Origami Finance: SKY+

Security Review

By Jacopod Audits

X

Auditors

Lead Security Researcher: Jacobo Lansac (@Jacopod)

June 25, 2025

Contents

1 Introduction

A time-boxed review of Origami's **SKY+** vault with focus on the security aspects of the smart contracts.

2 About Jacopod

Jacopod Audits provides thorough security assessments for smart contract applications, combining deep manual review with advaced testing techniches like fuzzing and invariant testing to identify and mitigate potential vulnerabilities before deployment across diverse DeFi protocols. Here some highlights:

- All-time rank **#6** in Hats Finance competitions and bug bounties by number of issues found, hats Leaderboard.
- Secured protocols with more than $200M TVL in total:
  - Origami_fi, $111M TVL.
  - TempleDao, $41M TVL.
  - PinLinkAi, $21M TVL.
  - Vectorreserve, $45M peak TVL, dead project now.
- Check the Complete Audit Portfolio

Contact for audits:

- Telegram: @jacopod_eth
- X: @jacopod

3 About Origami Finance

Origami Finance is a novel tokenised leverage protocol on Ethereum and Berachain. Fully integrated with third-party lenders, Origami allows users to achieve non-custodial portfolio exposure to popular looping and other yield strategies via a familiar vault UX. Origami vaults are fully automated to eliminate tedious reward harvesting, maximise capital efficiency, and minimise liquidation risk for the underlying position.

4 Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time and resource-bound effort to find as many vulnerabilities as possible, but there is no guarantee that all issues will be found. This security review does not guarantee against a hack. Any modifications to the code will require a new security review.

Security researchers hold no responsibility for the findings provided in this document. A security review is not an endorsement of the underlying business or product and can never be taken as a guarantee that the protocol is bug-free. This security review is focused solely on the security aspects of the smart contracts. Gas optimizations are not the primary focus, but significant inefficiencies will also be reported.

5 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

5.1 Impact

- **High**: leads to a significant material loss of assets in the protocol or significantly harms a group of users.

- **Medium**: only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low**: can lead to unexpected behavior with some of the protocol's functionalities that are not so critical.

5.2 Likelihood

- **High**: almost certain to happen, easy to perform or highly incentivized.
- **Medium**: only conditionally possible or incentivized, but still relatively likely
- **Low**: requires multiple unlikely conditions, or little-to-no incentive

5.3 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

6 Executive Summary

Summary

Project Name	Origami Finance
Repository	TempleDAO/origami/
Commit	63e3628ff753dbe3ccea4330238cf8052dd3e7af
Type of Project	Leveraged Yield Farming
Audit Timeline	2025-06-17 - 2025-06-25
Methods	Manual Review, Testing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	1
Gas Optimizations	0
Informational	2

6.1 Files in scope

Files in scope	**nSLOC**	Notes
contracts/interfaces/external/sky/ISkyLockstakeEngine.sol	23	
contracts/interfaces/external/sky/ISkyStakingRewards.sol	12	
contracts/interfaces/external/sky/ISkyVat.sol	3	
contracts/investments/sky/OrigamiSuperSkyManager.sol	307	Main focus
contracts/investments/OrigamiDelegated4626Vault.sol	72	
Total	**417**	

6.2 Architecture review

6.2.1 Architecture description

The **SKY+** vault is an ERC4262 vault that automatically stakes user deposits into Sky Protocol's LockState system and compounds the earned rewards back into the vault, increasing share value over time.

Key notes:

- Super Sky Vault: an `OrigamiDelegated4626Vault` that accepts SKY tokens
- Manager: `OrigamiSuperSkyManager` that handles the funds deposited to the **SKY+** vault by locking them into the Sky echosistem via the Lockstake Engine
- Yield strategy: Deposits SKY tokens into Sky Protocol -> get LSSKY (locked SKY) -> allocate LSSKY to Sky Farms to earn rewards
- An overlord switches between farms, allocating the capital to the highest yield farm (yield-cliff of 0.5%).
- Incentivized compounding: the function to harvest rewards is permissionless, and the caller receives a % of the harvested rewards. Origami takes also a performance fee on it.
- When rewards are collected, they are transferred to a swapper contract, which integrates with cow-protocol. Reward tokens are then swapped for SKY, which is sent again to the SuperSkyManager and deposited again into the Lockstake engine, increasing the ratio of SKY assets per vault share.
- The fees are taking on the moment of harvesting (before swapping), which means they are taken in whatever reward token is the farm giving.
- All external functions in `OrigamiSuperSkyManager` have strict access control except for `deposit()` and `claimFarmRewards()`

6.2.2 Architecture assessment

- Very robust architecture leveraging ERC4626 standard, and a manager contract to isolate funds management.
- Very thorough testing suite

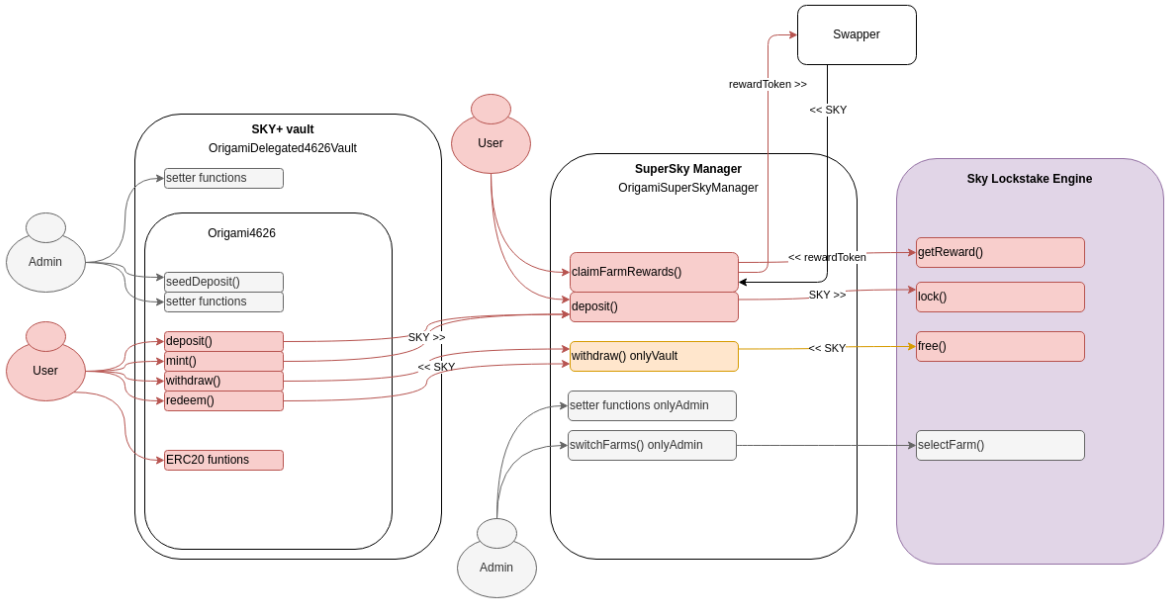

Figure 1: Architecture overview (Origami's)

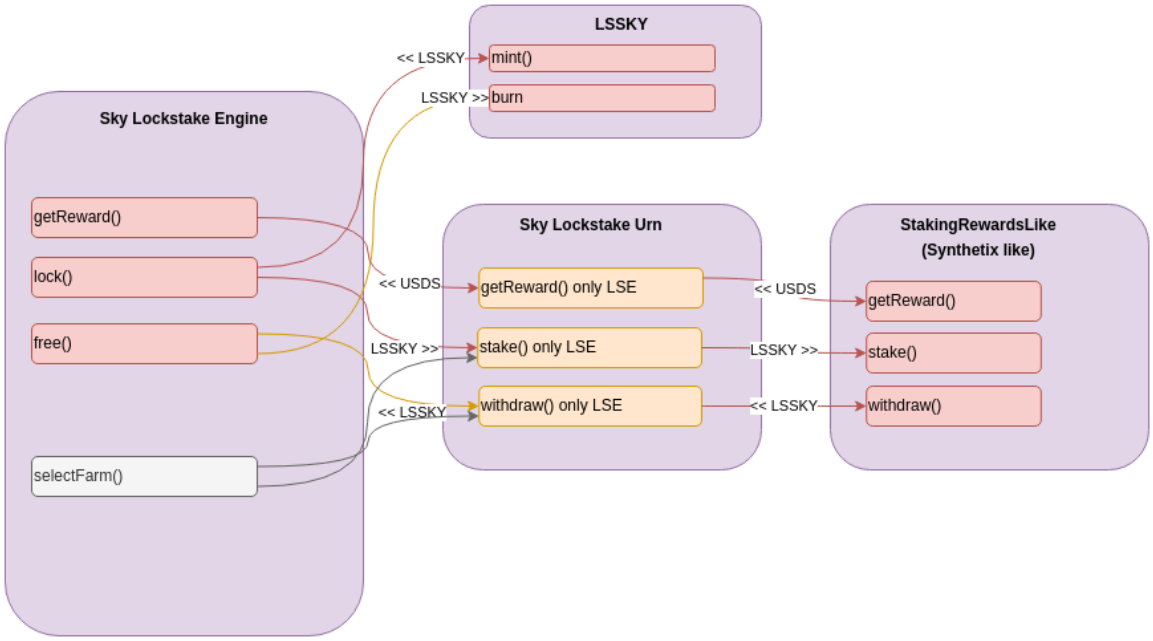

Figure 2: Architecture overview (Sky's components)

7 Findings

7.1 Low Risk

7.1.1 If `LockstakeEngine` implements a future withdrawal fee, the fee will be applied twice for depositors of `SKY+` vault

Summary

Sky's Lockstake Engine has a hardcoded withdrawal fee of 0%; however, the contract is upgradeable. Suppose a future implementation of the Lockstake Engine introduces a non-zero withdrawal fee. In that case, the `SKY+` vault will apply the fee twice, resulting in users receiving fewer assets than reported by the output of the `redeem()` and `withdraw()` functions.

The fee will be applied once by the Lockstake engine and again by the vault itself.

Detailed description

Let's take the `redeem()` function as an example following the funds.

When a user redeems, the fees are applied once when calling `_previewRedeem(shares, feeBps)`. The output value `assets` already accounts for the withdrawal fee:

OrigamiErc4626.sol:

```
    function redeem(
        uint256 shares,
        address receiver,
        address sharesOwner
    ) public virtual override nonReentrant returns (uint256) {
        uint256 feeBps = withdrawalFeeBps();
        uint256 maxShares = _maxRedeem(sharesOwner);
        if (shares > maxShares) {
            revert ERC4626ExceededMaxRedeem(sharesOwner, shares, maxShares);
        }

        // @audit the fee is applied here once, as _previewWithdraw reads the manager.withdrawalFees(),
        // which reads from the LSE fee.
        // here, shares will already be 99% of the original value
>>>     (uint256 assets, uint256 shareFeesTaken) = _previewRedeem(shares, feeBps);
        if (shareFeesTaken > 0) {
            emit InKindFees(FeeType.WITHDRAWAL_FEE, feeBps, shareFeesTaken);
        }

        // This function assumes that the full `assets` are withdrawn
>>>     _withdraw(_msgSender(), receiver, sharesOwner, assets, shares);

        // This return statement assumes that the full `assets` amount was received when calling
        ↪  `_withdraw()`
>>>     return assets;
    }
```

When `_withdraw()` is called, it requests that `assets` be withdrawn to the manager and assumes that the full amount will be withdrawn. However, this is not correct in the case of the SuperSkyManager, as the Lockstake Engine will also take a fee, so a smaller amount than `assets` will be received from the manager.

Root cause

When `_withdraw()` calls `_withdrawHook()` (which is overriden by `OrigamiDelegatedErc4626.sol`) it calls `withdraw()` on the manager:

OrigamiErc4626.sol:

```
    function _withdraw(
        address caller,
        address receiver,
        address sharesOwner,
        uint256 assets,
        uint256 shares
    ) internal virtual {
        if (receiver == address(0)) revert CommonEventsAndErrors.InvalidAddress(receiver);
        if (areWithdrawalsPaused()) revert CommonEventsAndErrors.IsPaused();

        if (caller != sharesOwner) {
            _spendAllowance(sharesOwner, caller, shares);
        }

        _burn(sharesOwner, shares);

        // If the vault has been fully exited, then reset the maxTotalSupply to zero, as if it were
        ↪  newly created.
        if (totalSupply() == 0) _maxTotalSupply = 0;

>>>     _withdrawHook(assets, receiver);

        // @audit the `assets` will only account for the fee applied once by `_previewRedeem()`.
        // However, it won't match the actual `assets` received by the user
>>>     emit Withdraw(caller, receiver, sharesOwner, assets, shares);
    }
```

OrigamiDelegated4626Vault:

```
    function _withdrawHook(
        uint256 assets,
        address receiver
    ) internal virtual override {
>>>     _manager.withdraw(assets, receiver);
    }
```

Now, the manager calls `LOCKSTAKE_ENGINE.free()`, and returns `assetsWithdrawn = assetsAmount` ignoring the actual amount of assets that were freed by `free()`.

```
    function withdraw(
        uint256 assetsAmount,
        address receiver
    ) external override onlyVault returns (uint256 assetsWithdrawn) {
>>>     assetsWithdrawn = assetsAmount == MAX_AMOUNT
            ? stakedBalance()
            : assetsAmount;

            // @audit this ignores the output from LockStake engine, which accounts for the fees
            // @audit instead, this function assumes that `assetsAmount` is fully freed
>>>     LOCKSTAKE_ENGINE.free(address(this), URN_INDEX, receiver, assetsWithdrawn);
    }
```

By looking at the implementation of `free()`, the output value can be less than the requested one when the `fee > 0`:

LockstakeEngine.sol:

```
>>> function free(address owner, uint256 index, address to, uint256 wad) external returns (uint256
↪  freed) {
        address urn = _getAuthedUrn(owner, index);
>>>     freed = _free(urn, wad, fee);
        sky.transfer(to, freed);
        emit Free(owner, index, to, wad, freed);
    }
```

In the end, the `redeem()` function will return the correct value (applying the fee once), but the actual amount of assets received by the user will have the fee applied twice:

- Once the fee is applied by the vault on `previewRedeem()`
- A second time, the fee is applied on `LockstakeEngine.free()`, which returns the actual freed amount.

Side effects

When `redeem()` is called, the `Withdraw()` event will emit the `assets` with the fee only applied once, but not the actual amount of assets received by the caller.

Impact

If a future implementation of `LockstakeEngine` has a withdrawal fee, the following will happen when calling `redeem()` and `withdraw()` from the **SKY+** vault:

- The user will receive fewer assets as the withdrawal fee will be applied twice.
- The return value from `redeem()` won't match the actual `assets` received (the output only accounts for the fee once so that the return value will be greater than the actual assets received)
- The `Withdraw` event emitted from `_withdraw()` will match the return value, but will be greater than the actual assets received by the user. This can lead to accounting problems in blockchain indexers like TheGraph.

Severity: low The likelihood of a future update increasing the withdrawal fee is very low.

Proof of code

A proof of code was provided as a PR, as it required mocking a LockstakeEngine with `vm.etch()`, because the `fee` is an immutable parameter hardcoded in the bytecode. https://github.com/TempleDAO/origami/pull/1827

Mitigation

- A more complex management of fees, passing the actual withdrawn amount from the manager to the vault.

Team response

Fixed in commit 19ff4be3cf85f256dee80e11c799f31fd814ebaa

The vault artificially increases the assets to be withdrawn from the Lockstake engine. This is a more straightforward fix of the problem, as only the `manager.withdraw()` needs an update, and no other contract is affected.

The only caveat is that `LockstakeEngine.free()` would revert if an attempt is made to withdraw the full position. However, that would only affect the last depositor, who could also withdraw smaller amounts until leaving some dust in the engine. The team is aware of the implications.

The following invariants were tested after the fix:

- The share price does not increase on withdrawals (ratio between vault shares and assets)
- If a user deposits and withdraws the full position, the total staked assets doesn't change (so a single user doesn't have control over the vault assets as a whole).

7.2 Informational

7.2.1 Pending rewards left unclaimed when switching farms

Context

When switching farms using `switchFarms()`, the current farm may have pending rewards that have not been claimed before the switch occurs. This leaves rewards in the old farm that can be claimed later through the `claimFarmRewards()` function.

Root cause

Location: `contracts/investments/sky/OrigamiSuperSkyManager.sol` in function `switchFarms()`.

The function switches farms without harvesting pending rewards from the current farm, potentially leaving rewards unclaimed.

```
function switchFarms(uint32 newFarmIndex) external override onlyElevatedAccess returns (
    uint256 amountWithdrawn,
    uint256 amountDeposited
) {
    // @audit-info consider claiming rewards here before leaving the farm to leave a clean state in the
    ↪  farm
    // ...
    LOCKSTAKE_ENGINE.selectFarm(
        address(this),
        URN_INDEX,
        address(newFarm.staking),
        newFarm.referral
    );
    // ...
}
```

Impact

No significant impact as the `claimFarmRewards()` function allows claiming from multiple farms at once, including farms where there are no currently staked tokens.

Mitigation

It is recommended to call `_harvestRewards()` on the current farm before switching to leave the farm in a clean state without pending rewards.

Note: This may require modifying `_harvestRewards()` to handle the case where farmIndex=0.

Team response

Fixed in commit `50150e68a726c7b0088defb290cf099036829a7b`. Rewards are harvested before switching farms.

7.2.2 Farm removal can be temporarily DOS'd by external deposits

Context

The `removeFarm()` function can only remove a farm when there are no pending rewards (`staking.earned(URN_ADDRESS) == 0`). However, anyone can deposit tokens on behalf of this contract into the Lockstake Engine, potentially creating pending rewards and preventing farm removal.

Root cause

Location: `contracts/investments/sky/OrigamiSuperSkyManager.sol` in function `removeFarm()`.

The function checks if there are pending rewards before allowing farm removal, but this check can be manipulated by external actors depositing on behalf of the contract.

```
function removeFarm(uint32 farmIndex) external override onlyElevatedAccess {
    // ...
    if (staking.earned(URN_ADDRESS) > 0) {
        revert FarmStillInUse(farmIndex);
    }
    // ...
}
```

Here is the `LockStakeEngine.loc()`, which allows locking on behalf of another account:

```
    function lock(address owner, uint256 index, uint256 wad, uint16 ref) external {
        address urn = _getUrn(owner, index);
        sky.transferFrom(msg.sender, address(this), wad);
        require(wad <= uint256(type(int256).max), "LockstakeEngine/overflow");
        address voteDelegate = urnVoteDelegates[urn];
        if (voteDelegate != address(0)) {
            sky.approve(voteDelegate, wad);
            VoteDelegateLike(voteDelegate).lock(wad);
        }
        vat.slip(ilk, urn, int256(wad));
        vat.frob(ilk, urn, urn, address(0), int256(wad), 0);
        lssky.mint(urn, wad);
        address urnFarm = urnFarms[urn];

        if (urnFarm != address(0)) {
            // OK: if the urn farm is DELETED or UNSUPPORTED, this lock will revert
            require(farms[urnFarm] == FarmStatus.ACTIVE, "LockstakeEngine/farm-deleted");
            LockstakeUrn(urn).stake(urnFarm, wad, ref);
        }
        emit Lock(owner, index, wad, ref);
    }

    function _getUrn(address owner, uint256 index) internal view returns (address urn) {
        urn = ownerUrns[owner][index];
        require(urn != address(0), "LockstakeEngine/invalid-urn");
    }
```

Impact

The removal of a farm can be delayed by an attacker depositing 1 wei right before the removal of the farm. However, the attack is easy to circumvent as admins can claim rewards and remove the farm in the same transaction to avoid the DOS.

Mitigation

It is not worth fixing the code, as there is probably no economic incentive to delay the removal of a farm, so the probability is very low. Also, bypassing this issue by crafting transactions that claim rewards and remove the farm atomically is very easy.

This is just an informational issue to make the team aware of the problem.