MetaStreet USDai Security Review

Prepared by: Kais Tlili, Independent Security Researcher
ktl@tlili.net

Date: May 1st to 14th, 2025

Contents

1. Introduction

`USDai` is an $M-backed stable coin that can be staked to mint `sUSDai`, which is an `IERC7540Redeem` vault implementing asynchronous redeems. Yield for `sUSDai` is generated by supplying liquidity to MetaStreet Pools and from $M emissions, which are converted to USDai before being distributed to staking users.

2. Scope

repo: https://github.com/metastreet-labs/metastreet-usdai-contracts
commit hash: e0d8fab7127459c0a3b84b1edb92cfc4f3bf4f9f

```
├── RedemptionLogic.sol
├── StakedUSDai.sol
├── StakedUSDaiStorage.sol
├── USDai.sol
├── interfaces
│   ├── IBasePositionManager.sol
│   ├── IERC7540.sol
│   ├── IERC7575.sol
│   ├── IMintableBurnable.sol
│   ├── IPoolPositionManager.sol
│   ├── IPriceOracle.sol
│   ├── IStakedUSDai.sol
│   ├── ISwapAdapter.sol
│   ├── IUSDai.sol
│   └── external
│       ├── IAggregatorV3Interface.sol
│       ├── IApproveAndCall.sol
│       ├── IPool.sol
│       ├── ISwapRouter02.sol
│       └── IWrappedMToken.sol
├── omnichain
│   ├── OAdapter.sol
│   ├── OToken.sol
│   └── OUSDaiUtility.sol
├── oracles
│   └── ChainlinkPriceOracle.sol
├── positionManagers
│   ├── BasePositionManager.sol
```

```
│   ├── PoolPositionManager.sol
│   └── PositionManager.sol
└── swapAdapters
    └── UniswapV3SwapAdapter.sol
```

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 mitigated without code changes 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: there are rare events but are theoretically possible under certain extreme but realistic market conditions

4. Executive Summary

Summary

Project Name	MetaStreet USDai
Repository	metastreet-usdai-contracts
Commit	e0d8fab
Type of Project	M0 Stablecoin, Staking, LayerZero
Review Timeline	May 1st to May 14th, 2025

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	1	1	0
Informational	2	0	2

5. Findings

5.1 Medium Risk

5.1.1 Lack of validation of `depositToken` in `OUSDaiUtility` could allow an attacker to grief other users messages

Severity: Medium risk

Description: When a user submits a composed cross chain message to `OUSDaiUtility`, `depositToken` is decoded from the incoming message but not constrained to be the same token as the whitelisted adapter that sent the message, from `OUSDaiUtility._deposit():

```
    function _deposit(uint256 depositAmount, bytes memory data) internal {

        (

            address depositToken,

            uint256 usdaiAmountMinimum,

            bytes memory path,

            SendParam memory sendParam,

            uint256 nativeFee

        ) = abi.decode(data, (address, uint256, bytes, SendParam, uint256));
```

Composed messages are not atomically executed when a message is received and `lzReceive()` is called on the adapter, from `OFTCore._lzReceive()`:

```
            // @dev Stores the lzCompose payload that will be executed in a
separate tx.

            // Standardizes functionality for executing arbitrary contract
invocation on some non-evm chains.

            // @dev The off-chain executor will listen and process the msg based
on the src-chain-callers compose options passed.
```

```
            // @dev The index is used when a OApp needs to compose multiple msgs
on lzReceive.

            // For default OFT implementation there is only 1 compose msg per
lzReceive, thus its always 0.

            endpoint.sendCompose(toAddress, _guid, 0 /* the index of the
composed message*/, composeMsg);
```

Once the composed message has been registered in the endpoint with `sendCompose()`, anyone (usually the executor) can call `Endpoint.lzCompose()` to execute the message in a separate transaction.

This could allow a malicious actor to grief other user's messages and freeze their funds on the home chain. For example, assuming whitelisted adapters are `USDC` and `USDT` and a victim sent a composed a message through the `USDT` adapter, an attacker could submit a message through the `USDC` adapter and supply `USDT` as `depositToken` and then race the call `Endpoint.lzCompose()` to execute his message before the victim's. The attacker would get the user's `USDT` while abandoning his USDC and breaking even but the subsequent call to `Endpoint.lzCompose()` to execute the victim's message will revert because `OUSDaiUtility` will not have enough `USDT`.

Currently the whitelisted adapters are `USDC` and `USDT`, both are stables with 6 decimals on Arbitrum. However it must be noted that if there was another whitelisted adapter for a token that is not a stable or is not 6 decimals, this issue could allow an attacker to steal user funds by sending a message through the higher decimal's token but providing `depositToken` as the lower decimal token.

MetaStreet: Fixed in commit [bcbcf665d5e1ad3fdfc129c5492c2d2766f2e8b1]

5.2 Low Risk

5.2.1 `PoolPositionManager.poolWithdraw()` slippage control might be ineffective in case of a race with a redemption fulfillment

Severity: Low risk

Description: When an admin calls `PoolPositionManager.poolWithdraw()` to withdraw liquidity from the pool and deposit tokens to `USDai`, he supplies `usdaiAmountMinimum` to control slippage on the swap from the underlying token to wrapped `$M`:

```
    function poolWithdraw(

        address pool,

        uint128 tick,

        uint128 redemptionId,

        uint256 usdaiAmountMinimum,

        bytes calldata data

    ) external onlyRole(STRATEGY_ADMIN_ROLE) nonReentrant returns (uint256) {

        /* Withdraw */

        (uint256 withdrawnShares, uint256 poolCurrencyAmount) =
IPool(pool).withdraw(tick, redemptionId);
```

The issue with the current slippage control is that for a partially redemption, `poolCurrencyAmount` returned by `Pool.withdraw()` can be larger than expected in case the call is front-ran with redemption fulfillment.

Consider the following scenario:

1. `sUSDai` has a 1000e18 redemption from `USDC` pool, only 500e18 are fulfilled.
2. Admin submits a call to `PoolPositionManager.withdraw()` with `usdaiAmountMinimum` set to `499e18`, expecting `poolCurrencyAmount` to be 500e18
3. Before the admin's transaction is executed, the redemption is fully fulfilled due to someone depositing or a loan repayment

4. Admin's transaction is then executed and the pool returns 1000e18 `USDC` to `sUSDai` while `usdaiAmountMinimum` is only 499e18.

MetaStreet: Fixed in commit [cbe8ba77243d5024ef77f848e55e11becb9cf9dc]

5.3 Informational

5.3.1 `exactInput()` might not consume all of the input amount leading to stuck tokens in `UniswapV3SwapAdapter`

Severity: Informational

Description: In some edge cases `exactInput()` might not consume all the input amount, this is extremely unlikely as it would only happen in case the user opens himself to sandwiching by supplying low slippage and there isn't enough liquidity in the Uniswap pool. The issue is that the unused tokens will remain stuck in `UniswapV3SwapAdapter`.

Proof of Concept: Copy paste the following into `test/usdai/Deposit.t.sol`:

```
    function test_adapter() public {
        uint256 amount = 40_000_000 ether; // there is only 20_000_000 ether in
pool

        uint256 usdBalance = usd.balanceOf(users.normalUser1);

        // User approves USDai to spend their USD
        vm.startPrank(users.normalUser1);
        usd.approve(address(usdai), amount);

        // User deposits 1000 USD into USDai
        uint256 mAmount = usdai.deposit(address(usd), amount, 0,
users.normalUser1);
        console.log("uniswapV3SwapAdapter bal after deposit %e",
usd.balanceOf(address(uniswapV3SwapAdapter)));
        // Assert user's USDai balance increased by mAmount
        assertEq(usdai.balanceOf(users.normalUser1), mAmount);

        // Assert user's USD balance decreased by amount
        assertEq(usd.balanceOf(users.normalUser1), usdBalance - amount);

        vm.stopPrank();
    }
```

Expected output:

```
[PASS] test_adapter() (gas: 25492254) Logs: uniswapV3SwapAdapter bal after
deposit 1.999699972497124720283982 2e25
```

MetaStreet: Acknowledged

5.3.2 Potential mispricing of pending shares from MetaStreet Pool redemptions could lead to inaccurate accounting in `StakedUSDai._assets()`

Severity: Informational

Description: In `StakedUSDai`, when calculating assets in `PoolPositionManager._assets()`, pending shares from all redemptions in each tick are summed and then priced at the current `depositSharePrice()` or `redemptionSharePrice()`:

```
    function _getPoolPosition(IPool pool, ValuationType valuationType) internal
view returns (uint256) {

        /* Get pool position */

        PoolPosition storage position =
_getPoolsStorage().position[address(pool)];



        /* Compute value across all ticks */

        uint256 value;

        for (uint256 i; i < position.ticks.length(); i++) {

            uint128 tick = uint128(position.ticks.at(i));



            /* Get shares */

            (uint256 shares,) = pool.deposits(address(this), tick);
```

```
            /* Get pending shares */
            uint256 pendingShares;
            for (uint256 j; j < position.redemptionIds[tick].length(); j++) {
                (uint128 pending,,) = pool.redemptions(address(this), tick,
uint128(position.redemptionIds[tick].at(j)));
                pendingShares += pending;
            }

            value += (
                valuationType == ValuationType.OPTIMISTIC
                    ? pool.depositSharePrice(tick) * (shares + pendingShares)
                    : pool.redemptionSharePrice(tick) * (shares + pendingShares)
            ) / FIXED_POINT_SCALE;
        }

        return value;
    }
```

The issue is that pending shares could already have been fulfilled at a price different from the current `depositSharePrice` or `redemptionSharePrice` creating a discrepancy between the share's reported and real price. This could have an impact if redeems are serviced by the admin while the discrepancy is significantly large.
The discrepancy is reconciled on withdrawals from the pool when the admin calls `PositionManager.poolWithdraw()`.

MetaStreet: Acknowledged