



## SMART CONTRACTS REVIEW



May 15th 2024 | v. 1.0

# Security Audit Score

**PASS**

Zokyo Security has concluded that this smart contract passed a security audit.



# # ZOKYO AUDIT SCORING STEADEFI

## 1. Severity of Issues:

- Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
- High: Important issues that can compromise the contract in certain scenarios.
- Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
- Low: Smaller issues that might not pose security risks but are still noteworthy.
- Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.

2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.

3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.

4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.

5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.

6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

# HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:

- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:

- 1 Critical issue: 1 resolved = 0 points deducted
- 2 High issues: 0 points deducted
- 6 Medium issues: 1 acknowledged and 5 resolved = - 3 points deducted
- 8 Low issues: 1 acknowledged and 7 resolved = - 1 points deducted
- 6 Informational issues: 6 resolved = 0 points deducted

Thus,  $100 - 3 - 1 = 96$

# TECHNICAL SUMMARY

This document outlines the overall security of the Steadefi smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Steadefi smart contract/s codebase for quality, security, and correctness.

## Contract Status



LOW RISK

There was 1 critical issue found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Steadefi team put in place a bug bounty program to encourage further active analysis of the smart contract/s.



# Table of Contents

Auditing Strategy and Techniques Applied	5
Executive Summary	7
Structure and Organization of the Document	8
Complete Analysis	9

# AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Steadefi repository:

Repo: <https://arbiscan.io/address/0x68915861F9444AcB0Ffea0cC2BCa71eD2455F25A#code>

Last commit with fixes - <https://gist.github.com/jefflam/c9cd85a8448d223660f5e233336921f4>

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- LendingVault.sol

## **During the audit, Zokyo Security ensured that the contract:**

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Steadefi smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

- |    |  |    |  |
|----|--|----|--|
| 01 | Due diligence in assessing the overall code quality of the codebase.       | 03 | Thorough manual review of the codebase line by line. |
| 02 | Cross-comparison with other, similar smart contract/s by industry leaders. | 04 | Thorough manual review of the codebase line by line. |





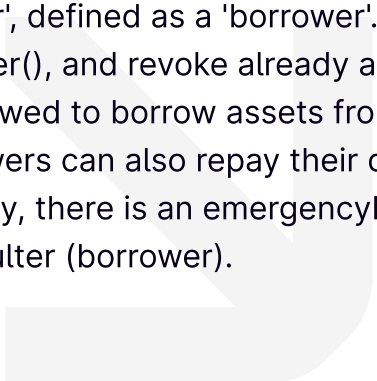
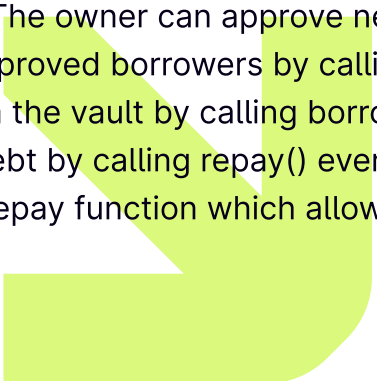
# Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contract submitted for auditing is well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

The LendingVault.sol smart contract from the Steadefi protocol is, as its name implies, a vault contract that has an underlying asset that can be set as a native asset or not.

Any user can deposit native tokens without any restriction if the vault is deployed as a native one. If not, they can deposit ERC20 assets' tokens in exchange for vault shares. Users can deposit assets when the vault is not paused, which could be triggered by the owner by executing functions restricted to the owner. On the other hand, users can withdraw the deposited assets by returning vault shares without any restrictions; they can withdraw even if the vault is paused.

The vault contains a 'special actor', defined as a 'borrower'. The owner can approve new borrowers by calling `approveBorrower()`, and revoke already approved borrowers by calling `revokeBorrower()`. Borrowers are allowed to borrow assets from the vault by calling `borrow()` when the vault is not paused. Borrowers can also repay their debt by calling `repay()` even when the vault is paused. Additionally, there is an `emergencyRepay` function which allows any user to repay the debt of a defaulter (borrower).



# STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Steadefi team and the Steadefi team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:



## **Critical**

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.



## **High**

The issue affects the ability of the contract to compile or operate in a significant way.



## **Medium**

The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior.



## **Low**

The issue has minimal impact on the contract's ability to operate.



## **Informational**

The issue has no impact on the contract's ability to operate.

# COMPLETE ANALYSIS

## FINDINGS SUMMARY

#	Title	Risk	Status
1	VAULT IS VULNERABLE TO INFLATION ATTACK	Critical	Resolved
2	BORROWERS CAN DRAIN THE VAULT FREELY	Medium	Acknowledged
3	ETH CAN GET STUCK IN THE CONTRACT	Medium	Resolved
4	PROTOCOL CAN BECOME UNUSABLE DUE TO DOS	Medium	Resolved
5	FRONT-RUNNING USERS IS POSSIBLE BY THE PROTOCOL TO OBTAIN A HIGHER FEE	Medium	Resolved
6	PRECISION LOSS	Medium	Resolved
7	performanceFee CAN LEAD TO DENIAL OF SERVICE (DOS)	Medium	Resolved
8	CHECK-EFFECTS-INTERACTIONS PATTERN NOT FOLLOWED	Low	Acknowledged
9	INTEREST RATE CALCULATION DOES NOT COVER COMMENTED CASE	Low	Resolved
10	RESTRICTED MODIFIER IS RECOMMENDED TO BE USED ONLY IN EXTERNAL FUNCTIONS	Low	Resolved
11	ENUMERABLESET'S FUNCTIONS `add()` and `remove()` RETURNED VALUES ARE NOT CHECKED	Low	Resolved
12	Missing Sanity checks for constructor parameters	Low	Resolved
13	Redundant checks	Low	Resolved
14	depositNative() does not check for isNativeAsset	Low	Resolved
15	Kink1 can be greater than kink2	Low	Resolved

#	Title	Risk	Status
16	UNNECESSARY CHECKS	Informational	Resolved
17	USE UNCHECKED ARITHMETIC	Informational	Resolved
18	CONSIDER USING IMMUTABLE VARIABLES	Informational	Resolved
19	CATCH ARRAY LENGTH	Informational	Resolved
20	USE ++i INSTEAD OF i++	Informational	Resolved
21	CALLING EXTERNAL FUNCTIONS CONSUME LESS GAS THAN PUBLIC	Informational	Resolved

## VAULT IS VULNERABLE TO INFLATION ATTACK

The `depositNative` and `deposit` functions allow users to deposit asset in exchange for shares of the LendingVault. The `mintShares` function is the one that calculates and mints the corresponding amount of shares to the users.

The calculation of the corresponding amount of shares depends on on-chain and manipulable factors like the amount of assets hold in the vault. In other words, it is vulnerable to inflation attack.

```
function _mintShares(uint256 assetAmt) internal returns (uint256) {
    uint256 _shares;

    if (totalSupply() == 0) {
        _shares = assetAmt * _to18ConversionFactor();
    } else {
        _shares = assetAmt * totalSupply() / (totalAsset() - assetAmt);
    }

    // Mint LvToken to user equal to liquidity share amount
    _mint(msg.sender, _shares);

    return _shares;
}
```

Consider this scenario:

- The vault has just been deployed so that totalAssets() = 0 and totalSupply() = 0.
- Attacker deposit 1 wei of asset so now totalAsset() = 1 and attacker gets minted 1 wei of shares, totalSupply() is equal to 1 wei now also. Attacker now owns 100% of the vault's supply.
- Victim now execute a deposit of 2\_000e18 assets.
- Attacker front-runs the victim and directly transfers 20\_000e18 assets to the pool.
- The calculation of the victim's shares would now be:

$$2\_000e18 * 1 / ((20\_000e18 + 1) - 2\_000e18) = 0$$

As a result the victim gets 0 shares minted and now the attacker can withdraw all the funds as he owns 100% of the supply.

Proof of concept:

```
function testPocInflationAttack() public {
    deal(bob, 10 ether);
    uint256 attackerBalanceBeforeAttack = address(bob).balance;
    vm.startPrank(bob);
    vault.depositNative{value: 1}(1, 0); // Attacker deposit 1 wei

    uint256 totalSupply = vault.totalSupply();
    uint256 totalAssets = vault.totalAsset();
    assert(totalAssets == 1 && totalSupply == 1); // TotalAssets and supply = 1

    // Attacker front-runs victim's tx
    uint256 inflatedAmount = 149275995181204;
    asset.transfer(address(vault), inflatedAmount);
    uint256 totalAssets2 = vault.totalAsset();
    assert(totalAssets2 == inflatedAmount + 1);
    vm.stopPrank();

    // Victim now deposits
    vm.startPrank(jim);
    deal(jim, 10 ether);
    uint256 victimDepositAmount = inflatedAmount / 2;
    vault.depositNative{value: victimDepositAmount}(victimDepositAmount, 0);
    uint256 victimShares = vault.balanceOf(jim);
    assert(victimShares == 0);
    vm.stopPrank();

    // Attacker withdraws the whole balance
    vm.startPrank(bob);
    uint256 attackersShares = vault.balanceOf(bob);
    vault.withdraw(attackersShares, 0);
    uint256 attackerBalanceAfterAttack = address(bob).balance;

    // We check that the difference in balance is the attacker's deposited amount +
    // victim's deposited amount
    assert(attackerBalanceAfterAttack - attackerBalanceBeforeAttack ==
    victimDepositAmount + inflatedAmount);
}
```

### Recommendation:

UniswapV2 solved this vulnerability by sending a considerable amount of shares to the zero address when `totalSupply() = 0`. (<https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol#L119-L124>). The same solution can be applied to this case. Adding a require to not allow minting 0 shares is also recommended.

There also other options for solving the vulnerability (<https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks>).

## BORROWERS CAN DRAIN THE VAULT FREELY

Borrowers are allowed to borrow assets but they are not paying any collateral so that they can leave with the asset and never pay their debt back. The borrow() function is only callable by users set as borrowers so that any user set as borrower can drain the vault easily and freely.

This can lead to scenarios where any single authorized borrower can run away with all the funds of all the lenders or depositors. For example, let's say there are 2 depositors- Alice and Jim. Alice deposits  $10 \times 10^{18}$  of Native tokens while Jim deposits  $1 \times 10^{18}$  Native tokens. Then it is possible that an authorized borrower say Bob borrows all of  $11 \times 10^{18}$  tokens and runs away.

- borrow(uint256 borrowAmt):

```
function borrow(uint256 borrowAmt) external nonReentrant whenNotPaused onlyBorrower {
    if (borrowAmt == 0) revert Errors.InsufficientBorrowAmount();
    if (borrowAmt > totalAvailableAsset()) revert Errors.InsufficientLendingLiquidity();

    // Update vault with accrued interest and latest timestamp
    _updateVaultWithInterestsAndTimestamp(0);

    // Calculate debt amount
    uint256 _debt = totalBorrows == 0 ? borrowAmt : borrowAmt * totalBorrowDebt /
totalBorrows;

    // Update vault state
    totalBorrows = totalBorrows + borrowAmt;
    totalBorrowDebt = totalBorrowDebt + _debt;

    // Update borrower state
    Borrower storage borrower = borrowers[msg.sender];
    borrower.debt = borrower.debt + _debt;
    borrower.lastUpdatedAt = block.timestamp;

    // Transfer borrowed token from vault to manager
    asset.safeTransfer(msg.sender, borrowAmt);
    emit Borrow(msg.sender, _debt, borrowAmt);
}
```



The described scenario is directly affecting the `withdraw()` functionality. If a borrower drains the vault freely then the rest of the users will not be able to withdraw any asset by calling the `withdraw()` function.

The described scenario would also impact on the amount of shares calculation for user's minting.

```
_shares = assetAmt * totalSupply() / (totalAsset() - assetAmt);
```

`totalAsset()` is used as denominator in the operation and `totalAsset()` is:

```
function totalAsset() public view returns (uint256) {  
    return totalBorrows + _pendingInterests(0) + totalAvailableAsset();  
}
```

As it can be seen `totalAsset()` is not only the actual amount of assets in the vault (`totalAvailableAsset()`) but the addition of it with every pending interest and the total amount of borrows (`totalBorrows`). This means that if a borrower can borrow unlimited amount without any collateral he can freely inflate `totalAsset()` and as a consequence reduce the amount of shares calculated.

Proof of concept:

```
function testBorrowTheWholeVaultFreely() public {  
    vm.startPrank(address(alice));  
    vault.approveBorrower(address(bob)); // Approve Bob as borrower  
    vm.stopPrank();  
  
    deal(jim, 10 ether);  
    vm.startPrank(jim);  
    vault.depositNative{value: 1 ether}(1 ether, 0); // User deposit 1 ether  
    vm.stopPrank();  
  
    vm.startPrank(bob);  
    uint256 balanceBefore = IERC20(asset).balanceOf(bob);  
    vault.borrow(1 ether); // bob borrows 1 ether and does not pay it back  
    uint256 balanceAfter = IERC20(asset).balanceOf(bob);  
  
    assert(balanceAfter > balanceBefore);  
    assert(balanceAfter == balanceBefore + 1 ether);  
}
```



**Recommendation:**

The borrow function must implement a mechanism that forces the borrower to deposit a collateral when borrowing assets. This collateral should be kept by the contract/protocol until borrowers pay their debt back. Ideally the collateral should be able to get liquidated, enforcing the borrower to keep a healthy position.

It is advised to add robust and concrete rules that govern borrowing of the assets. This can be done by adding more rules in the smart contract or writing a Borrow smart contract that governs and securely manages borrowing from the Vault including the borrow() function. In addition to that it is advised to decentralize the calling of approveBorrower() function further by the usage of multisigs.

## ETH CAN GET STUCK IN THE CONTRACT

The contract implements a `receive()` function. The purpose of this receive function is to receive the ETH as a result of a `withdraw()` from its wrapped version, they confirm it in the NatSpec comment: `Fallback function to receive native token sent to this contract, needed for receiving native token to contract when unwrapped`. The problem with this is that they added: `if (!isNativeAsset) revert Errors.OnlyNonNativeDepositToken();` This check ensures that the receive function is only callable when the vault's variable `isNativeAsset` has been set to true but it does not ensure that all the ETH received comes from the result of executing `withdraw()` in the wrapped version.

```
receive() external payable {
    if (!isNativeAsset) revert Errors.OnlyNonNativeDepositToken();
}
```

This scenario is also possible:

1. `isNativeAsset` has been set to true.
2. The contract is able to receive ETH as a result of `withdraw()` from the native token wrapped version, this is correct.
3. Any user can manually or as a result of an external integration send ETH to the contract. This ETH sent is getting stuck and can not be withdrawn.

Proof of concept:

- 1st: deploy a native vault:

```
function setUp() public {
    counter = new Counter();
    counter.setNumber(0);

    SteadefiAccessManager accessManagerContract = new SteadefiAccessManager(alice);
    asset = IERC20(0x82aF49447D8a07e3bd958D0d56f35241523fBab1); // WETHER in ARB
    vault = new LendingVault(
        "VAULT",
        "VLT",
        asset,
        true, // set as native
        10000,
        1000 ether,
        bob,
        address(accessManagerContract)
    );
}
```

- 2nd: directly send ETH

```
function testSendEtherToVault() public {  
    deal(address(vault), 10 ether);  
    assert(address(vault).balance == 10 ether);  
}
```

### Recommendation:

Instead of using this check `if (!isNativeAsset) revert Errors.5();` in the receive() function, use `if (msg.sender != address(asset) revert Errors.OnlyNonNativeDepositToken())`.

## PROTOCOL CAN BECOME UNUSABLE DUE TO DOS

There is a problem with the use of for loops in `_pendingInterests` functions. The loops go through `_approvedBorrowers` array so if this array grows enough to txs run out of gas then there is a problem. Specially with `_pendingInterests` function as it is used in `_updateVaultWithInterestsAndTimestamp` which is used in almost every important function of the protocol so the impact will be an entire DOS of the protocol. The `borrowAPR()` function also implements the same loop but it will not consume gas as it is a view function that is not called internally in the contract by a non view one.

- `_pendingInterests(uint256 assetAmt)`:

```
function _pendingInterests(uint256 assetAmt) internal view returns (uint256) {
    uint256 pendingInterests_;

    for (uint256 i = 0; i < _approvedBorrowers.length(); i++) {
        pendingInterests_ += _pendingInterest(_approvedBorrowers.at(i), assetAmt);
    }

    return pendingInterests_;
}
```

### Recommendation:

Add a maximum length check in `approveBorrower()` function to do not allow adding more than x borrowers to avoid having an extra large array that may run transactions out of gas. Be aware that if at some point transactions start running out of gas then some borrowers would have to get revoked.

## FRONT-RUNNING USERS IS POSSIBLE BY THE PROTOCOL TO OBTAIN A HIGHER FEE

The performanceFee variable is a key variable to calculate the fee obtained by the protocol. This variable can be changed by calling the `updatePerformanceFee` function by the `restricted` user/users.

```
function updatePerformanceFee(uint256 newPerformanceFee) external restricted {
    _updateVaultWithInterestsAndTimestamp(e);

    performanceFee = newPerformanceFee;

    emit PerformanceFeeUpdated(msg.sender, performanceFee, newPerformanceFee);
}
```

If a user executes a transaction, the user marked as `restricted` can front-run it and change the performanceFee to a higher value resulting on the protocol earning more fees, this could take place specially on large transaction with a huge amount of funds.

### Recommendation:

Do not allow performanceFee to be changed or add a grace period where the change is not directly applied.

## PRECISION LOSS

The `lendingAPR()` function which returns the current lendingAPR, calculated as  $\text{borrowAPR} * \text{utilizationRate} * (1 - \text{performanceFee})$  implements a division before a multiplication in the formula which can lead to rounding error as Solidity rounds down decimals.

The used formula:

```
return borrowAPR_ * utilizationRate_  
    / SAFE_MULTIPLIER  
    * ((1 * SAFE_MULTIPLIER) - performanceFee)  
    / SAFE_MULTIPLIER;
```

### Recommendation:

Execute every multiplication before divisions, the new formula would be:

```
return borrowAPR_ * utilizationRate_  
    * ((1 * SAFE_MULTIPLIER) - performanceFee)  
    / SAFE_MULTIPLIER / SAFE_MULTIPLIER;
```

## performanceFee CAN LEAD TO DENIAL OF SERVICE (DOS)

The function `updatePerformanceFee()` allows the restricted user to change and set a new performanceFee. If performanceFee is set to a value higher than SAFE\_MULTIPLIER (1e18) it will produce a denial of service due to an underflow in the `lendingAPR()` function.

The lendingAPR() function uses  $((1 * \text{SAFE\_MULTIPLIER}) - \text{performanceFee})$  as one of the factors for calculating the lending APR. If performanceFee is set to a value higher than 1e18, for example 1e19, then the mentioned factor would be  $1e18 - 1e19$  which would revert.

In the described scenario every call to lendingAPR() would revert.

Proof of concept:

```
function testPocPerformanceFeeCausePOC() public {
    vm.startPrank(address(alice));
    vault.approveBorrower(address(bob)); // Approve Bob as borrower

    // Set interest rate
    ILendingVault.InterestRate memory _interestRate;
    _interestRate.baseRate = 0;
    _interestRate.multiplier = 254319144689994600000;
    _interestRate.jumpMultiplier = 45777446044199026;
    _interestRate.kink1 = 100000000000000;
    _interestRate.kink2 = 50000000000000000;
    vault.updateMaxInterestRate(bob, _interestRate);
    vault.updateInterestRate(bob, _interestRate);

    // Update performance fee to a value higher than SAFE_MULTIPLIER
    uint256 SAFE_MULTIPLIER = 1e18;
    vault.updatePerformanceFee(SAFE_MULTIPLIER + 1);
    vm.stopPrank();

    // User deposits and borrows to increase borrowAPR and utilizationRate
    deal(bob, 10 ether);
    vm.startPrank(bob);
    vault.depositNative{value: 1 ether}(1 ether, 0);
    vault.borrow(1 ether);

    // Call to lendingAPR() reverts
    vm.expectRevert();
    vault.lendingAPR();
}
```

### Recommendation:

Implement a check that reverts if the new value for performanceFee is higher than SAFE\_MULTIPLIER. The check should be implemented not only in updatePerformanceFee() function but also in the constructor where the performanceFee is first set.

**CHECK-EFFECTS-INTERACTIONS PATTERN NOT FOLLOWED**

The checks effects interactions pattern has not been followed in the `depositNative()` function. This function allows external call to the native asset contract on line: 267, after which state changes are done.

```
IWNT(address(asset)).deposit{ value: msg.value }();
```

This can be risky and make the contract vulnerable to cross-contract reentrancy as some of the native assets can be upgradeable contracts and can contain malicious code or code with bugs even though non-reentrant modifiers are used.. Instead it is advised to call the external interaction after all the state changes are done.

**Recommendation:**

It is advised to use checks effects interactions pattern as a best practice for security and call the external function/interaction with the external contract at the end after all the state changes.

Comment: The team said that they can't push `deposit()` function down to follow CEI as it will result in `totalAssets()` function being incorrect resulting in `mintShares()` error.



## INTEREST RATE CALCULATION DOES NOT COVER COMMENTED CASE

The internal function `_calculateInterestRate()` is used for calculating the interest rate based on the borrower's model and overall utilization rate. The function contemplates 3 different cases depending on the value of `_utilization` compared with `_interestRate.kink2` and `_interestRate.kink1` from the user's `interestRate`.

The first case should come when `_utilization` is above `_interestRate.kink2`, the second case should come when `_utilization` is between `_interestRate.kink2` and `_interestRate.kink1` and the third case should come when `_interestRate` is below `_interestRate.kink1` as it has been stated in the code comments.

The issue is related with the `if` statement for the second and third described cases:

```
// If _utilization between kink1 and kink2, rates are flat
if (_interestRate.kink1 < _utilization && _utilization <= _interestRate.kink2) {
return _interestRate.baseRate + (_interestRate.kink1 * _interestRate.multiplier /
SAFE_MULTIPLIER);
}

// If _utilization below kink1, calculate borrow rate for slope up to kink 1
return _interestRate.baseRate + (_utilization * _interestRate.multiplier
/SAFE_MULTIPLIER);
```

The implementation for second case is including `kink2` but not `kink1` which is being included in the third case. Following the comment 'If `_utilization` below `kink1`' it can be derived that `kink1` should be part of the second case not the third one.

### Recommendation:

Use `<=` in the `if` statement instead of `<`.

The changes would result on this:

```
if (_interestRate.kink1 <= _utilization && _utilization <= _interestRate.kink2)
```

## RESTRICTED MODIFIER IS RECOMMENDED TO BE USED ONLY IN EXTERNAL FUNCTIONS

LendingVault.sol inherits from AccessManaged.sol, which is a library from OpenZeppelin (<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/manager/AccessManaged.sol>). One of the **'IMPORTANT'** notes described in the contract is that the ``restricted`` modifier should never be used on internal functions and judiciously used in ``public`` ones, it should be ideally used on ``external`` functions.

LendingVault.sol implements the ``restricted`` modifier in 2 ``public`` functions, ``updateInterestRate()`` and ``updateMaxInterestRate()``. These 2 functions are not called internally so they can be redefined as ``external``.

### Recommendation:

Redefine ``updateInterestRate()`` and ``updateMaxInterestRate()`` as ``external`` functions to comply with the library recommendations.

## ENUMERABLESET'S FUNCTIONS ``add()`` and ``remove()`` RETURNED VALUES ARE NOT CHECKED

The functions ``.add()`` and ``.remove()`` from Openzeppelin's EnumerableSet library (<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableSet.sol>) return ``true`` if the addition or removal has been correctly executed, meaning that the item to remove was present in the set. It is a good practice to check the return value and revert if false.

### Recommendation:

Check the return value from ``.add()`` and ``.remove()`` in ``approveBorrower()`` and ``revokeBorrower()`` functions and revert if false.

### Missing Sanity checks for constructor parameters

In contract LendingVault.sol, there are missing sanity checks and validations for `_performanceFee` and `_maxCapacity` parameters in the constructor.

#### Recommendation:

It is advised to add appropriate sanity checks for the same.

### Redundant checks

Redundant check on **line: 265**:

```
if (assetAmt == 0) revert Errors.InsufficientDepositAmount();
```

This check is redundant because **lines: 262** and **263** already cover the scenario for the check on **line: 265**. And thus can be removed.

#### Recommendation:

It is advised to remove the redundant check.

### depositNative() does not check for isNativeAsset

The **depositNative()** can be called even if the **isNativeAsset** is NOT set to true. The function is currently callable even if **isNativeAsset** is set to false, although it reverts if the asset it is not a native asset.

#### Recommendation:

It is still advised to explicitly add a require check to ensure that **isNativeAsset** is set to true when **depositNative()** is called for better error handling and debugging.

### Kink1 can be greater than kink2

In the function **updateInterestRate()** there is no check to see whether **kink1** is lesser than **kink2** or not. This can result in miscalculation or errors when **\_calculateInterestRate()** is being called and calculated.

#### Recommendation:

It is advised to add a specific check to see whether **kink1** is lesser than **kink2** or not when **updateInterestRate()** is being called.

### UNNECESSARY CHECKS

There are some ERC20 related checks that are no needed because they are internally checked.

Code parts:

- withdraw() function:

```
if (sharesAmt > balanceOf(msg.sender)) revert Errors.InsufficientWithdrawBalance();
```

- deposit() function:

```
if (assetAmt == 0) revert Errors.InsufficientDepositAmount();
```

#### Recommendation:

Remove the mentioned lines as they are internally checked.

**utilizationRate IS NOT IN TERMS OF ASSET'S DECIMALS.**

The function `utilizationRate()` is assuming that the asset's decimals are 18. There is a restriction that does not allow asset's decimals to be higher than 18 but it does allow them to be equal or lower than 18, for example, 6.

Following the implementation of the `utilizationRate()` function, if `totalAsset_` is not 0 then the `utilizationRate` is returned in terms of  $1e18$  instead of asset's decimals.

Example: let's say that asset has 6 decimals,

- `totalBorrows`:  $100 * 1e6$
- `SAFE_MULTIPLIER`:  $1e18$
- `totalAsset`:  $200 * 1e6$

`utilizationRate`:  $(100 * 1e6 * 1e18) / 200 * 1e6 = 100 / 200 * 1e18$

```
function utilizationRate() public view returns (uint256){
    uint256 totalAsset_ = totalAsset();

    return (totalAsset_ == 0) ? 0 : totalBorrows * SAFE_MULTIPLIER / totalAsset_;
}
```

**Recommendation:**

Use `asset.decimals()` instead of `SAFE_MULTIPLIER` for the multiplication to obtain the `utilizationRate` in terms of asset's decimals.

The above described scenario now would be:

- `utilizationRate`: `totalBorrows * asset.decimals() / totalAssets_`

`utilizationRate`:  $(100 * 1e6 * 1e6) / 200 * 1e6 = 100 / 200 * 1e6$

## USE UNCHECKED ARITHMETIC

The 'unchecked' block in Solidity is employed to bypass automatic overflow and underflow checks on arithmetic operations. When used within this block, arithmetic calculations proceed without triggering these automatic checks, leading to gas savings.

LendingVault.sol implements 2 for loops that can save gas if unchecked arithmetic is implemented in `_pendingInterests()` and `borrowAPR()`.

### Recommendation:

Add ``unchecked{}`` for `i++` increment in for loop.

## CONSIDER USING IMMUTABLE VARIABLES

An immutable variable, assigned once during contract deployment and thereafter read-only, can result in substantial gas savings. This stands in contrast to constant variables, which incur gas consumption each time they are accessed due to their evaluation process.

LendingVault.sol contains some variables that can not be changed after deployment which could save gas if marked as ``immutable``.

### Recommendation:

Consider marking variables which are not changing after deployment as ``immutable`` to save gas.

## CATCH ARRAY LENGTH

Storing the array length in a variable before the loop starts reduces the number of read operations, resulting in a saving of approximately 3 gas units per iteration. This practice can yield substantial gas savings, particularly in situations involving loops that iterate over large arrays.

LendingVault.sol implements 2 for loops that can save gas if `array.length` is caught before the ``for loop``.

### Recommendation:

Catch `array.length` before the ``for loop`` to save gas.

## USE `++i` INSTEAD OF `i++`

The post-increment operation, `i++`, requires more gas compared to pre-increment (`++i`). Post-increment involves both incrementing the variable `i` and returning its initial value, which necessitates the use of a temporary variable. This additional step results in extra gas consumption, approximately 5 gas units per iteration.

LendingVault.sol implements 2 for loops that can save gas if `i` is pre incremented.

### Recommendation:

Use `++i` instead of `i++` in for loops.

## CALLING EXTERNAL FUNCTIONS CONSUME LESS GAS THAN PUBLIC

Invoking an external function generally incurs lower gas costs compared to a public function when accessed externally. This efficiency arises from the fact that external functions can bypass one step of copying argument data, moving it directly from calldata to memory.

### **Recommendation:**

Consider defining public functions as external when they are not called inside the contract by other functions.



## LendingVault.sol

Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Steadefi team.

**The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.**

Zokyo Security recommends the Steadefi team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

