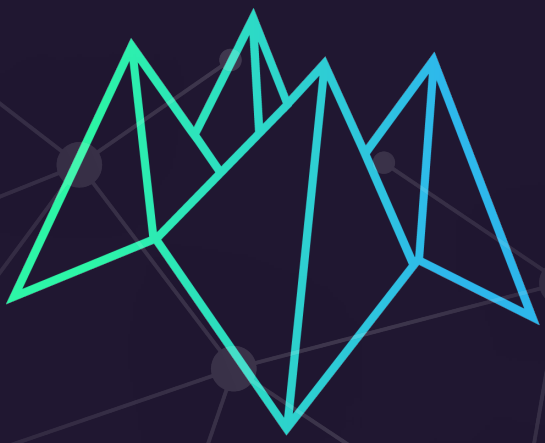
SB SECURITY

Origami Super Savings USDS Security Review



Oct 5, 2024

Conducted by:
Blckhv, Lead Security Researcher
Slavcheww, Lead Security Researcher

Contents

1. About SBSecurity

SBSecurity is a duo of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

	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 significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

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

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

Origami Super Savings USDS is an ERC-4626 vault that takes user deposits and automatically switches between Staked USDS staking and USDS staking in any currently available SKY farm depending on which currently has the higher APR. The vault will utilise CoWswap integration to swap SKY rewards back to USDS to auto-compound the rewards back into the underlying.

Overview

Project	Origami Super Savings USDS
Repository	https://github.com/TempleDAO/origami-public
Commit Hash	6a88b1c8f2cda15584cd2c07013fb98d2e7e8642
Resolution	97b715e0f1068947e6a9aae5815d183b6601240e
Timeline	September 22 - October 3, 2024

Scope

OrigamiErc4626.sol
OrigamiCowSwapper.sol
OrigamiSuperSavingsUsdsManager.sol
OrigamiSuperSavingsUsdsVault.sol

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low/Info Risk	7

5. Findings

5.1. Medium severity

5.1.1. Vault with deposit fee is prone to inflation attack

Severity: Medium Risk

Description: When `OrigamiERC4626` is used with only a **deposit fee the** first depositor can cause the second one to lose his deposited assets. If there is a deposit fee, let's say 50 bps first user will deposit 2 wei and will receive 1 share. The second deposit with 1000e18 tokens is queued and the first user performs frontrun with a direct donation of the same amount because `totalAssets` relies on `balanceOf`:

```
function _convertToShares(uint256 assets, OrigamiMath.Rounding rounding) internal view virtual returns
(uint256) {
    uint256 _totalSupply = totalSupply();

    return _totalSupply == 0
        ? assets.scaleUp(_assetsToSharesScalar)
        : assets.mulDiv(_totalSupply + DECIMALS_OFFSET_SCALAR, totalAssets() + 1, rounding);
        //10000e18 * (1 + 1) / (10000e18 + 2 + 1) = 1.99 = 1
}
```

`_convertToShares` of the second user returns 1 shares. After the fees are applied it will return 0.

```
function _previewDeposit(uint256 assets, uint256 feeBps) internal virtual view returns (
    uint256 shares,
    uint256 shareFeesTaken
) {
    shares = _convertToShares(assets, OrigamiMath.Rounding.ROUND_DOWN);

    // Deposit fees are taken from the shares in kind
    (shares, shareFeesTaken) = shares.splitSubtractBps(feeBps, OrigamiMath.Rounding.ROUND_DOWN);
    // 1 * 9950 / 10000
}
```

The first user withdraws his funds and receives his initial deposit back, while the funds of the second user are locked in the vault forever.

Recommendation: Do not allow deposit that will mint 0 shares, that will protect the second user from the inflation, and when there are assets in the vault the `share:assets` ratio should not be 1:1.

Resolution: Fixed

5.2. Low/Info severity

5.2.1. addFarm should use nextFarmIndex

Severity: Low Risk

Description: When a new farm is added, the for loop always iterates to `MAX_FARMS`, instead of until the `maxFarmIndex`, the index of the last added farm.

```
function addFarm(
    address stakingAddress,
    uint16 referralCode
) external override onlyElevatedAccess returns (
    uint32 nextFarmIndex
) {
    // Farm index starts at 1
    nextFarmIndex = maxFarmIndex + 1;
    if (nextFarmIndex > MAX_FARMS) revert MaxFarms();

    // Use removeFarm to delete
    if (address(stakingAddress) == address(0)) revert InvalidFarm(nextFarmIndex);

    // Check this farm isn't already setup
    for (uint256 i; i < MAX_FARMS; ++i) {
        if (address(_farms[i].staking) == stakingAddress) revert FarmExistsAlready(stakingAddress);
    }
```

Loops after `maxFarmIndex` are excessive since we are sure there are no farms after that index.

Recommendation: Replace `MAX_FARMS` with `maxFarmIndex`

Resolution: Fixed

5.2.2. No way to provide referral to sUSDS deposits

Severity: Low Risk

Description: sUSDS, similar to SKY farms also allows passing a referral, but `_depositIntoSavings` doesn't have such an option:

```
function _depositIntoSavings(uint256 assetsAmount) private returns (uint256 amountDeposited) {
    amountDeposited = assetsAmount == MAX_AMOUNT
        ? USDS.balanceOf(address(this))
        : assetsAmount;

    if (amountDeposited > 0) {
        sUSDS.deposit(amountDeposited, address(this));
    }
}
```

```
function deposit(uint256 assets, address receiver, uint16 referral) external returns (uint256 shares) {
    shares = deposit(assets, receiver);
    emit Referral(referral, receiver, assets, shares);
}
```

Recommendation: If you have a referral, allow passing it on to savings vault deposits.

Resolution: Fixed

5.2.3. Order expiration calculation is wrong in some cases

Severity: Low Risk

Description: When `order.validTo` is calculated, it adds `MIN_ORDER_EXPIRY_SECONDS` (90 seconds) to `block.timestamp` to prevent cases where the function is executed at a time that is close to the `expiryPeriodSecs` divisor.

But since CowProtocol has an order timeout limit of 60 seconds, it won't call functions that have 60 seconds left.

Although due to `MIN_ORDER_EXPIRY_SECONDS` which is 90 seconds, if the order was executed by CoWProtocol within those last 90 seconds but before the last 60 seconds, `validTo` will be different for the executed order and give a different hash while the order must be valid.

```
function _calcOrderExpiry(uint32 expiryPeriodSecs) internal view returns (uint32) {
    // slither-disable-next-line divide-before-multiply
    return (
        (uint32(block.timestamp + MIN_ORDER_EXPIRY_SECONDS) / expiryPeriodSecs) * expiryPeriodSecs
    ) + expiryPeriodSecs;
}
```

Recommendation: Consider remove the `MIN_ORDER_EXPIRY_SECONDS` which will make the expiration period multiple of `expiryPeriodSecs`.

Resolution: Fixed

5.2.4. Rewards are not harvested before updating the performance fees and on farm removals

Severity: Low Risk

Description: When performance fees are updated, no harvest is done, meaning the updated fees will be applied to the previously accrued rewards. While this doesn't lead to loss of funds to anyone involved, it is important to update all the params that depend on the value that will be changed.

Recommendation: Harvest the rewards for the current active farm.

```
function setPerformanceFees(uint48 callerFeeBps, uint48 origamiFeeBps) external override
onlyElevatedAccess {
      uint48 newTotalFee = callerFeeBps + origamiFeeBps;

      // Only allowed to decrease the total fee or change allocation
      uint48 existingTotalFee = _performanceFeeBpsForCaller + _performanceFeeBpsForOrigami;
      if (newTotalFee > existingTotalFee) revert CommonEventsAndErrors.InvalidParam();

+     if(currentFarmIndex != 0) {
+        HarvestRewardCache memory cache = HarvestRewardCache({
+           swapper: swapper,
+           caller: msg.sender,
+           feeCollector: feeCollector,
+           feeBpsForCaller: _performanceFeeBpsForCaller,
+           feeBpsForOrigami: _performanceFeeBpsForOrigami
+       });
+        _harvestRewards(currentFarmIndex, _getFarm(currentFarmIndex), cache);
+     }
      emit PerformanceFeeSet(newTotalFee);
      _performanceFeeBpsForCaller = callerFeeBps;
      _performanceFeeBpsForOrigami = origamiFeeBps;
  }
```

Resolution: Fixed

5.2.5. `_maxDeposit` rounds in wrong direction

Severity: Low Risk

Description: In `_maxDeposit` rounding used is down. This is wrong since we always have to round in favor of the protocol and now slightly fewer assets will be able to be deposited.

According to the EIP that function can underestimate `maxAssets` if necessary but there is no need to do so.

Recommendation: Consider rounding UP instead of DOWN in, so more assets can be deposited.

```
function _maxDeposit(uint256 feeBps) internal view returns (uint256 maxAssets) {
      uint256 _maxTotalSupply = maxTotalSupply();
      if (_maxTotalSupply == type(uint256).max) return type(uint256).max;

      uint256 _totalSupply = totalSupply();
      if (_totalSupply > _maxTotalSupply) return 0;

      uint256 availableShares;
      unchecked {
          availableShares = _maxTotalSupply - _totalSupply;
      }

      return _convertToAssets(
-         availableShares.inverseSubtractBps(feeBps, OrigamiMath.Rounding.ROUND_DOWN),
+         availableShares.inverseSubtractBps(feeBps, OrigamiMath.Rounding.ROUND_UP),
-         OrigamiMath.Rounding.ROUND_DOWN
+         OrigamiMath.Rounding.ROUND_UP
      );
  }
```

Resolution: Fixed

5.2.6. updateAmountsAndPremiumBps should not allow updating limitPricePremiumBps if limitPriceOracle is 0

Severity: Low Risk

Description: `updateAmountsAndPremiumBps()` allows updating `maxSellAmount`, `minBuyAmount` and `limitPricePremiumBps`. But initially `limitPricePremiumBps` can only be configured if `limitPriceOracle ≠ address 0`. But here it can be set even if `limitPriceOracle` is address 0

```
function updateAmountsAndPremiumBps(
    address sellToken,
    uint96 maxSellAmount,
    uint96 minBuyAmount,
    uint16 limitPricePremiumBps
) external override onlyElevatedAccess {
    if (maxSellAmount == 0) revert CommonEventsAndErrors.ExpectedNonZero();
    if (minBuyAmount == 0) revert CommonEventsAndErrors.ExpectedNonZero();

    // Ensure it's configured first.
    OrderConfig storage config = _getOrderConfig(IERC20(sellToken));

    config.maxSellAmount = maxSellAmount;
    config.minBuyAmount = minBuyAmount;
    config.limitPricePremiumBps = limitPricePremiumBps; <--------------------

    emit OrderConfigSet(sellToken);
}
```

Recommendation: Add a check like in `setOrderConfig()` for `limitPricePremiumBps`.

Resolution: Fixed

5.2.7. `_depositIntoFarm` uses type max instead of MAX_AMOUNT

Severity: Information Risk

Description: `_depositIntoFarm` uses `type(uint256).max` instead of `MAX_AMOUNT`, unlike savings deposit and withdrawal functions:

```
function _depositIntoFarm(uint256 assetsAmount, uint32 farmIndex) private returns (uint256
amountDeposited) {
    Farm storage farm = _getFarm(farmIndex);

    amountDeposited = assetsAmount == type(uint256).max
        ? USDS.balanceOf(address(this))
        : assetsAmount;
}
```

Recommendation: Although this is just a code improvement and does not represent vulnerability, consider using `MAX_AMOUNT` in `_depositIntoFarm` as well:

Resolution: Fixed