate:

```
uint256 rTs = newRewards * (eTs - cTs) / rewards;
```

With 18-decimal token amounts, if `newRewards` is dust (e.g. 1 wei) and `rewards` is any meaningful amount (e.g. 143e18), `rTs` rounds to 0. The resulting `newVestPeriod` equals `(eTs - cTs) + 0`, which is just the remaining time in the current vesting period.

The reset condition then checks:

```
if (newVestPeriod > vestPeriod || newVestPeriod < (vestPeriod - 1 days))
```

Since `_claimVested()` already advanced `cTs` to `block.timestamp`, the remaining time `eTs - cTs` will be less than `vestPeriod - 1 days` (6 days) whenever more than 1 day has passed in the current period. This

triggers the reset branch, setting `cTs = block.timestamp` and `eTs = block.timestamp + vestPeriod`, stretching all remaining rewards over a fresh 7-day window.

Example:

1. 1000e18 dreUSD rewards, vestPeriod = 7 days. Rate: 143e18/day.
2. At day 6: `_claimVested()` claims ~857e18. Remaining: 143e18 over 1 day.
3. Moderator adds 1 wei and calls `addRewards()`.
4. RTs = `1 * 86400 / 143e18` = 0.
5. NewVestPeriod = 1 day, which is < 6 days, so RESET triggers.
6. Result: 143e18 + 1 wei now vests over 7 days instead of 1 day.

The larger the remaining reward, the more the perceived unfairness and amount of times this can be repeated.

Recommendation: The recommended condition is not to extend the time period by vestedPeriod if `block.timestamp + vestedPeriod > rTs`.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.3.12 Insufficient guardrails in `setDailyFiatMintCap()` may be risky

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: DreUSD supports minting of dreUSD against offchain USD deposits by approved custodians. `setDailyFiatMintCap(...)` is a privileged function that sets a `dailyFiatMintCapUsd` used to enforce a daily fiat mint cap across all such mints. However, this function only applies a zero check on the set value. This may lead to two issues:

1. Prevents completely disabling fiat mints across custodians by setting `dailyFiatMintCapUsd` to 0. The comment `"// will revert automatically if dailyFiatMintCapUsd is 0"` in `_checkAndUpdateDailyFiatMint(...)` indicates that `setDailyFiatMintCap(...)` should allow `dailyFiatMintCapUsd == 0`.
2. Allows the Moderator to accidentally set an unreasonably high value, which could effectively allow infinite mints. For example, in October 2025, Paxos that issues PayPal's PYUSD stablecoin accidentally minted 300 trillion PYUSD tokens instead of what was supposed to be likely 300 million tokens. This was presumably because of the lack of a sanity check guardrail. Aave temporarily froze PYUSD trading as a precaution and the stablecoin briefly dipped about 0.5% from its $1 peg before recovering.

Recommendation: Consider:

1. Removing the zero check in `setDailyFiatMintCap(...)` or enforcing a reasonable lower bound.
2. Introducing a reasonable upper bound to prevent accidental infinite mints.

Dre Labs: Fixed in commit 05c1fc1b.

Spearbit: Fix verified.

5.3.13 Lack of fallback oracles is risky

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: DreUSD protocol currently relies only on Chainlink oracle to provide critical price feeds for USDC/USD, USDT/USD and other stablecoin pairs. The fairness and solvency of the protocol relies entirely on

the availability and freshness of these oracle price feeds. While the protocol has specified plans to implement fallback oracles from Redstone and Pyth, the current implementation relying only on Chainlink oracles is risky.

Recommendation: Consider implementing fallback oracles from Redstone and Pyth before deployment.

Dre Labs: Acknowledged. Intended.

Spearbit: Acknowledged.

5.3.14 Withdrawals of dust dreUSD amounts are not possible due to precision loss

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: DreUSD has a precision of 18 decimals. Stablecoin Chainlink price feeds denominated in USD generally have a precision of 8 decimals. `DreUSDOracle.getTokenAmount(...)` performs the below calculation to convert `dreUSDAmount` to `tokenAmount` during withdrawals:

```
// Convert dreUSDAmount from dreUSD decimals to price decimals
    uint256 usdAmountInPriceDecimals;
    if (dreUsdDecimals > priceDecimals) {
        usdAmountInPriceDecimals = dreUSDAmount / (10 ** (dreUsdDecimals - priceDecimals));
    } else {
        usdAmountInPriceDecimals = dreUSDAmount * (10 ** (priceDecimals - dreUsdDecimals));
    }

    // tokenAmount = usdAmountInPriceDecimals * 10^tokenDecimals / price
    // forge-lint: disable-next-line(unsafe-typecast)
    tokenAmount = (usdAmountInPriceDecimals * (10 ** tokenDecimals)) / uint256(answer);
```

The truncating division `usdAmountInPriceDecimals = dreUSDAmount / (10 ** (dreUsdDecimals - priceDecimals))` followed by the multiplication `tokenAmount = (usdAmountInPriceDecimals * (10 ** tokenDecimals)) / uint256(answer)` leads to precision loss whereby any dreUSD amount smaller than 10^10 truncates to zero. So for such withdrawals of dust dreUSD amounts, the user burns dreUSD but receives 0 tokens.

Recommendation: Consider:

1. Performing the multiplication before division to prevent precision loss or.
2. Enforcing a minimum withdrawal amount that avoids precision loss.

Dre Labs: Fixed in commit b10c5d13.

Spearbit: Fix verified.

5.3.15 DreUSDs vault does not prevent share transfers when paused

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: DreUSDs vault implements an emergency pausing mechanism, which is enforced on deposits and withdrawals. However, it does not prevent share transfers because the `_update(...)` override is missing `whenNotPaused` modifier.

During an emergency pause (e.g., responding to an exploit), shares can still be transferred to other addresses. The ability to transfer shares could allow fund extraction before any involved accounts are frozen/sanctioned.

Recommendation: Consider adding `whenNotPaused` modifier to `_update(...)`.

Dre Labs: Fixed in commit ac7ab3d6.

Spearbit: Fix verified.

5.3.16 Insufficient guardrails in `setVestPeriod()` may be risky

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `setVestPeriod(...)` is a privileged function that sets the `vestPeriod` used to enforce a linear vesting reward schedule. However, this function only applies a zero check on the set value. This may lead to two issues:

1. A tiny value will effectively vest all rewards immediately without achieving the linear vesting required. This will allow attackers to time their deposits and withdrawals around `addRewards()` such that it leads to yield dilution for long-term stakers.
2. An unreasonably large value will effectively never vest all rewards completely thus leading to yield loss for stakers.

Recommendation: Consider implementing reasonable lower and upper thresholds for `vestPeriod`.

Dre Labs: Fixed in commit f404a876.

Spearbit: Fix verified. `vestPeriod` was changed to a constant of `7 days`.

5.3.17 New `withdrawalWaitingTime` is applied retrospectively to pending withdrawal requests

Severity: Low Risk

Context: dreUSDManager.sol#L259-L263

Description: `withdrawalWaitingTime` is the "Minimum waiting time before withdrawal positions can be filled (default 7 days)". This value is used when fulfilling normal withdrawals via the `fillWithdrawal()` function.

```
// Skip if waiting time has not passed
if (block.timestamp < position.createdAt + withdrawalWaitingTime) revert NotReady();
```

But when this waiting period is updated using the `updateWithdrawal()` function, the `withdrawalWaitingTime` changes immediately and the new waiting period will be applied retrospectively to requests placed when the waiting period was different.

This happens because `fillWithdrawal()` uses the current value of `withdrawalWaitingTime` to determine if a withdrawal request can be fulfilled.

This will allow withdrawal positions to be filled earlier/ later than the expected time.

Recommendation: Consider caching the `withdrawalWaitingTime` (in nft Position struct) at request time to be used later at filling time.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.3.18 Permit Front-Run DOS on Permit-Based Mint Flows

Severity: Low Risk

Context: dreUSDManager.sol#L807-L821

Description: `_executePermit()` calls `IERC20Permit.permit()` unconditionally without first checking whether the spender already has sufficient allowance.

An attacker who observes a pending transaction in the mempool can extract the permit signature and front-run it by calling `permit()` directly on the token contract. The permit nonce is consumed, so when the victim's transaction executes, the internal `permit()` call reverts on the already-used nonce, reverting the entire mint transaction.

The three affected entry points are `mint (with permit)`, `mintFrom()`, and `mintAndStake()`. The non-permit mint overload is unaffected and serves as a fallback path using pre-approved allowance.

Recommendation: Wrap the `permit()` call in a try/catch or check `allowance(owner, spender) >= amount` before calling permit, so the function proceeds if allowance was already set by the front-runner.

Dre Labs: Fixed in commit e46ca9ca.

Spearbit: Fix verified.

5.3.19 Incorrect event emission in `_queueExpressWithdrawal()` and `_queueWithdrawal()`

Severity: Low Risk

Context: dreUSDManager.sol#L686, dreUSDManager.sol#L703

Description: In `_queueWithdrawal()`, the following event is emitted :

```
emit WithdrawalRequested(user, tokenId, usdcAmount, usdcAmount);
```

As per the event definition, the third parameter should be `dreUSDAmount` instead. In `_queueExpressWithdrawal()`, the following event is emitted :

```
emit ExpressWithdrawalRequested(user, tokenId, usdcAmount, userReceivesUsdc, feeUsdc);
```

As per the event definition, the third parameter here should be `dreUSDAmount`.

Recommendation: Consider modifying these lines to use the correct value as per event definition.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

5.3.20 Insufficient guardrails in `updateWithdrawal()` may be risky

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `updateWithdrawal(...)` is a privileged function that sets a `withdrawalWaitingTime` used to enforce a minimum withdrawal waiting time across all standard queue withdrawals between the request and fill times. While this is initialized to `7 days`, the setter does not enforce any guardrails. This may lead to two issues:

1. Setting to 0 allows immediate withdrawals or faster fills than even the express queue.
2. Setting to an unreasonably high value prevents withdrawals.

Recommendation: Consider reasonable enforcement of lower and upper bounds.

Dre Labs: Fixed in PR 10.

Spearbit: Fixed by enforcing a `minWaitingTime` and `maxWaitingTime` of `1 days` and `14 days` respectively.

5.3.21 Withdrawal fills are not prevented when paused

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Protocol implements an emergency pausing mechanism, which is enforced on DreUSD mints/withdrawals, DreUSDs vault deposits/withdrawals and DreRewardsDistributor `addRewards()` using the `whenNotPaused` modifier. However, this is not enforced on DreUSDManager `fillWithdrawal(...)`, `fillExpressWithdrawals(...)` and DreAaveAdapter `withdraw(...)` for withdrawals that have already been requested.

While `fillWithdrawal(...)` is restricted to `TREASURY_ROLE`, `fillExpressWithdrawals(...)` to `EXPRESS_OPERATOR_ROLE` and `withdraw(...)` to `WITHDRAWER_ROLE`, any fills triggered automatically (for e.g., because of out-of-sync automated offchain logic/scheduling) after an emergency pause will be filled and

funds will be transferred out of the protocol. This will reduce the expected impact of the emergency pause and may cause loss of funds.

Recommendation: Consider enforcing the emergency pausing mechanism on DreUSDManager `fillWithdrawal(...)`, and `fillExpressWithdrawals(...)`. Assuming DreUSDManager `fillWithdrawal(...)` is the only caller of DreAaveAdapter `withdraw(...)` (as currently implemented), `withdraw(...)` itself may not need the `whenNotPaused` modifier.

Dre Labs: Fixed in commit c0252d7b.

Spearbit: Fix verified.

5.3.22 `fillWithdrawal()` and `fillExpressWithdrawals()` can be DoS'ed by transferring withdrawal NFTs to sanctioned addresses

Severity: Low Risk

Context: dreUSDManager.sol#L513-L517, dreUSDManager.sol#L565-L569

Description: In `fillWithdrawal()` function, it is required that the current owner of the withdrawal position NFT is not sanctioned.

```
// Check owner is not sanctioned using canonical dreUSD sanctions list
address list = IdreUSD(dreUSD).sanctionsList();
if (list != address(0) && ISanctionsList(list).isSanctioned(currentOwner)) {
    revert SanctionedAddress(currentOwner);
}
```

But this quirk can be intentionally misused to revert these calls, and prevent/ delay withdrawals for the whole batch of tokenIDs.

This check can be used to DOS `fillWithdrawal()`/ `fillExpressWithdrawals()` because sanctions are not checked when transferring that NFT. This is the attack path :

- User X holds an NFT and is not sanctioned.
- They control one of the sanctioned addresses.
- They can purposefully transfer the NFT to that sanctioned address, which will DOS this batch of withdrawals.
- They can easily recover that NFT back as they control both the addresses.

Since sanctions are not checked when transferring the NFTs, a sanctioned user can move their NFT to a new address before their withdrawal request gets filled. This helps them bypass the sanction checks and complete the withdrawal.

Recommendation: Consider changing the logic in these functions to continue with next withdrawals in the batch, instead of reverting in case an NFT owner is found to be sanctioned.

Also sanction checks should be applicable when transferring withdrawal NFTs to further tighten restrictions.

Dre Labs: Fixed in commit 7525b8a2.

Spearbit: Fix verified. Worth mentioning that if theres a discrepancy between the santion list (say chainanalysis) and circles blocklist, then we would can end up reverting during transfer whilst not skipping.

Dre Labs: Acknowledged.

5.3.23 Large express withdrawals can exhaust available pool to block other users

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `requestExpressWithdrawal(...)` function allows any user to consume the entire `expressWithdrawalAvailable` pool in a single transaction. There is no per-user or per-request cap but the

amount is only capped to the current pool balance. A single user with sufficient dreUSD can drain the pool to zero, causing all subsequent express withdrawal requests to revert until the off-chain operational pipeline replenishes the pool via fill and payback, which can take hours or longer. This will force other withdrawals to take the slower standard queue.

The comment `// Express amount = min( totalUsdcAmount, expressWithdrawalAvailable, maxExpressAmount)` references a `maxExpressAmount` suggesting a per-request cap was intended but never implemented.

Recommendation: Consider implementing the missing per-request cap `maxExpressAmount` as noted in the comment.

Dre Labs: Acknowledged.

Spearbit: Acknowledged. There is apparently no requirement to enforce a per-request or per-user cap. This will be managed by adjusting `expressWithdrawalMaxLimit` and `expressWithdrawalAvailable` accordingly.

5.3.24 `mintAndStake()` lacks slippage protection on ERC4626 deposit shares

Severity: Low Risk

Context: dreUSDManager.sol#L386

Description: `mintAndStake` has a `minAmountOut` slippage check for the stablecoin to dreUSD conversion (enforced in `_transferAndMint()`). However, there is no slippage check on the second step: the ERC4626 `deposit()` call into dreUSDs.

The share exchange rate in dreUSDs depends on `totalAssets()`, which includes `vestedAmount()` from the rewards distributor. If `addRewards()` is called between the user signing the transaction and it being executed or if stealth deposits are made directly into the vault both will increase the current share price and result in less shares output for the user.

Recommendation: Consider adding slippage checks after the dreUSDs share price calculation during deposit, to ensure users can control what minimum amount of shares they wish to receive. The same should be added for assets redeemed using `withdraw()`.

Dre Labs: Fixed in commit e81e17bc.

Spearbit: Verified fixed. Virtual balance now prevents share price manipulation via direct deposits. Some inflation is still possible through yield but the effects are limited through linear vesting.

5.3.25 Global express debt can misroute payback

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Express withdrawal debt is accumulated globally, but repayment is sent to a single mutable address. The debt variable does not track which filler actually fronted funds and the repayment path always transfers to `expressPaybackAddress`. In a system with more than one express operator, this can misattribute repayment and redirect funds away from the filler that incurred the liability.

```
// in fillExpressWithdrawals
expressFillerDebt += totalRequired;

// in _paybackExpressFiller
IERC20(usdc).safeTransferFrom(msg.sender, expressPaybackAddress, amount);
```

Because `expressPaybackAddress` is configurable while debt may already exist, historical debt can be repaid to a newly configured recipient that did not provide liquidity for earlier fills. This is a correctness and fund-allocation risk in multi-operator deployments.

Recommendation: Either enforce a strict single-filler model at contract level or track debt per filler address and repay per-filler debt only. If a single recipient model is kept, prevent changing the payback address while debt is non-zero.

```
mapping(address => uint256) public fillerDebt;

// on fill
fillerDebt[msg.sender] += totalRequired;

// on payback
function payExpressDebtFor(address filler, uint256 amount) external onlyRole(TREASURY_ROLE) {
    require(amount <= fillerDebt[filler], "exceeds filler debt");
    fillerDebt[filler] -= amount;
    IERC20(usdc).safeTransferFrom(msg.sender, filler, amount);
}
```

Dre Labs: Acknowledged. We don't want to change the payback address. We need to have the option for our partner to update the receiving wallet address for debt (the fills of express withdrawals). Filler can be any wallet addreess of our operators that has this role.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 Cache storage reads in `withdraw()`

Severity: Gas Optimization

Context: dreAaveAdapter.sol#L92-L113

Description: The `dreAaveAdapter` `withdraw` flow pulls several configuration addresses from storage and uses them across multiple calls. Caching these values into locals makes the code slightly cheaper and easier to read, and it prevents accidental repeated `SLOAD`s and repeated interface casts if this function is extended in future changes.

The current `withdraw()` reads `aUsdc`, `vault`, `aavePool`, and `usdc` directly at call sites:

```
// Check vault has enough aUSDC and has given us allowance
uint256 available = getAvailableBalance();
if (available < amount) revert InsufficientBalance(available, amount);

// Transfer aTokens from vault to this contract
IERC20(aUsdc).safeTransferFrom(vault, address(this), amount);

// Withdraw from Aave (burns aTokens, sends USDC to recipient)
withdrawn = IAaveV3Pool(aavePool).withdraw(usdc, amount, to);
```

Reducing repeated storage reads keeps the withdraw path easier to audit and decreases the gas paid by users.

Recommendation: Cache the relevant storage addresses into locals once at the top of `withdraw()` and use those locals for subsequent calls. If you want the caching to actually remove repeated `SLOAD`s across the full flow, introduce an internal helper that takes the cached addresses as parameters and have both `withdraw()` and the public `getAvailableBalance()` reuse it.

```
address _aUsdc = aUsdc;
address _vault = vault;
address _pool = aavePool;
address _usdc = usdc;
```

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.4.2 Use `_init_unchained` variants in initializers to avoid redundant parent initialization

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: Across the codebase, all upgradeable contract initializers call the full `__X_init()` functions (e.g., `__AccessControl_init()`, `__Pausable_init()`, `__ERC20Permit_init()`) rather than their `__X_init_unchained()` counterparts. The `_init()` variants recursively initialize all parent contracts in the inheritance chain, which means shared ancestors like Initializable or ERC20Upgradeable get their initialization logic executed multiple times across sibling calls.

While the initializer modifier prevents actual double-initialization bugs, the redundant internal calls still waste gas at deployment. This pattern appears in `dreUSD`, `dreUSDManager`, `dreUSDs`, `dreRewardsDistributor`, `dreWithdrawalNFT`, `dreUSDOracle`, and `dreAaveAdapter`.

Recommendation: Consider replacing `__X_init()` calls with `__X_init_unchained()` in each initializer, ensuring that each parent's initialization is called exactly once in the correct C3-linearized order.

For example in `dreUSD.sol`: `__AccessControl_init_unchained()`, `__Ownable_init_unchained(defaultAdmin)`, `__ERC20Permit_init_unchained("dreUSD")`.

Dre Labs: Fixed in commit edd4eb9e.

Spearbit: Fix verified.

5.4.3 Use `ReentrancyGuardTransient` for gas savings

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: `dreUSDManager.sol` imports the standard `ReentrancyGuard` from OpenZeppelin, which uses persistent storage (SSTORE/SLOAD) for the reentrancy lock. Since the contract targets Solidity ^0.8.28 and EVM chains that support EIP-1153 transient storage, it should use `ReentrancyGuardTransient` instead. Transient storage (TSTORE/TLOAD) costs only 100 gas per operation compared to the 2900/2100 cold/warm costs of regular storage, saving significant gas on every nonReentrant function call. The lock is automatically cleared at the end of each transaction.

Recommendation: Consider using ReentrancyGuardTransient library by Openzeppelin.

Dre Labs: Fixed in commit 1ae09c5d.

Spearbit: Fix verified.

5.5 Informational

5.5.1 Roles require post-deploy grants

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Several contracts rely on `AccessControl` roles for core operations but do not grant some operational roles during `initialize()`. This creates a deployment and operations footgun where the system appears deployed but key functionality is unusable until additional `grantRole(...)` transactions are executed. If those post-deploy grants are missed or delayed, critical flows can be wedged and require governance or admin intervention to restore.

The roles documentation contains statements that imply operational roles are granted during initialization when they are not. For example, ROLES.md includes the following excerpts:

```
## dreUSDManager
- DEFAULT_ADMIN_ROLE: Full administrative control, can grant/revoke all other roles
  - Grants: All roles during initialization
```

```
## dreRewardsDistributor
- DEFAULT_ADMIN_ROLE: Full administrative control
  - Grants: All roles during initialization

## dreWithdrawalNFT
- DEFAULT_ADMIN_ROLE: Full administrative control
  - Grants: All roles during initialization
```

In `dreRewardsDistributor`, `addRewards()` and `setVestPeriod()` are gated by `MODERATOR_ROLE`, but `initialize()` grants only `DEFAULT_ADMIN_ROLE`, `UPGRADER_ROLE`, and `PAUSER_ROLE`. As a result, no account can start or update reward streaming until `MODERATOR_ROLE` is granted after deployment.

```
function initialize(address defaultAdmin) public initializer {
//...
    _grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
    _grantRole(UPGRADER_ROLE, defaultAdmin);
    _grantRole(PAUSER_ROLE, defaultAdmin);
}
```

In `dreAaveAdapter`, `withdraw()` is gated by `WITHDRAWER_ROLE`, but `initialize()` grants only `DEFAULT_ADMIN_ROLE` and `UPGRADER_ROLE`. If the intended withdrawer (for example, `dreUSDManager`) is not granted `WITHDRAWER_ROLE` post-deploy, vault-based withdrawals will revert.

```
_grantRole(DEFAULT_ADMIN_ROLE, admin);
_grantRole(UPGRADER_ROLE, admin);
```

In `dreWithdrawalNFT`, `mint()` and `burn()` are gated by `MINTER_ROLE` and `BURNER_ROLE`, but `initialize()` grants only `DEFAULT_ADMIN_ROLE` and `UPGRADER_ROLE`. These operational roles must be granted post-deploy to the manager or other system components that create and settle withdrawal positions.

```
_grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
_grantRole(UPGRADER_ROLE, defaultAdmin);
```

In `dreUSDManager`, `initialize()` grants `MODERATOR_ROLE`, `WITHDRAWAL_CONFIG_ROLE`, and `PAUSER_ROLE` to the default admin, but does not grant `KEEPER_ROLE`, `EXPRESS_OPERATOR_ROLE`, or `TREASURY_ROLE`. If operators expect these to be present immediately after initialization, fiat mints and withdrawal fills will be unusable until roles are granted.

The impact is operational fragility: a partial deployment can block reward distribution, block adapter-based fills, or block minting and settlement of withdrawal positions until roles are correctly configured.

Recommendation: Make role configuration explicit and difficult to miss. A concrete approach is to accept the intended role holders as initializer parameters and grant `MODERATOR_ROLE`, `WITHDRAWER_ROLE`, `MINTER_ROLE`, and `BURNER_ROLE` during initialization. If you prefer post-deploy grants, enforce the required role assignments in deployment automation with hard-failing post-deploy assertions that check `hasRole(...)` for every required role holder before the system is considered live.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `0fb77b1`, `dreUSDManager.initialize` was upgraded to accept explicit role addresses and grant `KEEPER_ROLE`, `EXPRESS_OPERATOR_ROLE`, and `TREASURY_ROLE` during initialization.
2. In fix commit `2b5d4b8`, `dreAaveAdapter.initialize` now grants `WITHDRAWER_ROLE` to the manager during initialization.
3. In fix commits `97b74fd` and `9661a3b`, setup automation grants cross-contract operational roles that remain post-deploy by design (`MODERATOR_ROLE` on dreRewardsDistributor; `MINTER_ROLE`/`BURNER_ROLE` on withdrawal NFTs).

4. In fix commit `db2c3ab` (and present in `3c0e338`), deployment flow enforces hard-failing post-deploy role assertions via `RoleGrantHelper.checkRolesGranted(...)`, preventing a live deployment with missing required roles.

5.5.2 Rewards distributor scripts miswire vault

Severity: Informational

Context: dreRewardsDistributor.sol#L161

Description: The rewards distributor contract transfers vested `dreUSD` rewards to an immutable vault address when `claimVested` is called, and the `dreUSDs` `ERC4626` vault relies on this by claiming vested rewards during deposits and withdrawals. This requires the rewards distributor's immutable vault address to be the `dreUSDs` contract address so the claimed `dreUSD` actually lands in the staking vault.

The standalone rewards distributor deployment script sets this immutable vault address from a pre-deploy `VAULT_ADDRESS` constant that is typically an `EOA` or multisig. If this script is used to deploy a production distributor, vested rewards will be routed to the wrong address and stakers will not receive yield as intended. This can also destabilize `dreUSDs` accounting, because `dreUSDs` `totalAssets` includes the distributor's vested amount but claimed rewards would not end up in the vault contract.

```
// dreRewardsDistributor
address public immutable vault;

function _claimVested() internal returns (uint256 vested) {
//...
    IERC20(dreUSD).safeTransfer(vault, vested);
//...
}
```

```
// DeployDreRewardsDistributor.run
address vault = Config.VAULT_ADDRESS;
dreRewardsDistributor deployedImpl = new dreRewardsDistributor(dreUSD, vault);
```

Separately, the allowance and setup scripts suggest approving the distributor to pull `dreUSD` from the `VAULT_ADDRESS`, but the current distributor implementation does not pull rewards via `transferFrom` and instead expects rewards to be transferred into the distributor before `addRewards` is called. This mismatch can lead to deployments where rewards never vest or are routed incorrectly.

```
// SetupDreSystem
IERC20(dreUSDAddress).approve(rewardsDistributor, Config.REWARDS_DISTRIBUTOR_APPROVAL_AMOUNT);
```

Recommendation: Deploy the rewards distributor with the `dreUSDs` contract address as its immutable vault parameter. Update scripts and docs to reflect the actual funding model: transfer `dreUSD` rewards into the distributor before calling `addRewards`, rather than relying on approvals. Consider renaming the `VAULT_ADDRESS` constant to reduce ambiguity and add post-deploy assertions that the distributor vault address equals the deployed `dreUSDs` address.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit bc55091, `DeployDreRewardsDistributor.s.sol` was changed to use `Config.DREUSDS_ADDRESS` (not `Config.VAULT_ADDRESS`) as the immutable `vault` constructor argument for the distributor.
2. In final commit `3c0e338`, that deploy script still requires and passes `DREUSDS_ADDRESS` as the distributor vault address.
3. The outdated approval-based funding path was removed: `script/rewardsDistributor/AllowSpendTokens.s.sol` is absent in 3c0e338, and setup guidance now states to transfer `dreUSD` into the distributor and call `addRewards()` (no approval flow).

5.5.3 `DeployDreSystem` deploys `ShareOFT` on hub

Severity: Informational

Context: deployShareOFT.s.sol#L19-L23

Description: The `OVault` design expects the `ShareOFT` contract to exist on spoke chains, while the hub chain uses a `ShareOFTAdapter` that wraps the real vault share token. The standalone `ShareOFT` deployment script enforces this by reverting on Base (hub chain).

The combined deployment script deploys `ShareOFT` unconditionally and only conditionally deploys the hub-only components. As a result, running the combined deployment on Base can deploy both a `ShareOFT` and a `ShareOFTAdapter` on the hub chain. This increases the risk of operational wiring mistakes, such as configuring peers or enforced options against the wrong `OApp` address, which can break cross-chain share transfers or strand funds.

```
// DeployShareOFT.run
require(
    block.chainid != 84532 && block.chainid != 8453,
    "ShareOFT should NOT be deployed on Base (hub chain). Use ShareOFTAdapter instead."
);
```

```
// DeployDreSystem.run
_deployShareOFT(factory, defaultAdmin);
if (block.chainid == 84532 || block.chainid == 8453) {
    _deployBaseComponents(token, defaultAdmin, factory);
}
```

Recommendation: Update the combined deployment flow to deploy `ShareOFT` only on non-hub chains. Add an explicit guard so the internal `ShareOFT` deploy helper cannot be invoked on Base. Consider adding post-deploy checks and logging that confirm the hub chain has only the adapter and composer, and that the spoke chains have only the `ShareOFT`.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `8b65641`, the `ShareOFT` hub-chain guard was moved into `_deployShareOFT(...)`, ensuring even internal calls cannot deploy `ShareOFT` on Base hub chains.
2. In final commit `3c0e338`, `DeployDreSystem.run()` branches by chain: hub chains deploy hub components, while spoke chains deploy ShareOFT only.
3. In final commit 3c0e338, `_verifyDeployment` adds explicit hub/spoke assertions, including a hard-fail check that ShareOFT is not deployed on hub and that adapter/composer are not deployed on spokes.

5.5.4 `msgType` 2 options miss `lzReceive` gas

Severity: Informational

Context: wireDreUSD.s.sol#L23

Description: The wiring scripts build enforced options for `SEND_AND_CALL` (`msgType` 2) using only an executor `lzCompose` option, without also including an executor `lzReceive` gas option. For example:

```
// script/ovault/wireDreUSD.s.sol
sendAndCallOptions: OptionsBuilder.newOptions().addExecutorLzComposeOption(0, 400_000, 0)
```

For `OFT` sends that include a non-empty `composeMsg`, `OFTCore` selects `msgType` 2 and calls `combineOptions(dstEid, msgType, extraOptions)`, meaning the `msgType` 2 enforced options are what gets applied for composed sends:

```
// lib/devtools/packages/oft-evm/contracts/OFTCore.sol
uint16 msgType = hasCompose ? SEND_AND_CALL : SEND;
options = combineOptions(_sendParam.dstEid, msgType, _sendParam.extraOptions);
```

`LayerZero`'s executor option parsing requires a non-zero `lzReceive` gas amount to be present in the final executor options blob; if no `lzReceive` option exists, it reverts with `Executor_ZeroLzReceiveGasProvided()`:

```
// lib/LayerZero-v2/.../ExecutorFeeLib.sol
if (lzReceiveGas == 0) revert Executor_ZeroLzReceiveGasProvided();
```

As a result, any composed sends that rely on enforced options (and do not include an explicit `lzReceive` option in caller-supplied `extraOptions`) can fail at quote time or execution time. This can brick `SEND_AND_CALL` flows (deposits/withdrawals via compose) until configuration is corrected.

Realistic Failure Example: Any client/UX that initiates a composed send but relies on enforced options (leaving `extraOptions` as empty/default type-3) will revert. For example, a frontend implementing a spoke → hub deposit via the composer:

```
SendParam memory p = SendParam({
    dstEid: hubEid,
    to: bytes32(uint256(uint160(Config.DRE_OVAULT_COMPOSER_ADDRESS))),
    amountLD: 100e18,
    minAmountLD: 100e18,
    extraOptions: OptionsBuilder.newOptions(), // relies on enforced options
    composeMsg: hex"01", // non-empty => msgType = 2 (SEND_AND_CALL)
    oftCmd: ""
});

// Reverts with Executor_ZeroLzReceiveGasProvided() because enforced msgType=2 options include
↪  lzCompose only.
MessagingFee memory fee = dreUSD(Config.DREUSD_ADDRESS).quoteSend(p, false);
```

In-Repo Impact: The current project scripts under `script/*.s.sol` do not hit this revert as written:

- `script/ovault/depositSpoke.s.sol` includes both `lzReceive` and `lzCompose` in `extraOptions` for the composed asset send.
- Other scripts set `composeMsg: ""` and therefore use `msgType` 1 (`SEND`).

However, the deployed enforced options are still unsafe for any external integration (frontend/SDK/custom scripts) that sends composed messages while relying on enforced options (i.e., omitting an explicit `lzReceive` option in `extraOptions`).

Recommendation: When building `msgType` 2 enforced options, include both an executor `lzReceive` option and the executor `lzCompose` option in the same type-3 options blob. For example, build `msgType` 2 options as `lzReceive` plus `lzCompose`, with gas values sized for the destination `OFT` `lzReceive` and destination composer `lzCompose` execution. Ensure scripts consistently apply these `msgType` 2 options on every route where compose is expected to work.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `8b65641`, both `script/ovault/wireDreUSD.s.sol` and `script/ovault/wireShareOFT.s.sol` were updated so `sendAndCallOptions` include both `addExecutorLzReceiveOption(...)` and `addExecutorLzComposeOption(...)`.
2. In final commit `3c0e338`, those scripts still configure `msgType` 2 enforced options with the combined `lzReceive` + `lzCompose` options blob on compose-enabled routes.
3. This directly prevents the `Executor_ZeroLzReceiveGasProvided()` failure mode for composed sends that rely on enforced options.

5.5.5 ULN confirmations use destination values

Severity: Informational

Context: wireShareOFT.s.sol#L19-L38

Description: The wiring base script sets `ULN` confirmations for a pathway in a way that makes confirmations effectively destination-centric and identical across all inbound sources for a given local chain. Outbound send config uses the remote chain's confirmations value, while inbound receive config uses the local chain's confirmations value for every `remoteEid`:

```
// script/ovault/wire.s.sol
_setSendConfig(local, remoteEid, remote.confirmations);
_setReceiveConfig(local, remoteEid); // uses local.confirmations internally
```

In `ULN`, confirmations is used as a block confirmation delay before `DVNs` relay and sign, and as the minimum confirmations threshold when verifying `DVN` submissions on the receive side:

```
// lib/LayerZero-v2/.../ILayerZeroDVN.sol
// _confirmations - block confirmation delay before relaying blocks
```

Because confirmations is fundamentally a property of the source chain's finality and reorg tolerance, the receive-side confirmation requirement for messages coming from a given `srcEid` should be keyed to that source chain. With the current wiring logic, a chain will apply the same confirmation requirement to all inbound sources (local.confirmations), and will also set outbound send confirmations based on the destination entry, which can invert intended security and latency tradeoffs (for example, requiring few confirmations for a reorg-prone source, or many confirmations for a stable source).

For example, `wireShareOFT` sets Sepolia to `confirmations = 15` and Base Sepolia to `confirmations = 1`:

```
// wireShareOFT
chains[0] = OFTWireBase.ChainConfig({
    chainId: 11155111, // Ethereum Sepolia
    oapp: Config.DRE_SHARE_OFT_ADDRESS,
    confirmations: 15,
    sendOptions: OptionsBuilder.newOptions().addExecutorLzReceiveOption(80_000, 0),
    // enable compose for spoke -> hub share sends
    sendAndCallOptions: OptionsBuilder.newOptions().addExecutorLzComposeOption(0, 300_000, 0)
});

chains[1] = OFTWireBase.ChainConfig({
    chainId: 84532,  // Base Sepolia
    oapp: Config.DRE_SHARE_OFT_ADAPTER_ADDRESS,
    confirmations: 1,
    sendOptions: OptionsBuilder.newOptions().addExecutorLzReceiveOption(80_000, 0),
    sendAndCallOptions: ""
});
```

With the current wiring base, this produces a direction swap:

```
If run on Sepolia (11155111):
send(Sepolia -> Base) uses remote.confirmations = 1
receive(Base -> Sepolia) uses local.confirmations = 15

If run on Base Sepolia (84532):
send(Base -> Sepolia) uses remote.confirmations = 15
receive(Sepolia -> Base) uses local.confirmations = 1
```

Even though confirmations should be chosen based on the source chain for each direction, the current wiring effectively applies them based on the destination entry (and sets one inbound confirmations value for all sources to a given chain).

Recommendation: Treat confirmations as a source-chain parameter. When configuring local to remote outbound send config, use the local chain's confirmations value (since local is the source). When configuring remote to local inbound receive config, use the remote chain's confirmations value (since remote is the source). If you need destination-specific security policies, store confirmation requirements per remote `EID` (per pathway) rather than a single confirmations number per chain.

Dre Labs: Fixed in PR 10.

Spearbit: Fix verified.

1. In fix commit `621af77`, `wire.s.sol` changed send config wiring to use `local.confirmations` for local → remote (source-side) configuration.
2. In the same fix, receive config was changed to accept an explicit confirmations parameter and is now called with `remote.confirmations` for remote → local (source-side) configuration.
3. In final commit `3c0e338`, `_configurePathway` retains this corrected mapping (`_setSendConfig(..., local.confirmations)` and `_setReceiveConfig(..., remote.confirmations)`), aligning confirmations with source-chain finality assumptions.

5.5.6 Overloading critical roles is risky

Severity: Informational

Context: dreAaveAdapter.sol#L86-L87, dreRewardsDistributor.sol#L75-L77, dreUSDManager.sol#L185-L189, dreUSDOracle.sol#L48-L50, dreUSD.sol#L57-L59, dreUSDs.sol#L54-L56, dreWithdrawalNFT.sol#L59-L60

Description: Overloading roles/addresses violates the security principle of "Separation of Privileges", where it's recommended to separate out the roles and their backing privileged actors/addresses.

The `defaultAdmin` address is initialized to all the critical roles across the protocol. For example, in `dreUSDManager.initialize(...)`:

```
_grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
_grantRole(UPGRADER_ROLE, defaultAdmin);
_grantRole(MODERATOR_ROLE, defaultAdmin);
_grantRole(WITHDRAWAL_CONFIG_ROLE, defaultAdmin);
_grantRole(PAUSER_ROLE, defaultAdmin);
```

This violates the intended goal of achieving separation of privileges via role-based access control. Even though all these roles may be the platform-side responsibility, separating them out and their backing addresses will prevent all of them from being exploited if one of them is compromised.

Recommendation:

1. Consider initializing different roles to different addresses. Review roles/addresses across all contracts for similar concerns.
2. Review the opsec level required for different addresses based on the criticality of their assigned roles.

Dre Labs: Fixed in commit 00068a39.

Spearbit: Fix verified.

5.5.7 Unpause functionality controlled by the same role as pause may be risky

Severity: Informational

Context: dreRewardsDistributor.sol#L83-L92, dreRewardsDistributor.sol#L142-L144, dreUSDManager.sol#L288-L294, dreUSDManager.sol#L415-L419, dreUSDManager.sol#L968-L977, dreUSDs.sol#L88-L97, dreUSDs.sol#L112-L116, dreUSDs.sol#L121-L131

Description: The `unpause()` functions across the protocol are controlled by the same `PAUSER_ROLE` as their `pause()` functions. This means the account granted the capability to pause critical contract functions can also

unpause them. In security-sensitive scenarios, unpausing is considered a higher-privilege operation because it restores the ability to perform the critical functions, potentially impacting the system's security intentions for pausing.

Critical functions including DreUSD mint/withdrawals, DreUSDs vault deposits/withdraws and `dreRewardsDistributor.claimVested()` have the `whenNotPaused` modifier.

Recommendation: Consider separating the roles for pause and unpause functionalities. A higher-privileged role (with higher opsec for its backing address) should control the unpause functionality. This approach ensures stricter control over resuming paused critical functions.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.8 ERC-4626 Non-Compliance: max* View Functions Ignore Paused and Sanctioned States

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `dreUSDs` does not override `maxDeposit`, `maxMint`, `maxWithdraw`, or `maxRedeem`. The inherited OZ defaults return `type(uint256).max` for deposit/mint and the user's full balance for withdraw/redeem. These values remain non-zero when the contract is paused (operations revert in `_deposit` / `_withdraw` via `whenNotPaused`) and for sanctioned/frozen users (blocked by `_update` calling `_validateAddress`). Per the ERC-4626 spec: "MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0." Integrators relying on these view functions to determine deposit/withdrawal capacity will receive incorrect values.

Recommendation: Override `maxDeposit`, `maxMint`, `maxWithdraw`, and `maxRedeem` to return 0 when paused and for sanctioned/frozen addresses.

Dre Labs: Fixed in PR 10.

Spearbit: Verified fix. Added necessary overrides to the max* functions.

5.5.9 `DailyFiatMintCapUsd` Not Initialized Blocking Minting Until `MODERATOR_ROLE` Configures Cap

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `dailyFiatMintCapUsd` is not set in `initialize` and defaults to 0. Since `_checkAndUpdateDailyFiatMint` reverts when `newTotal > dailyFiatMintCapUsd` (any non-zero amount exceeds 0), both `mintFromUsd` and `mintRewards` are effectively blocked until `setDailyFiatMintCap` is called by `MODERATOR_ROLE`.

Recommendation: If this is intentional as a safety default, it should be documented. If not, the cap should be set in `initialize`.

Dre Labs: Acknowledged.

Spearbit: Acknowledged. Added comment to clarify intentional behaviour.

5.5.10 Sequential nonces are suboptimal

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `mintFrom(...)` allows minting of dreUSD by transferring stablecoins from a different `from` address (instead of `msg.sender`) who is required to have signed a permit and an explicit `authorizeSig` for this transfer. Replay attacks are currently prevented by using a per-address nonce that is incremented on each `mintFrom(...)`.

However, these sequential nonces require signers to keep track of the last used nonce. If a nonce is missed or a transaction is delayed, subsequent transactions are impacted. This can be made more flexible using a `mapping(uint256 → bool)` of supplied nonces and marking them as used appropriately.

This approach will also allow a simpler cancel signature ability to recover from an incorrect/accidental `mintFrom` signature by marking a specific nonce as used.

Recommendation: Consider supporting a non-sequential nonce usage and implementing a cancellation capability.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.11 Documentation Has Several Inconsistencies With Implementation

Severity: Informational

Context: dreUSDManager.sol#L446

Description: Several function signatures and behaviors documented in ARCHITECTURE.md do not match the implemented code:

1. `RequestExpressWithdrawal` is documented as returning (uint256 expressTokenId, uint256 withdrawalTokenId) with automatic splitting between express and long queues. The code returns only uint256 expressTokenId - there is no long-queue fallback. If express capacity is insufficient, the code caps the express amount and only burns proportional dreUSD, leaving the remainder with the user.
2. `RequestExpressWithdrawal` is documented with a `maxExpressAmount` parameter that does not exist in the code. The actual signature is `requestExpressWithdrawal(dreUSDAmount, minUsdcAmount, deadline)`.
3. The permit variant of mint is documented as mint(asset, amountIn, receiver, minAmountOut, deadline, permitSig) with a receiver parameter. The code has no receiver parameter - it always mints to msg.sender.
4. MintAndStake is documented with a minSharesOut parameter implying slippage protection on vault shares. received. The code parameter is minAmountOut which only protects the intermediate dreUSD minting amount.

Additionally Natspec contains the following errors:

1. NatSpec says _cap is ”in USD (6 decimals)” but the state variable comment (`dreUSDManager.sol#L72`) says ”2 decimals, e.g., 10_000_000_00 for 10M USD”. FiatMint.usdAmount also uses 2 decimals.
2. The comment on L464 in `dreUSDManager` states `Express amount = min(totalUsdcAmount, expressWithdrawalAvailable, maxExpressAmount)`. But the code only caps against `expressWithdrawalAvailable`. There is no `maxExpressAmount` variable or per-request limit - `expressWithdrawalMaxLimit` is a pool ceiling, not a per-withdrawal cap. The comment implies a three-way min that doesn't exist in the implementation.

Recommendation: If the current code behavior is intentional, update ARCHITECTURE.md to reflect the actual function signatures and withdrawal flow. If the documented behavior was the intended design, the code needs to be updated accordingly.

Dre Labs: Fixed in commit 30370b34.

Spearbit: Fixed with the exception of NatSpec issue (1) still references 6 decimals instead of 2.

5.5.12 `addRewards()` comment says 7 days

Severity: Informational

Context: dreRewardsDistributor.sol#L125

Description: In `dreRewardsDistributor`, the `addRewards()` function uses the configurable `vestPeriod` parameter to decide when to reset the vesting schedule. However, an inline comment describes the behavior as redistributing everything over ”7 days”, which is not necessarily true if `vestPeriod` is set to a different value.

The logic resets the vesting schedule to `block.timestamp + vestPeriod` when the computed `newVestPeriod` is out of bounds, but the comment implies a fixed 7-day redistribution:

```
// if higher than vestingPeriod or lower we redistribute everything over 7 days with new
↪  rewards rate
if (newVestPeriod > vestPeriod || newVestPeriod < (vestPeriod - 1 days)) {
    cTs = block.timestamp;
    eTs = block.timestamp + vestPeriod;
} else {
    // extend end timestamp by time equivalent to new rewards at current rate
    eTs = eTs + rTs;
}
```

This is a documentation inconsistency that can mislead reviewers and operators who rely on comments to understand the intended vesting behavior.

Recommendation: Update the comment to describe the behavior in terms of `vestPeriod` rather than a fixed 7-day duration, for example: "redistribute everything over `vestPeriod` with the new rewards rate" or "reset the schedule to now + `vestPeriod`".

Dre Labs: Fixed in commit 30370b34.

Spearbit: Fix verified.

1. In the code state included by commit `30370b3` (and retained in `3c0e338`), `dreRewardsDistributor` uses a fixed `VEST_PERIOD = 7 days` constant rather than a configurable `vestPeriod`.
2. In final commit `3c0e338`, `addRewards()` reset logic uses `VEST_PERIOD` directly (`eTs = block.timestamp + VEST_PERIOD`), so the inline "over 7 days" wording matches actual behavior.
3. The configurability path that made the comment potentially misleading (`setVestPeriod`) is absent in the final contract/interface, removing the original docs-vs-logic mismatch.

5.5.13 Deduplicate mint pre-checks

Severity: Informational

Context: dreUSDManager.sol#L287-L389

Description: The `dreUSDManager` contract exposes multiple mint entrypoints (`mint` with permit, `mint` without permit, `mintFrom`, and `mintAndStake`). These functions repeat the same basic validation block around amount, configuration, allowlist, and deadline checks. The logic is currently consistent, but the duplication increases the chance of drift in future edits, especially if one path is updated to add a new guard, tweak an error, or change ordering while another path is missed.

For example, both `mint` overloads independently perform the same checks before calling `_transferAndMint`:

```
if (amountIn == 0) revert ZeroAmount();
if (custodianVault == address(0)) revert ZeroAddress();
if (!allowed[asset]) revert StablecoinNotAllowed(asset);
if (block.timestamp > deadline) revert OrderExpired(deadline, block.timestamp);
```

```
if (amount == 0) revert ZeroAmount();
if (custodianVault == address(0)) revert ZeroAddress();
if (!allowed[asset]) revert StablecoinNotAllowed(asset);
if (block.timestamp > deadline) revert OrderExpired(deadline, block.timestamp);
```

Similar checks also exist in `mintFrom` and `mintAndStake` alongside additional parameter validation. This is not a security issue by itself, but it is a maintenance risk. If the checks ever diverge, one mint path could end up accepting inputs that another path rejects (for example, allowing a disallowed asset, skipping an expired deadline guard, or behaving differently when `custodianVault` is unset).

Recommendation: Factor shared mint validations into a small internal helper and call it from each mint entrypoint. Keep entrypoint-specific checks (such as `from` and `receiver` non-zero, or address compliance validation) in the relevant function, but centralize the common "amount, config, allowlist, deadline" guards to reduce drift risk.

```
function _preMintChecks(address asset, uint256 amountIn, uint256 deadline) internal view {
    if (amountIn == 0) revert ZeroAmount();
    if (custodianVault == address(0)) revert ZeroAddress();
    if (!allowed[asset]) revert StablecoinNotAllowed(asset);
    if (block.timestamp > deadline) revert OrderExpired(deadline, block.timestamp);
}
```

Dre Labs: Fixed in commit 35ff500c.

Spearbit: Fix verified.

1. In fix commit `35ff500`, `dreUSDManager` introduced a shared internal helper `_preMintChecks(asset, amountIn, deadline)` containing the common amount/config/allowlist/deadline guards.
2. In final commit `3c0e338`, all mint entrypoints (`mint` with permit, `mint` without permit, `mintFrom`, and `mintAndStake`) call `_preMintChecks(...)` instead of duplicating those checks inline.
3. The centralized helper in `3c0e338` retains the expected guard set (`ZeroAmount`, `ZeroAddress` for unset custodian vault, `StablecoinNotAllowed`, and `OrderExpired`), reducing drift risk across mint paths.

5.5.14 Project contract names not `CapWords`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Multiple first-party contracts in this codebase use lower camelCase names starting with `dre...` rather than the typical Solidity `CapWords` convention for contract names. This is stylistic, but it is inconsistent with common Solidity conventions and can trigger linters or internal tooling that assumes contract names start with an uppercase letter.

For example, the codebase includes contract declarations like:

```
contract dreUSDManager is
contract dreRewardsDistributor is
contract dreUSDs is
contract dreUSD is
contract dreUSDOracle is
contract dreAaveAdapter is
contract dreWithdrawalNFT is
contract dreShareOFT is
contract dreShareOFTAdapter is
contract dreOVaultComposer is
contract dreTimelockController is TimelockBase
```

This is not a security issue, but it's and easy-to-implement update that is considered a best practice.

Recommendation: Decide on a single naming convention for contract types and apply it consistently. If you want to align with typical Solidity style, rename first-party contracts to CapWords (for example, `DreUSDManager`, `DreAaveAdapter`, `DreTimelockController`) and update all references (deployment scripts, tests, documentation, imports, and any off-chain tooling that keys on artifact names). If the current lower camelCase naming is intentional, add a targeted linter suppression or a short style note so the convention is explicit and does not generate repeated lint noise across the codebase.

Dre Labs: Fixed in commit 065cb473.

Spearbit: Fix verified.

1. In fix commit `065cb47`, `docs/STYLE.md` was added with an explicit "Contract naming (intentional)" policy stating first-party `dre...` lower camelCase contract names are intentional and should not be auto-converted to CapWords.
2. In fix commit `065cb47`, `foundry.toml` added a lint note and `mixed_case_exceptions = ["ERC", "URI", "dre"]`, making the intentional naming convention tooling-aware and reducing repeated lint noise.
3. In final commit `3c0e338`, the same convention remains documented and enforced (`docs/STYLE.md` present, `README.md` links the Style doc, and `foundry.toml` keeps the mixed-case exception for `dre`).

5.5.15 `getSkippedTokenIds` can use `values()`

Severity: Informational

Context: dreWithdrawalNFT.sol#L156-L166

Description: The `dreWithdrawalNFT` contract exposes `getSkippedTokenIds()` as a view helper that returns the contents of the `_skippedTokenIds` `EnumerableSet.UintSet`. The current implementation manually allocates an array and copies elements one-by-one using `.at(i)`. This is correct, but it is verbose and duplicates logic that OpenZeppelin's `EnumerableSet` already provides via a `values()` helper for `UintSet`.

Current implementation:

```
function getSkippedTokenIds() external view returns (uint256[] memory skippedTokenIds) {
    uint256 len = _skippedTokenIds.length();
    skippedTokenIds = new uint256[](len);
    for (uint256 i = 0; i < len; i++) {
        skippedTokenIds[i] = _skippedTokenIds.at(i);
    }
}
```

In this repository, the vendored OpenZeppelin `EnumerableSet` includes `values(UintSet)` which returns a `uint256[] memory` directly. Using it can make the getter shorter and clearer.

Recommendation: If you are comfortable with OpenZeppelin's `values()` implementation (which uses a memory-safe assembly cast), replace the manual materialization with `_skippedTokenIds.values()`. If you prefer to avoid the assembly cast, keep the current code but consider minor loop cleanups (for example caching `len` and using `unchecked` for the increment).

```
function getSkippedTokenIds() external view returns (uint256[] memory skippedTokenIds) {
    return _skippedTokenIds.values();
}
```

Dre Labs: Fixed in commit 54f32753.

Spearbit: Fix verified.

1. In fix commit `54f3275`, `dreWithdrawalNFT` removed `EnumerableSet` usage, removed `_skippedTokenIds`, and removed `getSkippedTokenIds()` (the manual `.length()/.at(i)` materialization loop no longer exists).
2. In fix commit `54f3275`, queue-read APIs were refactored to `getPositions(...)`, `getPendingRange()`, and `getTokensByIndexes(...)`, replacing the prior skipped-ID getter path.
3. In final commit `3c0e338`, `contracts/dreWithdrawalNFT.sol`, `contracts/interfaces/IWithdrawalNFT.sol`, and `test/dreWithdrawalNFT.t.sol` contain no `getSkippedTokenIds` references, confirming the original pattern was fully removed.

5.5.16 `lastBurnedTokenId` comment is misleading

Severity: Informational

Context: dreWithdrawalNFT.sol#L35-L36

Description: The `dreWithdrawalNFT` contract uses `lastBurnedTokenId` and `_skippedTokenIds` to track progression through withdrawal positions. The current doc comment for `lastBurnedTokenId` states that all token IDs from 1 through `lastBurnedTokenId` are filled consecutively. That description does not hold once `burn()` is called out of order.

Today, when a higher `tokenId` is burned before intermediate IDs, the contract advances `lastBurnedTokenId` to that higher value and records the intermediate IDs in `_skippedTokenIds`. This makes `lastBurnedTokenId` a high-water mark or frontier, not a guarantee that every smaller ID has been filled. The existing comment can therefore mislead reviewers and off-chain operators into assuming that all IDs up to `lastBurnedTokenId` are settled, which is not necessarily true.

The misleading comment appears directly on the state variable:

```
/// @notice Highest token ID filled in order (1..lastBurnedTokenId are filled consecutively)
uint256 public lastBurnedTokenId;
```

And the out-of-order burn path shows how gaps are created while still advancing the frontier:

```
} else if (tokenId > lastBurnedTokenId + 1) {
    // Burning out of order: mark [lastBurnedTokenId+1, tokenId-1] as skipped
    for (uint256 id = lastBurnedTokenId + 1; id < tokenId; id++) {
        _skippedTokenIds.add(id);
    }
    lastBurnedTokenId = tokenId; // Pending range becomes [tokenId+1, ...]
}
```

This is a documentation issue rather than a security issue, but it can cause operational mistakes if indexers, dashboards, or scripts interpret `lastBurnedTokenId` as implying a contiguous filled prefix rather than a frontier that may include pending gaps.

Recommendation: Rewrite the comments for `lastBurnedTokenId` and `_skippedTokenIds` to reflect the actual frontier semantics. Make it explicit that `lastBurnedTokenId` is the highest processed token ID, and that token IDs less than or equal to this value can still be pending when out-of-order burns occur, with those pending gaps tracked in `_skippedTokenIds`. Consider also aligning nearby comments and view function documentation so that readers understand how to combine the frontier and skipped set to determine which positions remain pending.

Dre Labs: Fixed in commit 3c0e338a.

Spearbit: Fix verified.

1. In final commit `3c0e338`, `dreWithdrawalNFT.lastBurnedTokenId` is documented as `Highest token ID that has been burned (processed) so far`, removing the incorrect claim that `1..lastBurnedTokenId` are filled consecutively.
2. In final commit `3c0e338`, the old `_skippedTokenIds` model is removed and queue documentation now explains frontier/gap semantics explicitly (see `IWithdrawalNFT.getPendingRange` and `dreWithdrawalNFT.getPendingRange` comments describing pending IDs that can still exist below the frontier).

5.5.17 `UpdateVaultAdapter` docs mismatch on zero

Severity: Informational

Context: IdreUSDManager.sol#L323

Description: The `IdreUSDManager` interface documents that the vault adapter can be set to the zero address to disable the vault-withdrawal path. However, the `dreUSDManager` implementation rejects `address(0)` in `updateVaultAdapter` and reverts with `ZeroAddress`. This mismatch makes it unclear whether disabling the adapter is a supported operational action or a configuration error. The interface documentation states that the adapter can be zero:

```
/**
 * @notice Update the vault adapter for filling withdrawals
```

```
 * @param adapter Address of the vault adapter (e.g., dreAaveAdapter). Can be zero to disable.
 */
function updateVaultAdapter(address adapter) external;
```

But the implementation forbids zero:

```
function updateVaultAdapter(address adapter) external onlyRole(WITHDRAWAL_CONFIG_ROLE) {
    if (adapter == address(0)) revert ZeroAddress();
    address oldAdapter = withdrawalVaultAdapter;
    withdrawalVaultAdapter = adapter;
    emit VaultAdapterUpdated(oldAdapter, adapter);
}
```

Recommendation: Make the behavior consistent and explicit. Either allow `adapter == address(0)` in `updateVaultAdapter` to support disabling the vault adapter (and keep the existing guard in `fillWithdrawal` that rejects `useVault == true` when no adapter is configured), or update the interface/NatSpec to remove the "Can be zero to disable" claim and document that a non-zero adapter is always required.

Dre Labs: Fixed in commit 18289e47.

Spearbit: Fix verified.

1. In fix commit `18289e4`, `contracts/interfaces/IdreUSDManager.sol` was updated to remove the NatSpec claim `Can be zero to disable` from `updateVaultAdapter(address adapter)`.
2. In final commit `3c0e338`, the interface documentation still reflects the non-zero requirement, and `dreUSDManager.updateVaultAdapter` still enforces it with `if (adapter == address(0)) revert ZeroAddress();`, so docs and implementation are now aligned.

5.5.18 `withdraw()` rejects MAX sentinel

Severity: Informational

Context: dreAaveAdapter.sol#L92-L113

Description: Some Aave integrations treat `type(uint256).max` as a sentinel value meaning "withdraw the full available balance". Aave V3 Pool `withdraw` supports this convention, so operators and integrators may reasonably assume it works end-to-end through adapter wrappers as well.

In `dreAaveAdapter`, `withdraw()` does not support the sentinel. The function computes `available = getAvailableBalance()` and reverts if `available < amount`. If `amount` is `type(uint256).max`, this check will always fail and the call will revert with `InsufficientBalance`, even when there is a non-zero withdrawable balance. This behavior is correct given the current adapter semantics, but it is non-obvious and easy to trip over in scripts that pass MAX by default.

Current logic:

```
function withdraw(
    uint256 amount,
    address to
) external onlyRole(WITHDRAWER_ROLE) returns (uint256 withdrawn) {
    if (amount == 0) revert ZeroAmount();
    if (to == address(0)) revert ZeroAddress();

    uint256 available = getAvailableBalance();
    if (available < amount) revert InsufficientBalance(available, amount);

    IERC20(aUsdc).safeTransferFrom(vault, address(this), amount);
    withdrawn = IAaveV3Pool(aavePool).withdraw(usdc, amount, to);
    if (withdrawn < amount) revert WithdrawalFailed();
```

```
    emit Withdrawn(to, withdrawn);
}
```

The impact is operational confusion rather than value loss. It can cause avoidable reverts in automation that attempts to "withdraw all" by passing MAX, and it can slow down incident response or routine operations if callers expect Aave-like sentinel behavior.

Recommendation: Make this behavior explicit and fail fast with a clear reason. Add a NatSpec note on `withdraw()` stating that MAX sentinel values are not supported, and add an explicit guard that reverts with a dedicated custom error when `amount == type(uint256).max`.

```
error MaxAmountNotSupported();

if (amount == type(uint256).max) revert MaxAmountNotSupported();
```

Dre Labs: Fixed in commit eaae07f9.

Spearbit: Fix verified.

1. In fix commit `eaae07f`, `dreAaveAdapter.withdraw()` added an explicit MAX-sentinel guard: `if (amount == type(uint256).max) revert MaxSentinelNotSupported();`.
2. In the same fix commit, `IAaveV3Adapter` added `error MaxSentinelNotSupported();` and NatSpec on `withdraw` explicitly documenting that `type(uint256).max` is not supported by the adapter.
3. In final commit `3c0e338`, the MAX guard and interface documentation remain, and `test/dreAaveAdapter.t.sol` includes `test_Withdraw_RevertIf_MaxSentinel`, confirming expected behavior.

5.5.19 Users can frontrun account freeze to dodge restrictions

Severity: Informational

Context: dreUSD.sol#L80-L84

Description: In `dreUSD.sol`, an account can be frozen which is meant to place restrictions on the movement of assets from that address, and interaction with the system. But an account owner can frontrun a `freeze()` call by the guardian, and move their assets to a different address. They can do this repeatedly to dodge the restrictions. Given that the main deployment chain is Base, this might not be a big issue right now.

Recommendation: Consider documenting this risk and steps taken to manage it, when deploying on chains with public mempools.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.20 Privileged functions are missing event emits

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Privileged functions `dreRewardsDistributor.addRewards()` and `dreAaveAdapter.recoverToken()` are missing event emissions. This will impact offchain monitoring, integrations and transparency of these critical actions.

Recommendation: Consider adding event emits for all privileged functions.

Dre Labs: Fixed in commit 3f034106.

Spearbit: Fix verified.

5.5.21 `withdrawn < amount` check in `dreAaveAdapter.withdraw()` can be improved

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `dreAaveAdapter.withdraw()` implements the below logic:

```
 // Transfer aTokens from vault to this contract
IERC20(aUsdc).safeTransferFrom(vault, address(this), amount);

// Withdraw from Aave (burns aTokens, sends USDC to recipient)
withdrawn = IAaveV3Pool(aavePool).withdraw(usdc, amount, to);

if (withdrawn < amount) revert WithdrawalFailed();
```

Aave V3's `Pool.withdraw()` either returns exact amount or reverts internally but never returns a partial amount. So `withdrawn < amount` check provides no extra safety.

Recommendation: Consider implementing the below check instead, which will detect any Aave V3 aToken integer division rounding issue that can cause the adapter to receive fractionally less than `amount` in scaled terms.

```
IERC20(aUsdc).safeTransferFrom(vault, address(this), amount);
uint256 actualBalance = IERC20(aUsdc).balanceOf(address(this));
withdrawn = IAaveV3Pool(aavePool).withdraw(usdc, actualBalance, to);
if (withdrawn < amount) revert WithdrawalFailed();
```

Dre Labs: Fixed in commit e379ece7.

Spearbit: Fix verified.

5.5.22 Setters can implement defensive checks to prevent temporary operational failures

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Privileged setters can implement defensive checks as a best-practice, which otherwise would result in temporary operational failures. Two examples are:

1. `dreAaveAdapter.setVault(...)` is missing a check to validate that the new vault has approved the adapter to spend its aUSDC, which will cause all subsequent `withdraw()` calls to revert until the new vault grants allowance. This creates an operational risk if the admin and vault operators are not coordinated.
2. `dreUSDOracle.setOracle(...)` is missing a check to ensure that `oracleAddress` actually implements `AggregatorV3Interface`, returns a sane number of `decimals()`, or that the feed corresponds to the correct token/USD pair. A misconfiguration (e.g., setting the ETH/USD feed for USDC) would silently produce incorrect prices causing user/protocol loss.

Recommendation: Consider adding relevant and reasonable defensive checks to privileged setters.

Dre Labs: Fixed in commit 4e80aa5e.

Spearbit: Fix verified.

5.5.23 Missing contract address in `_computeFiatMintStructHash` enables cross-contract signature replay

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `_computeFiatMintStructHash(...)` function in `dreUSDManager.sol` computes a hash over five fields: `mintRef`, `receiver`, `usdAmount`, `validUntil` and `chainId`. However, it does not include the contract address `address(this)`.

While `chainId` prevents cross-chain replay, the absence of the contract address means a valid custodian signature can be replayed against any other `dreUSDManager` instance deployed on the same chain. While this scenario should typically never be the case, if there were to be such a scenario for some reason, a compromised Keeper could take a legitimate custodian-signed FiatMint intended for one contract and submit it to another, minting unauthorized dreUSD from the second instance without needing a new custodian approval. Signatures are not bound to a specific contract, effectively breaking domain separation.

Recommendation: Consider:

1. Including `address(this)` in the hash computation to bind custodian signatures to the specific contract instance and update the call site to pass the contract address. The off-chain custodian signing service must also be updated to include the target contract address when computing the signature hash. Or, alternatively.
2. Replacing the custom hash construction with EIP-712 typed structured data signing, which already includes `_authDomainSeparator()` and will be consistent with the other existing signature flows.

Dre Labs: Fixed in commit fd9a905b.

Spearbit: Fix verified.

5.5.24 `mintAndStake()` requiring permit signature limits integrations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `mint(...)` function offers both a permit variant and a permit-less variant for users who have already granted allowance via `approve(...)`. However, `mintAndStake(...)` only has the permit variant. This prevents smart contracts (multisigs, routers, DeFi aggregators) that cannot produce EIP-2612 signatures, users who have already pre-approved via `approve(...)`, and tokens without EIP-2612 permit support from using the stake-on-mint flow.

Recommendation: Consider adding a permit-less `mintAndStake(...)` function that mirrors the existing permit-less `mint(...)` pattern.

Dre Labs: Acknowledged. Intended behaviour.

Spearbit: Acknowledged.

5.5.25 Upgradeable contracts missing storage gaps risk storage collisions on upgrade

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: None of the upgradeable contracts in the protocol declare a `__gap` storage variable. All use the UUPS proxy pattern and inherit from OpenZeppelin upgradeable base contracts. Without storage gaps, adding new state variables in a future upgrade will shift the storage layout and collide with slots used by child contracts or inherited bases, leading to silent data corruption.

While none of the contracts are currently inherited by other contracts, which limits the immediate risk, storage gaps are a defensive best practice for any upgradeable contract. Future upgrades may introduce inheritance hierarchies and retrofitting gaps after deployment requires careful manual slot accounting to avoid breaking the existing layout.

Recommendation: Consider adding a `__gap` array at the end of each contract's storage declarations to reserve slots for future use. The gap size should be chosen per contract based on the number of existing storage variables, following the OpenZeppelin convention of reserving enough slots to total a round number (typically 50) per contract.

Spearbit: Practical convention is that when we add `__gap` to a contract, we should count every storage slot used in that contract and set `__gap[50 - slots_used]`. This fix adds `uint256[50] private __gap` to all contracts irrespective of the storage slots already used. This breaks the conventional invariant and wastes gas, but should be ok functionally.

Dre Labs: Fixed in commit c9bc7fb3.

Spearbit: Fix verified.

5.5.26 Inaccurate filler in `WithdrawalFilled` event may limit offchain monitoring

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `WithdrawalFilled` always emits `msg.sender` as the filler. However, when `useVault == true`, the actual source of funds is `withdrawalVaultAdapter` and not `msg.sender`. Offchain indexers therefore cannot distinguish between vault-funded and `TREASURY_ROLE` operator-funded fills.

Recommendation: Consider emitting the actual fund source as the filler. For e.g.,

```
address filler = useVault ? withdrawalVaultAdapter : msg.sender;
emit WithdrawalFilled(tokenId, currentOwner, usdcAmount, filler);
```

Dre Labs: Fixed in commit 0f6a3845.

Spearbit: Fix verified.

5.5.27 Uninitialized `expressFeeRecipient` creates a partially configured express withdrawal feature

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `initialize(...)` function sets `expressWithdrawalMaxLimit` and `expressWithdrawalFeeBps` but does not set `expressFeeRecipient`. This makes express withdrawals partially configured but they will revert at runtime when `_queueExpressWithdrawal()` hits the `FeeRecipientNotSet()` check. `WITHDRAWAL_CONFIG_ROLE` must separately call `updateExpressWithdrawal(...)` to make the feature fully functional.

Recommendation: Consider:

1. Setting `expressFeeRecipient` in `initialize(...)` alongside the other express withdrawal defaults, or...
2. Removing the fee/limit defaults from `initialize(...)` so the feature is fully unconfigured until explicitly enabled by the admin.

Dre Labs: Fixed in commit b024db88.

Spearbit: Fix verified.

5.5.28 Inconsistent sanctions/freeze enforcement between permit and approve

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description:: The `permit` function is overridden in DreUSD to validate both `owner` and `spender` against the sanctions list and freeze mapping, reverting if either is flagged. However, the standard `approve` function inherits the default ERC20 implementation with no such check. This allows frozen or sanctioned addresses to set arbitrary allowances via `approve` while the same operation would revert via permit. Although `_update(...)` prevents actual token movement, any such executed approvals become instantly usable if the address is later unfrozen or removed from the sanctions list, circumventing any intended controls.

Recommendation: Consider:

1. Overriding `approve` (and other related functions) to apply the same `_validateAddress` checks on both `msg.sender` and `spender`, consistent with the permit override. Or...
2. Removing the sanctions check from permit to make the behavior uniform.

Dre Labs: Fixed in commit a080e115.

Spearbit: Fix verified.

5.5.29 Dual governance paths via `owner` and `DEFAULT_ADMIN_ROLE` can diverge

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description:: dreUSD inherits both `AccessControlUpgradeable` (used for role management, sanctions, upgrades) and `OwnableUpgradeable` (via `OFTUpgradeable` → `OAppUpgradeable`, used for LayerZero configuration). The `owner` controls critical cross-chain settings such as setting OFT peers and delegates, while `DEFAULT_ADMIN_ROLE` controls access control roles, the sanctions list, and UUPS upgrades.

After deployment, these two authorities can diverge where ownership can be transferred independently of admin role grants. A compromised or mismanaged `owner` could configure malicious OFT peers to redirect bridged tokens, while the `DEFAULT_ADMIN_ROLE` holder would have no ability to override LayerZero configuration.

Recommendation: Consider:

1. Aligning the two governance paths by either overriding `transferOwnership` to require `DEFAULT_ADMIN_ROLE` or by transferring ownership to the same governance contract (e.g., the timelock) that holds the admin role. Or, alternatively.
2. Documenting the trust assumptions explicitly and ensure deployment scripts and governance procedures treat both authorities as equally privileged.

Dre Labs: Fixed in commit 3d45935a.

Spearbit: Fix verified. Both transferring and renouncing ownership is in the hands of the `DEFAULT_ADMIN_ROLE` now.

5.5.30 Aave pool liquidity squeeze can temporarily DoS long-queue withdrawal fills

Severity: Informational

Context: dreAaveAdapter.sol#L123

Description: When `fillWithdrawal()` is called with `useVault=true`, it checks aave balance per iteration in dreUSDManager.sol#L504 and reverts with `NoBalance()` if insufficient. If Aave pool utilization is high - either naturally due to heavy borrowing or intentionally via a liquidity squeeze - the entire batch of withdrawal fills reverts even if only one position exceeds available liquidity. This can also occur unintentionally mid-batch as each successive withdrawal drains pool liquidity, causing later iterations to fail despite passing the pre-check.

Recommendation: Consider allowing `fillWithdrawal()` to skip positions that exceed current liquidity (using continue instead of revert) so that partial batch fills succeed. Alternatively, the `TREASURY_ROLE` operator can work around this by reducing batch sizes or using `useVault=false` to transfer USDC directly from treasury.

Dre Labs: Acknowledged.

Spearbit: Acknowledged.