CANTINA

USDai & sUSDai
Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
Anurag Jain, Security Researcher

May 12, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

USDai is an $M-backed stablecoin. Staked USDai (sUSDai) is a yield-bearing vault token backed by USDai $M emissions and MetaStreet Pool loans against AI hardware, compute, and DePIN assets.

From Apr 26th to May 7th the Cantina team conducted a review of metastreet-usdai-contracts on commit hash 1128aced. The team identified a total of **20** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	8	6	2
Gas Optimizations	1	1	0
Informational	10	8	2
Total	**20**	**16**	**4**

3 Findings

3.1 Medium Risk

3.1.1 Inflation Attack

Severity: Medium Risk

Context: StakedUSDai.sol#L418-L448

Description: MetaStreet has a provision for `LOCKED_SHARES` on first mint (totalSupply==0) to prevent Inflation attacks. But it seems that contract currently misses to mint these `LOCKED_SHARES` to say `0xdead` address.

Proof of Concept:

1. User deposits `LOCKED_SHARES + 1`.
2. `convertToShares` is called internally. Since this is first mint (totalSupply==0) so `initialDeposit` is true and `depositSharePrice` will be `FIXED_POINT_SCALE` making `shares=assets` .

```
function convertToShares(
        uint256 assets
    ) public view nonZeroUint(assets) returns (uint256) {
        /* Check if initial deposit */
        bool initialDeposit = totalShares() < LOCKED_SHARES;

        /* Compute shares */
        uint256 shares = ((assets * FIXED_POINT_SCALE) / depositSharePrice());

        /* Check if initial deposit and shares is less than locked shares */
        if (initialDeposit && shares <= LOCKED_SHARES) revert InvalidAmount();

        /* Compute shares. If initial deposit, lock subset of shares */
        return shares - (initialDeposit ? LOCKED_SHARES : 0);
    }
```

3. So value returned would be `(LOCKED_SHARES + 1)-LOCKED_SHARES = 1`. Thus `1` share get minted to User.
4. So, `1` share is now worth `LOCKED_SHARES + 1` assets.
5. Attacker can now front run any victim deposit by large donation, causing victim to get lower than expected shares especially if deposit was made with min amount `0`.

Note: The proof of concept only shows deposit flow, but same is also true for mint flow. In both cases `LOCKED_-SHARES` should get minted to `0xdead` address on first mint (`totalSupply == 0`).

Impact Explanation: Impact is High since Attacker can perform this attack to cause loss of victim funds.

Likelihood Explanation: The likelihood is Low since this can only happen when `totalSupply == 0`.

Recommendation: Mint `LOCKED_SHARES` to say `0xdead` address on first mint which makes this attack expensive.

MetaStreet: Fixed in commit 4e214097.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Controller-based DoS vulnerability in redemption requests

Severity: Low Risk

Context: RedemptionLogic.sol#L256-L265

Summary: The `StakedUSDai` contract is vulnerable to a denial-of-service attack where malicious users can prevent specific controllers from creating new redemption requests by exploiting the per-controller limit on active redemptions. This vulnerability could temporarily block legitimate users from redeeming their shares when they attempt to use a targeted controller address.

Description: The `StakedUSDai` contract's redemption system is vulnerable to a denial-of-service attack where malicious users can block legitimate controllers from creating new redemption requests. The vulnerability stems from the way redemption requests are tracked by controller address rather than the owner of the shares.

In the `_requestRedeem()` function of the `RedemptionLogic` library, there's a limit enforced on the number of active redemptions per controller:

```
/* Validate active redemptions count is less than max allowed */
if (redemptionState_.redemptionIds[controller].length() == MAX_ACTIVE_REDEMPTIONS_COUNT) {
    revert IStakedUSDai.InvalidRedemptionState();
}
```

This limit is set to a constant value of 50:

```
/**
 * @notice Max active redemptions per controller
 */
uint256 internal constant MAX_ACTIVE_REDEMPTIONS_COUNT = 50;
```

An attacker can exploit this by creating multiple small redemption requests (e.g., 1 share each) targeting a specific controller address until the maximum limit is reached. Once this happens, any legitimate attempt to create a new redemption request with that controller will fail, effectively blocking the controller from processing further redemptions.

Proof of Concept:

1. An attacker identifies a controller address (potentially a major protocol integration or a significant user).
2. The attacker creates 50 separate redemption requests with 1 share each using the targeted controller address.
3. When the legitimate controller tries to create a new redemption request, the transaction will revert with `InvalidRedemptionState`.
4. The controller is effectively blocked from creating new redemption requests until some of the existing ones are fully processed and removed.

Impact Explanation: The impact of this vulnerability is medium. While it doesn't directly lead to the loss of funds, it can significantly disrupt the normal operation of the protocol by preventing legitimate users from initiating redemption requests through specific controllers. This could result in users being temporarily unable to redeem their shares, affecting the liquidity and usability of the protocol. In scenarios where timely redemptions are critical (such as during market volatility), this denial of service could lead to indirect financial losses for affected users.

Likelihood Explanation: The likelihood is low. The attack requires an attacker to pay gas for multiple transactions, and provides no direct financial gain. Additionally, users can choose different controllers, and the attack effect is temporary as redemptions are eventually serviced.

Recommendation: Consider refactoring the code to track redemption requests by the caller rather than the controller address. This would isolate redemption requests from different users and prevent targeted DoS attacks.

```diff
  function _requestRedeem(
      StakedUSDaiStorage.RedemptionState storage redemptionState_,
      uint64 timelock_,
      uint256 shares,
     address controller
  ) external returns (uint256) {
      /* Validate active redemptions count is less than max allowed */
-     if (redemptionState_.redemptionIds[controller].length() == MAX_ACTIVE_REDEMPTIONS_COUNT) {
+     if (redemptionState_.redemptionIds[msg.sender].length() == MAX_ACTIVE_REDEMPTIONS_COUNT) {
          revert IStakedUSDai.InvalidRedemptionState();
      }

      /* Rest of the function... */

      /* Add redemption ID */
-     redemptionState_.redemptionIds[controller].add(redemptionId);
+     redemptionState_.redemptionIds[msg.sender].add(redemptionId);

      return redemptionId;
  }
```

The rest of the redemption logic needs to be adapted accordingly as well.

MetaStreet: Fixed in commit eff38bac.

Cantina Managed: Fix verified.

3.2.2 Redemption queue stalling through small requests

Severity: Low Risk

Context: RedemptionLogic.sol#L309-L320

Summary: The `StakedUSDai` contract is vulnerable to a queue stalling attack where malicious users can create multiple small redemption requests to delay the processing of legitimate redemptions, forcing admins to waste gas on processing these low-value requests.

Description: The redemption system in `StakedUSDai` processes redemption requests in a first-in-first-out (FIFO) queue manner. However, there is no minimum threshold for the size of a redemption request. This allows an attacker to create many small redemption requests (e.g., 1 share each) that must be processed before later, legitimate redemption requests.

When admins call the `serviceRedemptions()` function to process pending redemptions, they have to process the queue in order. Small redemption requests at the front of the queue force admins to spend gas processing these low-value transactions before getting to larger, legitimate redemption requests further back in the queue.

The relevant code in the `_processRedemptions()` function shows how redemptions are processed in FIFO order:

```solidity
/* Process redemptions */
uint256 remainingShares = shares;
uint256 amountProcessed;
while (remainingShares > 0 && head != 0) {
    /* Get redemption */
    IStakedUSDai.Redemption storage redemption_ = redemptionState_.redemptions[head];

    /* Compute shares to fulfill */
    uint256 fullfilledShares = Math.min(redemption_.pendingShares, remainingShares);

    /* ... processing logic ... */

    /* If redemption is completely fulfilled, update head */
    if (redemption_.pendingShares == 0) head = redemption_.next;
}
```

Impact Explanation: The impact is low to medium. This vulnerability cannot permanently block redemptions, but it can delay the processing of legitimate redemption requests, causing inconvenience to users and increasing gas costs for admins. The attack only delays rather than prevents redemptions, as the queue will eventually be processed in full.

Likelihood Explanation: The likelihood is low. The attack requires creating multiple transactions that provide no direct financial gain to the attacker. The gas costs of creating numerous small redemption requests are substantial, making this attack economically unattractive except as a griefing attack.

Recommendation: Implement a minimum threshold for redemption requests to prevent excessively small requests from entering the queue:

```
  function _requestRedeem(
      StakedUSDaiStorage.RedemptionState storage redemptionState_,
      uint64 timelock_,
      uint256 shares,
      address controller
  ) external returns (uint256) {
      /* Validate active redemptions count is less than max allowed */
      if (redemptionState_.redemptionIds[controller].length() == MAX_ACTIVE_REDEMPTIONS_COUNT) {
          revert IStakedUSDai.InvalidRedemptionState();
      }

+     /* Enforce minimum share amount for redemption requests */
+     if (shares < 1e18) {
+         revert IStakedUSDai.InvalidAmount();
+     }

      /* Rest of the function... */
 }
```

MetaStreet: Fixed in commit eff38bac.

Cantina Managed: Fix verified.

3.2.3 Deviations from ERC-7540 specification

Severity: Low Risk

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

Description: The `StakedUSDai` contract implements many aspects of the ERC-7540 asynchronous ERC-4626 standard, but contains several deviations from the specification which could affect interoperability with other systems expecting full compliance. These deviations are as follows:

1. Missing ERC-7575 support: The ERC-7540 specification mandates support for ERC-7575, particularly the `share()` method. The current implementation does not include this method or explicitly inherit from an ERC-7575 interface.

2. Incomplete ERC-165 implementation: While the contract implements `supportsInterface()`, it does not check for the ERC-7575 interface ID (0x2f0a18c5) as required by ERC-7540.

3. Non-standard request ID system: The contract uses a redemption queue with a linked-list data structure instead of the simpler request ID system described in the specification. Additionally, the redemption service mechanism doesn't explicitly ensure that all requests with the same ID become claimable at the same pro-rata rate.

4. Incomplete requestRedeem approval logic: The ERC-7540 specification states that redemption request approval may come either from ERC-20 approval over the shares of the owner or from operator approval. The current implementation checks for operator approval but does not utilize standard ERC-20 allowances when the caller is neither the owner nor an approved operator.

5. Incorrect Withdraw event parameter ordering: The spec requires that when the Withdraw event is emitted, the first parameter should be the controller and the second parameter should be the receiver. The current implementation emits `Withdraw(msg.sender, controller, receiver, amount, shares)`, which places the controller in the third position rather than the first.

Recommendation: Consider the following changes to ensure full compliance with the ERC-7540 specification:

1. Implement the ERC-7575 interface, particularly the `share()` method.

2. Update the `supportsInterface()` method to check for the ERC-7575 interface ID (`0x2f0a18c5`).

3. Consider aligning the request ID implementation with the ERC-7540 specification, ensuring requests with the same ID become claimable at the same pro-rata rate.

4. Consider adapting the `requestRedeem()` approval logic to check ERC-20 allowances when the caller is not the owner or an approved operator.
5. Adjust the Withdraw event parameters to follow the ordering specified in ERC-7540, with controller as the first parameter and receiver as the second.

MetaStreet: Fixed in commit d2f4a39f.

Cantina Managed: Fix verified.

3.2.4 Rounding in minimum amount calculation can lead to higher slippage than expected

Severity: Low Risk

Context: USDai.sol#L210

Summary: In the USDai contract, the `_deposit()` function uses `_unscale()` to convert the minimum USDai amount to base token amount when performing swaps. Since `_unscale()` rounds down by using integer division, users may receive less USDai than their specified minimum amount, allowing for higher slippage than intended.

Description: The issue occurs in the `_deposit()` function when handling deposits of non-base tokens:

```
if (depositToken != address(_baseToken)) {
    /* Approve the adapter to spend the token in */
    IERC20(depositToken).approve(address(_swapAdapter), depositAmount);

    /* Swap in deposit token for base token */
    usdaiAmount = _scale(_swapAdapter.swapIn(depositToken, depositAmount, _unscale(usdaiAmountMinimum), data));
}
```

The `_unscale()` function performs integer division which always rounds down:

```
function _unscale(uint256 value) public view returns (uint256) {
    return value / _scaleFactor();
}
```

For example, if:

- A user sets `usdaiAmountMinimum = 1.111111555555555555e18`.
- `_scaleFactor() = 1e12` (for a 6 decimal base token).
- `_unscale(usdaiAmountMinimum) = 1111111` (rounded down from 1.111111555555).

This means the actual minimum amount used in the swap will be lower than intended, potentially allowing for higher slippage than the user specified.

Impact Explanation: The impact is low as this only results in minor increases in potential slippage and does not lead to significant loss of funds. The maximum additional slippage is bounded by the scaling factor (e.g., `1e-12` for 6 decimal tokens).

Likelihood Explanation: The likelihood is medium as this will affect any deposit transaction where the minimum amount is not cleanly divisible by the scaling factor, which is common with calculated minimum amounts that include slippage tolerances.

Recommendation: Modify the `_unscale()` function to round up when used for minimum amounts:

```
+ /// @notice Helper function to scale down a value, rounding up
+ /// @param value Value
+ /// @return Unscaled value rounded up
+ function _unscaleUp(uint256 value) public view returns (uint256) {
+     return (value + _scaleFactor() - 1) / _scaleFactor();
+ }

  function _deposit(
      address depositToken,
      uint256 depositAmount,
      uint256 usdaiAmountMinimum,
      address recipient,
      bytes calldata data
  ) internal nonZeroUint(depositAmount) nonZeroAddress(recipient) returns (uint256) {
      // ...
      if (depositToken != address(_baseToken)) {
          IERC20(depositToken).approve(address(_swapAdapter), depositAmount);
-         usdaiAmount = _scale(_swapAdapter.swapIn(depositToken, depositAmount, _unscale(usdaiAmountMinimum),
↪  data));
+         usdaiAmount = _scale(_swapAdapter.swapIn(depositToken, depositAmount, _unscaleUp(usdaiAmountMinimum),
↪  data));
      }
      // ...
 }
```

This ensures that the minimum amount used in swaps is always rounded up, providing at least the slippage protection requested by users.

MetaStreet: Fixed in commit 2c25d54a.

Cantina Managed: Fix verified.

3.2.5 Blacklisted User redemption cannot be skipped

Severity: Low Risk

Context: StakedUSDai.sol#L731

Description: If a user request redeem tokens and MetaStreet figures it to be a malicious request, then blacklisting this user wont help and project will be forced to allocate funds for the blacklisted request.

This is because, all redemptions are processed from head to tail and there is no provision to skip any redemption id. This means that all redemption made post Attacker redemption can only be completed once Attacker redemption is completed.

Proof of Concept:

1. User `A` request redemption of its `100` staked token.
2. User `B` request redemption of its `50` staked token.
3. MetaStreet figures `A` to be a malicious user and blacklist it.
4. Startegy Admin calls `serviceRedemptions` to process redemption request.
5. Both User `A` & `B` transaction are in redemption queue.
6. Redemption starts from head to tail and currently there is no way to skip any redemption request.
7. Thus User `A` (who is blacklisted) request need to be redeemed before User `B` redemption can happen.

```
while (remainingShares > 0 && head != 0) {
    /* Get redemption */
    IStakedUSDai.Redemption storage redemption_ = redemptionState_.redemptions[head];

    /* Compute shares to fulfill */
    uint256 fullfilledShares = Math.min(redemption_.pendingShares, remainingShares);

    /* Compute amount to fulfill */
    uint256 fullfilledAmount = Math.mulDiv(fullfilledShares, redemptionSharePrice_, FIXED_POINT_SCALE);

    /* Update redemption pending, redeemable shares, and withdrawable amount */
    redemption_.pendingShares -= fullfilledShares;
    redemption_.redeemableShares += fullfilledShares;
    redemption_.withdrawableAmount += fullfilledAmount;

    /* Update remaining shares and amount processed */
    remainingShares -= fullfilledShares;
    amountProcessed += fullfilledAmount;

    /* Emit RedemptionProcessed */
    emit IStakedUSDai.RedemptionProcessed(
        head, redemption_.controller, fullfilledShares, fullfilledAmount, redemption_.pendingShares
    );

    /* If redemption is completely fulfilled, update head */
    if (redemption_.pendingShares == 0) head = redemption_.next;
}
```

Impact Explanation: Impact will be Medium since:

1. If Admin decides to not allocate funds for Blacklisted user redemption then any future redemption request post Blacklisted user request will DOS since redemption always happen from head to tail.
2. If Admin allocates USDai funds for Blacklisted user redemption then these funds will remain locked in contract with no way of recovery.

Likelihood Explanation: The likelihood is Low as Blacklisting feature is still on its early implementation phase.

Recommendation: Admin should be able to provide blacklisted id(s) which are meant to be skipped while performing `serviceRedemptions`.

MetaStreet: Acknowledged.

Cantina Managed: Acknowledged.

3.2.6 Blacklisting can be bypassed

Severity: Low Risk

Context: StakedUSDai.sol#L135, StakedUSDai.sol#L757-L774

Description: Blacklisting can be bypassed by simply transferring/bridging shares from blacklisted to non-blacklisted user. Contract currently misses restrictions on:

1. Mint/Burn by Bridge: Blacklisted user can simply bridge there tokens to another chain. Since `mint` and `burn` does not have blacklist checks, thus bridging is completed without any restrictions. Once bridging is complete, the bridged asset has no restrictions.
2. Transfer/TransferFrom: Blacklisted user can simply use `transfer/transferFrom` for transferring its token to another whitelisted address.

Proof of Concept:

1. Observe that both `mint` and `burn` function misses blacklisting checks.

```
function mint(address to, uint256 amount) external onlyRole(BRIDGE_ADMIN_ROLE) {
    /* Mint supply */
    _mint(to, amount);

    /* Update bridged supply */
    _getBridgedSupplyStorage().bridgedSupply -= amount;
}

/**
 * @inheritdoc IMintable
 */
function burn(address from, uint256 amount) external onlyRole(BRIDGE_ADMIN_ROLE) {
    /* Burn supply */
    _burn(from, amount);

    /* Update bridged supply */
    _getBridgedSupplyStorage().bridgedSupply += amount;
}
```

2. Similarly, observe that `_transfer` or `_update` function is not overridden and no restriction is placed on `transfer/transferFrom`.

Impact Explanation: Impact is Medium. User can bypass blacklisting by transferring or bridging funds from blacklisted account to whitelisted account.

Likelihood Explanation: The likelihood is Low as Blacklisting feature is still on its early implementation phase.

Recommendation: Override `_update` function to check blacklisting on `msg.sender`,`from`,`to` address, if any is blacklisted then revert.

MetaStreet: Fixed in commits 9d0eea89 and 14131da.

Cantina Managed: Fix verified:

1. `_update` checks blacklisting status for `msg.sender`,`from`,`to` and revert if any of these is blacklisted.
2. `notBlacklisted(operator)` is removed while calling `setOperator` which allows users to remove blacklisted operators.
3. However now users can set blacklisted operator (malicious operation with current code cannot be done due to blacklisting checks) which is acknowledged by Team
4. As of now, Blacklisting will only be applicable for Staked USDai and not USDai as acknowledged by Team.
5. As acknowledged by team, `maxWithdraw`,`maxRedeem`, `maxDeposit`, `maxMint` will not be limited by constraints like pausing, blacklisting

3.2.7 Stale price check is missing in Oracle

Severity: Low Risk

Context: ChainlinkPriceOracle.sol#L235-L240

Description: The contract uses `tokenPriceFeed.latestRoundData()` to get latest price, but does not validate the `updatedAt` timestamp. This can allow stale prices to be used in worst scenario.

```
/* Get token price */
(, int256 tokenPrice,,,) = tokenPriceFeed.latestRoundData();
uint8 tokenDecimals = tokenPriceFeed.decimals();
tokenPrice = _scalePrice(tokenPrice, tokenDecimals);

/* Get M NAV price with [1](https://cantina.xyz/code/97d21db4-66af-4775-aac0-9d9ee4f0da0c/src/oracles/Chainlin⌋
↪  kPriceOracle.sol?comment_id=25a2e93a-eb98-4452-a4a9-c5517d6b91b9&lines=235,240)0 ** _mNavDecimals as
↪  ceiling */
(, int256 mNavPrice,,,) = _mNavPriceFeed.latestRoundData();
mNavPrice = _scalePrice(mNavPrice < M_PRICE_CEILING ? mNavPrice : M_PRICE_CEILING, _mNavDecimals);

return (tokenPrice * USDAI_SCALING_FACTOR) / mNavPrice;
```

Recommendation: Check for price staleness using below logic.

```
require(updatedAt >= block.timestamp - MAX_DELAY, "Stale price");
```

MetaStreet: Acknowledged.

Cantina Managed: Acknowledged.

3.2.8 Asset amounts should be rounded in vault's favor for deposits

Severity: Low Risk

Context: StakedUSDai.sol#L320-L329

Description: When converting shares to assets during the mint process, the current implementation rounds down which slightly favors depositors. While the impact is minimal due to share price calculations and zero amount checks, it's safer to consistently round in the vault's favor.

While rounding down slightly favors the depositor (they could get a maximum discount of 1 asset unit), this is generally acceptable because:

1. The deposit share price calculation already favors the vault.
2. The magnitude of potential value capture is limited to 1 asset unit per deposit.
3. The check for `amount == 0` prevents extreme rounding cases.
4. This will become even less significant when proper minimum share requirements are implemented.

However, for consistency and maximum safety, it is recommended to always round in the vault's favor.

Recommendation: Modify the share to asset conversion to round up for deposits:

```
- uint256 amount = convertToAssets(shares);
+ // Round up required assets for deposits to favor the vault
+ uint256 amount = _convertToAssets(shares, Math.Rounding.Up);

  /* If amount is 0 or more than max amount, revert */
  if (amount == 0 || amount > maxAmount) revert InvalidAmount();

  /* Mint shares */
  _mint(receiver, shares);

  /* Deposit assets */
  IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
```

This change ensures the vault always receives at least the required assets to back issued shares, maintaining a more conservative approach to rounding in the vault's favor.

MetaStreet: Fixed in commit 21aa8aa0.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Cache decimals scaling factor as immutable

Severity: Gas Optimization

Context: USDai.sol#L158-L160

Description: The `_scaleFactor()` function in USDai makes an external call to get the base token's decimals every time scaling is needed:

```
function _scaleFactor() internal view returns (uint256) {
    return 10 ** (18 - IERC20Metadata(address(_baseToken)).decimals());
}
```

Since ERC20 decimals are (generally) immutable, this value can be calculated once during construction and stored as an immutable variable to save gas on repeated external calls.

Recommendation: Cache the scale factor during construction:

```diff
  contract USDai is ... {
+     uint256 private immutable _SCALE_FACTOR;

      constructor(address swapAdapter_) {
          _disableInitializers();
          _swapAdapter = ISwapAdapter(swapAdapter_);
          _baseToken = IERC20(_swapAdapter.baseToken());
+         _SCALE_FACTOR = 10 ** (18 - IERC20Metadata(address(_baseToken)).decimals());
      }

-     function _scaleFactor() internal view returns (uint256) {
-         return 10 ** (18 - IERC20Metadata(address(_baseToken)).decimals());
-     }
+     function _scaleFactor() internal view returns (uint256) {
+         return _SCALE_FACTOR;
+     }
  }
```

This change eliminates repeated external calls to `decimals()`, reducing gas costs for operations.

MetaStreet: Fixed in commit 29e6a9a9.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 SafeERC20 methods should be used consistently for token operations

Severity: Informational

Context: USDai.sol#L207

Description: The contracts have inconsistent usage of SafeERC20 methods for token operations:

1. In USDai's `_deposit()` function:

   ```
   IERC20(depositToken).transferFrom(msg.sender, address(this), depositAmount);
   IERC20(depositToken).approve(address(_swapAdapter), depositAmount);
   ```

2. In USDai's `_withdraw()` function:

   ```
   _baseToken.approve(address(_swapAdapter), baseTokenAmount);
   IERC20(withdrawToken).transfer(recipient, withdrawAmount);
   ```

3. In UniswapV3SwapAdapter's `swapIn()` function:

   ```
   IERC20(inputToken).transferFrom(msg.sender, address(this), inputAmount);
   IERC20(inputToken).approve(address(_swapRouter), inputAmount);
   ```

4. In UniswapV3SwapAdapter's `swapOut()` function:

   ```
   _baseToken.transferFrom(msg.sender, address(this), baseAmount);
   _baseToken.approve(address(_swapRouter), baseAmount);
   ```

Some tokens like USDT require approvals to be set to 0 first before setting a new non-zero value. Additionally, some tokens don't return a boolean value on transfers as required by the ERC20 standard, or may revert silently on failure. While these issues are not currently problems with the specific tokens used, using OpenZeppelin's SafeERC20 methods (`safeTransfer()`, `safeTransferFrom()`, and `forceApprove()`) would make the contracts more robust against potential future changes or non-standard tokens.

Recommendation: Use SafeERC20's methods consistently throughout the contracts:

1. For USDai, update all token operations:

```
    contract USDai is ... {
        using SafeERC20 for IERC20;

        function _deposit(...) internal ... {
            // ...
-           IERC20(depositToken).transferFrom(msg.sender, address(this), depositAmount);
+           IERC20(depositToken).safeTransferFrom(msg.sender, address(this), depositAmount);
            // ...
-           IERC20(depositToken).approve(address(_swapAdapter), depositAmount);
+           IERC20(depositToken).forceApprove(address(_swapAdapter), depositAmount);
            // ...
        }

        function _withdraw(...) internal ... {
            // ...
-           _baseToken.approve(address(_swapAdapter), baseTokenAmount);
+           _baseToken.forceApprove(address(_swapAdapter), baseTokenAmount);
            // ...
-           IERC20(withdrawToken).transfer(recipient, withdrawAmount);
+           IERC20(withdrawToken).safeTransfer(recipient, withdrawAmount);
            // ...
        }
    }
```

2. For `UniswapV3SwapAdapter`, update all token operations:

```
    contract UniswapV3SwapAdapter is ISwapAdapter, AccessControl {
        using EnumerableSet for EnumerableSet.AddressSet;
        using SafeERC20 for IERC20;

        function swapIn(...) external ... {
            // ...
-           IERC20(inputToken).transferFrom(msg.sender, address(this), inputAmount);
+           IERC20(inputToken).safeTransferFrom(msg.sender, address(this), inputAmount);
            // ...
-           IERC20(inputToken).approve(address(_swapRouter), inputAmount);
+           IERC20(inputToken).forceApprove(address(_swapRouter), inputAmount);
            // ...
        }

        function swapOut(...) external ... {
            // ...
-           _baseToken.transferFrom(msg.sender, address(this), baseAmount);
+           _baseToken.safeTransferFrom(msg.sender, address(this), baseAmount);
            // ...
-           _baseToken.approve(address(_swapRouter), baseAmount);
+           _baseToken.forceApprove(address(_swapRouter), baseAmount);
            // ...
        }
    }
```

3. Similar changes (`approve` $\rightarrow$ `forceApprove`) should also be applied to `BasePositionManager` and `PoolPositionManager` contracts.

These changes ensure compatibility with non-standard ERC20 tokens and improve the overall safety of token operations, making the contracts more robust for future integrations.

MetaStreet: Fixed in commit 07f76e56.

Cantina Managed: Fix verified.

3.4.2 Improve yield harvesting function separation and accuracy

Severity: Informational

Context: BasePositionManager.sol#L110-L133

Description: The `harvestBaseYield()` function in BasePositionManager combines two distinct operations:

1. Claiming accrued yield from both the contract and USDai addresses.
2. Converting wrapped mTokens to USDai.

Additionally, there are several minor issues with the current implementation:

- The event emission is misleading as base yield can be harvested outside this function by anyone.
- The function name doesn't accurately reflect its primary purpose of depositing/converting tokens.
- The emitted amount may be inaccurate due to scaling operations and minimum amount differences.

Recommendation: Split the functionality into separate functions and improve accuracy:

```diff
- function harvestBaseYield(uint256 usdaiAmount) external onlyRole(STRATEGY_ADMIN_ROLE) nonReentrant {
+ /// @notice Claims accrued yield from both addresses
+ function claimBaseYield() external {
       /* Claim yield */
       _wrappedMToken.claimFor(address(_usdai));
       _wrappedMToken.claimFor(address(this));
+      emit BaseYieldClaimed();
+ }

+ /// @notice Converts wrapped mTokens to USDai
+ /// @param usdaiAmount Amount of USDai to receive (will be scaled)
+ /// @return Amount of USDai actually received
+ function depositBaseToken(uint256 usdaiAmount) external onlyRole(STRATEGY_ADMIN_ROLE) nonReentrant returns
↪  (uint256) {
       /* Scale down the USDai amount */
       uint256 wrappedMAmount = _unscale(usdaiAmount);

       /* Validate balance */
       if (wrappedMAmount > _wrappedMToken.balanceOf(address(this))) {
           revert InsufficientBalance();
       }

       /* Approve wrapped M token to spend USDai */
       _wrappedMToken.approve(address(_usdai), wrappedMAmount);

       /* Swap wrapped M token to USDai */
-      _usdai.deposit(address(_wrappedMToken), wrappedMAmount, 0, address(this));
+      uint256 receivedAmount = _usdai.deposit(address(_wrappedMToken), wrappedMAmount, 0, address(this));

-      /* Emit BaseYieldHarvested */
-      emit BaseYieldHarvested(usdaiAmount);
+      /* Emit BaseTokenDeposited with actual received amount */
+      emit BaseTokenDeposited(receivedAmount);
+      return receivedAmount;
  }

  event BaseYieldClaimed();
- event BaseYieldHarvested(uint256 amount);
+ event BaseTokenDeposited(uint256 receivedAmount);
```

MetaStreet: Fixed in commit 13a04f47.

Cantina Managed: Fix verified.

3.4.3 DoS on removing `supportedToken` under certain scenarios

Severity: Informational

Context: ChainlinkPriceOracle.sol#L267

Description: If a token used in an active pool with pending shares is removed then any attempt to retrieve its price via `_priceOracle.price` will revert. This happens since `_priceOracle.price` reverts if provided `token` is not supported.

```solidity
function price(
    address token_
) external view returns (uint256) {
    /* Validate token is supported */
    if (!supportedToken(token_)) revert UnsupportedToken(token_);
```

Proof of Concept:

1. Strategy Admin deposits amount `5` in Pool `P1` with currency token `Curr1`.

2. Admin observe issue with the pool and calls `setTokenPriceFeeds` with `priceFeeds_` for `Curr1` as `address(0)`.

```solidity
function _setTokenPriceFeed(address[] memory tokens_, address[] memory priceFeeds_) internal {
    /* Validate tokens and price feeds */
    if (tokens_.length != priceFeeds_.length) {
        revert InvalidLength();
    }

    /* Set token price feeds */
    for (uint256 i = 0; i < tokens_.length; i++) {
        /* Validate token */
        if (tokens_[i] == address(0)) revert InvalidAddress();

        if (priceFeeds_[i] != address(0)) {
            /* Add token */
            _tokens.add(tokens_[i]);
        } else {
            /* Remove token */
            _tokens.remove(tokens_[i]);
        }

        /* Set token price feed which can be zero address if it is for unsetting */
        _priceFeeds[tokens_[i]] = AggregatorV3Interface(priceFeeds_[i]);

        /* Emit event */
        emit TokenPriceFeedSet(tokens_[i], priceFeeds_[i]);
    }
}
```

3. This removes `Curr1` from supported token.
4. Strategy Admin calls `poolRedeem` to redeem all `5` amount from the impacted pool.
5. This schedules a pending request which is meant to be fulfilled at future time `X`.

```solidity
function poolRedeem(
    address pool,
    uint128 tick,
    uint256 shares
) external onlyRole(STRATEGY_ADMIN_ROLE) nonReentrant returns (uint128) {
    /* Redeem */
    uint128 redemptionId = IPool(pool).redeem(tick, shares);

    /* Add redemption ID */
    _getPoolsStorage().position[pool].redemptionIds[tick].add(redemptionId);

    /* Emit PoolRedeemed */
    emit PoolRedeemed(pool, tick, shares, redemptionId);

    return redemptionId;
}
```

6. From now till `X`, all staked USDai operation fails since `_assets` function fails. Below flow shows the issue:

```
_assets -> PoolPositionManager._assets(valuationType) -> Navigate pools -> _getTickPosition -> Get shares for
↪  pool linked with `Curr1` -> Calculate NAV -> _value(Curr1,...) -> _priceOracle.price(Curr1) -> if
↪  (!supportedToken(token_)) revert UnsupportedToken(token_) -> Reverts
```

Impact Explanation: Impact is Medium since this will cause multiple Staked USDai operations to fail. Severity is marked Informational since this requires Admin to perform error.

Likelihood Explanation: The likelihood is Low since this would require Strategy Admin to remove a token used in pool with pending shares.

Recommendation: Add a check on `setTokenPriceFeeds` to ensure that token removed is not used in any active pool with pending shares.

MetaStreet: Acknowledged.

Cantina Managed: Acknowledged.

3.4.4 Introduce Separate Role for Unpausing the Contract

Severity: Informational

Context: StakedUSDai.sol#L720

Description: Currently both `pause` and `unpause` operation can be performed by `PAUSE_ADMIN_ROLE`. For improved access control and clearer separation of responsibilities, it would be beneficial to introduce a new role `UNPAUSER_ADMIN_ROLE` which would be solely responsible for unpausing the contract.

Recommendation: Introduce a dedicated `UNPAUSER_ADMIN_ROLE` and restrict the `unpause` operation to this role only.

MetaStreet: Acknowledged.

Cantina Managed: Acknowledged.

3.4.5 Unrequired approval for USDai

Severity: Informational

Context: PoolPositionManager.sol#L305

Description: When Strategy Admin deposit on Pool then `USDai` is withdrawn in exchange for the required pool currency. The `withdraw` function burns provided `USDai` token and never transfers the `USDai` token. This means no approval is required for spending `USDai`. However, `poolDeposit` gives this unrequired approval.

```
function poolDeposit(
    address pool,
    uint128 tick,
    uint256 usdaiAmount,
    uint256 poolCurrencyAmountMinimum,
    uint256 minShares,
    bytes calldata data
) external onlyRole(STRATEGY_ADMIN_ROLE) nonReentrant returns (uint256) {
    // ...
    /* Approve USDai */
    _usdai.approve(address(_usdai), usdaiAmount);

    /* Swap USDai to pool currency token */
    uint256 poolCurrencyAmount =
        _usdai.withdraw(poolCurrency, usdaiAmount, poolCurrencyAmountMinimum, address(this), data);
    // ...
```

Recommendation: Make below changes:

```
- _usdai.approve(address(_usdai), usdaiAmount);
```

MetaStreet: Fixed in commit 241e8b4a.

Cantina Managed: Fix verified.

3.4.6 Unrequired check placed on `convertToAssets`

Severity: Informational

Context: StakedUSDai.sol#L443-L444

Description: Below check is added on both `convertToShares` and `convertToAssets`.

```
/* Check if initial deposit and shares is less than locked shares */
if (initialDeposit && shares <= LOCKED_SHARES) revert InvalidAmount();
```

Now,

1. `convertToShares` is used in `deposit` function where User defines a amount and get back shares worth that amount. If its initial mint then user gets shares worth `amount-LOCKED_SHARES` . In this case it is important that `(amount=shares) <= LOCKED_SHARES` else it would underflow.

2. `convertToAssets` is used in `mint` function where User defines a share to receive . If its initial mint then user gets the shares specified but has to pay extra amount of `(amount=shares)+LOCKED_SHARES` . In this case it does not matter to check `shares <= LOCKED_SHARES`.

Thus in case of deposit, if amount was `LOCKED_SHARES+1`, user gets 1 share (assuming inflation issue is fixed). However in case of mint, if share asked was `1`, he would be asked to pay `LOCKED_SHARES+1` which is correct. But this fails since `1<=LOCKED_SHARES`.

Recommendation: Remove below condition from `convertToAssets`.

```
  /* Check if initial deposit and shares is less than locked shares */
- if (initialDeposit && shares <= LOCKED_SHARES) revert InvalidAmount();
```

MetaStreet: Fixed in commit bd69d3b7.

Cantina Managed: Fix verified.

3.4.7 Bridge Minting and Burning Should Be Disabled When Contract Is Paused

Severity: Informational

Context: StakedUSDai.sol#L757-L774

Description: If Staked USDai is paused then `mint` and `burn` via Bridge should also be paused. In case if this restriction is meant to be placed on MetaStreet UI then this can be skipped.

Recommendation: Add `whenNotPaused` to functions below in Staked USDai contract:

```
function mint(address to, uint256 amount) external onlyRole(BRIDGE_ADMIN_ROLE) {
    // ...
}

function burn(address from, uint256 amount) external onlyRole(BRIDGE_ADMIN_ROLE) {
    // ...
}
```

Note: Regular token transfer will not be paused as confirmed by team.

MetaStreet: Fixed in commit acdca206.

Cantina Managed: Fix verified.

3.4.8 Ensure Price feed decimals are not greater than `18`

Severity: Informational

Context: ChainlinkPriceOracle.sol#L216

Description: Any token whose oracle price feed decimals is more than 18, if added by Admin will fail while retrieving price. This happens since all scaling is done till USDai decimals and any decimals above that will cause underflow error as shown in below function.

```
function _scalePrice(int256 price_, uint8 priceDecimals) internal pure returns (int256) {
    return price_ * int256(10 ** uint256(USDAI_DECIMALS - priceDecimals));
}
```

Recommendation: While adding new token's price feed, check that `priceFeeds_[i].decimals() <= 18`.

MetaStreet: Fixed in commit 9d638b18.

Cantina Managed: Fix verified.

3.4.9 Missing minimum amount validation for base token withdrawals

Severity: Informational

Context: USDai.sol#L253-L255

Description: In the USDai contract's `_withdraw()` function, the minimum amount check is only performed when withdrawing non-base tokens through the swap adapter. When withdrawing the base token directly, the `withdrawAmountMinimum` parameter is ignored:

```
if (withdrawToken != address(_baseToken)) {
    uint256 baseTokenAmount = _unscale(usdaiAmount);
    _baseToken.approve(address(_swapAdapter), baseTokenAmount);
    withdrawAmount = _swapAdapter.swapOut(withdrawToken, baseTokenAmount, withdrawAmountMinimum, data);
} else {
    withdrawAmount = _unscale(usdaiAmount);  // withdrawAmountMinimum not checked
}
```

While the conversion from USDai to base token is deterministic due to the fixed scaling factor, for consistency and explicit behavior documentation, this should either be validated or documented.

Recommendation: Either add the validation:

```
  } else {
      withdrawAmount = _unscale(usdaiAmount);
+     if (withdrawAmount < withdrawAmountMinimum) revert SlippageExceeded();
  }
```

Or add a notice in the function documentation:

```
  /**
   * @notice Withdraw
   * @param withdrawToken Withdraw token
   * @param usdaiAmount USD.ai amount
   * @param withdrawAmountMinimum Minimum withdraw amount
+ * @dev withdrawAmountMinimum is only checked for non-base token withdrawals as base token
+ *      conversion is exact based on the scaling factor
  */
  function _withdraw(
```

The recommendation depends on the desired behavior:

1. Adding the check provides stronger guarantees and consistency.
2. Adding documentation makes the behavior explicit while avoiding gas costs for an almost-always-passing check.

MetaStreet: Fixed in commit 5bbf4d8c.

Cantina Managed: Fix verified.

3.4.10 Missing validation for token removal in `ChainlinkPriceOracle`

Severity: Informational

Context: ChainlinkPriceOracle.sol#L210-L213

Description: In the ChainlinkPriceOracle's `_setTokenPriceFeed()` function, when removing a token (setting its price feed to address(0)), the code does not validate whether the token exists in the set:

```
if (priceFeeds_[i] != address(0)) {
    /* Add token */
    _tokens.add(tokens_[i]);
} else {
    /* Remove token */
    _tokens.remove(tokens_[i]);  // Returns false if token doesn't exist
}
```

Attempting to remove a non-existent token fails silently as `_tokens.remove()` returns false without reverting. This could mask configuration errors where an admin attempts to remove a token that isn't actually registered.

Recommendation: Add explicit validation for token removal:

```diff
  if (priceFeeds_[i] != address(0)) {
      /* Add token */
      _tokens.add(tokens_[i]);
  } else {
      /* Remove token */
-     _tokens.remove(tokens_[i]);
+     bool removed = _tokens.remove(tokens_[i]);
+     if (!removed) revert TokenNotFound(tokens_[i]);
  }
```

Also add the error:

```
error TokenNotFound(address token);
```

This change ensures that attempting to remove non-existent tokens results in a clear error rather than failing silently.

MetaStreet: Fixed in commit ffb679ca.

Cantina Managed: Fix verified.