Rivus Security Review

Pashov Audit Group

Conducted by: defsec, ubermensch, peanuts

October 28th - October 31st

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **SocksNFlops/rivus-staking** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Rivus

Rivus Staking lets users deposit, stake, and unstake tokens while providing flexibility to update fees, caps, and other parameters. It includes mechanisms to calculate staking and unstaking fees, enforce caps, manage approvals, and control specific actions like pausing, upgrading, and rebasing.

5. Risk Classification

Severity	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 any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or 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. Security Assessment Summary

review commit hash - f5472fba6e56b14ec333f93cd8fe4a7bacb1293b

fixes review commit hash - 03b457bc6c4b22a02ef4d1a002317bb2f41b8ab4

Scope

The following smart contracts were in scope of the audit:

- `StakingToken`
- `RsTAO`
- `RsNMT`
- `RebasingERC20Upgradeable`

7. Executive Summary

Over the course of the security review, defsec, ubermensch, peanuts engaged with Rivus to review Rivus. In this period of time a total of **10** issues were uncovered.

Protocol Summary

Protocol Name	Rivus
Repository	https://github.com/SocksNFlops/rivus-staking
Date	October 28th - October 31st
Protocol Type	Liquid staking

Findings Count

Severity	**Amount**
High	1
Medium	3
Low	6
Total Findings	**10**

Summary of Findings

ID	Title	Severity	Status
[H-01]	principle not updated in rebase() leading to underflow and withdrawal failures	High	Resolved
[M-01]	Incorrect principle tracking due to fee inclusion	Medium	Resolved
[M-02]	Decimals discrepancy causes the incorrect pegging	Medium	Resolved
[M-03]	Users can frontrun rebase to extract profits at the expense of long-term stakers	Medium	Acknowledged
[L-01]	Missing unstake request cancellation functionality can lead to locked funds	Low	Resolved
[L-02]	Unrestricted rebase authority can lead to token value manipulation	Low	Acknowledged
[L-03]	renounceRole should be overridden and reverted	Low	Resolved
[L-04]	GasFee should not be zero	Low	Resolved
[L-05]	Excess msg.value is not returned to the user	Low	Resolved
[L-06]	convertToShares() rounds up, potentially leading to insolvency issues	Low	Resolved

8. Findings

8.1. High Findings

[H-01] `principle` not updated in `rebase()` leading to underflow and withdrawal failures

Severity

Impact: High

Likelihood: Medium

Description

The `principle` variable in the `StakingToken` contract is intended to track the total amount of underlying tokens deposited into the contract. However, within the `rebase` function, which adjusts the total supply of tokens (e.g., to reflect accrued interest or rewards), the `principle` variable is not updated to match the new total supply.

This mismatch leads to a critical issue: when the total supply increases due to a rebase, but `principle` remains the same, subsequent calculations that rely on `principle` can underflow. Specifically, during the unstaking process in the `_approveUnstakeRequest` function, the contract attempts to subtract the unstaked amount from `principle`. If the unstaked amount exceeds the unchanged `principle`, this subtraction underflows, causing transactions to fail.

As a result, users will be unable to withdraw their funds after a rebase that increases the total supply, leading to a DoS, especially for the last withdrawing users.

POC:

```
function test_rebase_doesnt_update_principle() public {
    uint256 amount = 1235038820;
    uint256 minStakingAmount = 1e6;
    uint16 stakingFeeBPS = 0;
    uint256 bridgingFee = 0;

    address rebaseAccount = makeAddr("Rebaser");
    address withdrawingAccount = makeAddr("Withdrawer");

    // Have the admin grant the WITHDRAWAL_ROLE  and REBASE_ROLE to the accounts
    vm.startPrank(admin);
    stakingToken.grantRole(stakingToken.REBASE_ROLE(), rebaseAccount);
    stakingToken.grantRole(stakingToken.WITHDRAWAL_ROLE(), withdrawingAccount);
    vm.stopPrank();

    // Have the config manager set configs
    vm.startPrank(configManager);
    stakingToken.setMinStakingAmount(minStakingAmount);
    stakingToken.setStakingFeeBPS(stakingFeeBPS);
    stakingToken.setBridgingFee(bridgingFee);
    stakingToken.setProtocolVault(protocolVault);
    stakingToken.setMaxDepositPerRequest(type(uint256).max);
    stakingToken.setCap(type(uint256).max);
    vm.stopPrank();

    // Mint the amount of undelying tokens to userA
    mockUnderlyingToken.mint(userA, amount);

    // Have userA approve the staking token to pull the underlying token from
    // them
    vm.prank(userA);
    mockUnderlyingToken.approve(address(stakingToken), amount);

    // Have userA deposit half the amount of underlying tokens
    vm.startPrank(userA);
    stakingToken.deposit(amount/2);
    vm.stopPrank();

    // Have the account call rebase with the new total supply +20%
    vm.startPrank(rebaseAccount);
    stakingToken.rebase(stakingToken.totalSupply() * 6/5);
    vm.stopPrank();

    // Have userA deposit the second half of underlying tokens
    vm.startPrank(userA);
    stakingToken.deposit(amount/2);
    vm.stopPrank();

    // Get the staking token balance of userA
    uint256 unstakingAmount = stakingToken.balanceOf(userA);

    // Mint the unstakingAmount of underlyingTokens to the withdrawingAccount
    // and approve the staking token to pull them
    vm.startPrank(withdrawingAccount);
    mockUnderlyingToken.mint(withdrawingAccount, unstakingAmount);
    mockUnderlyingToken.approve(address(stakingToken), unstakingAmount);
    vm.stopPrank();

    vm.startPrank(userA);
    stakingToken.requestUnstake(unstakingAmount);
    vm.stopPrank();



    console.log("Asset Amount = %s", stakingToken.convertToAssets
      (request.shares));
    console.log("principle = %s", stakingToken.principle());

    // Have the withdrawingAccount approve the unstake request
    //(Which will revert due to underflow)
    vm.startPrank(withdrawingAccount);
    vm.expectRevert(stdError.arithmeticError);
    stakingToken.approveUnstakeRequest();
    vm.stopPrank();
}
```

Recommendations

Update the `principle` variable within the `rebase` function to accurately reflect changes in the total supply. By synchronizing `principle` with the adjusted total supply, you ensure that all calculations dependent on `principle` remain accurate, preventing underflow errors and allowing users to withdraw their funds without issues.

Specifically, consider adding logic to the `rebase` function similar to:

```
function rebase(uint256 newTotalSupply) external onlyRole(REBASE_ROLE) {
    // Existing rebase logic...
    _updateApy(newTotalSupply, currentTotalSupply);
    lastRebaseTimestamp = block.timestamp;
    lastTotalSupply = currentTotalSupply;
    _rebase(newTotalSupply);

+   // Update principle to match the new total supply
+   principle = newTotalSupply;
}
```

This adjustment ensures that `principle` remains consistent with the total supply after a rebase, maintaining the integrity of the contract's financial calculations.

8.2. Medium Findings

[M-01] Incorrect principle tracking due to fee inclusion

Severity

Impact: Medium

Likelihood: Medium

Description

In the deposit function, the `principle` state variable is incorrectly updated by adding the full deposit amount, including fees that are not actually staked:

```
function deposit(uint256 amount) external whenNotPaused returns
  (uint256 mintedAmount) {
    // Calculate fees
    (mintedAmount, stakingFee) = _calculateMintAmountAndFees(amount);

    // Incorrect: Updates principle with full amount including fees
    principle += amount;  // @audit - includes stakingFee and bridgingFee

    // Transfer and mint logic...
    IERC20(underlyingToken).safeTransfer(protocolVault, stakingFee);
    bool success = _bridge(mintedAmount + bridgingFee);
}
```

The principle should only track the actual staked amount (`mintedAmount`) rather than the total deposit amount which includes:

- Staking fee (`stakingFee`)
- Bridging fee (`bridgingFee`)
- Actually staked amount (`mintedAmount`)

Impact

- Protocol reports inflated total principle amount
- TVL calculations will be incorrect
- APY calculations may be skewed due to incorrect base values

Recommendations

Update the principle to only include the actual staked amount.

[M-02] Decimals discrepancy causes the incorrect pegging

Severity

Impact: Medium

Likelihood: Medium

Description

The `StakingToken` contract is intended to create a staking token pegged to the underlying token `wTAO`. According to the NatSpec comments in the `IStakingToken` interface:

```
/**
 *
    @notice The address of the underlying token that this staking token is pegged to
 */
function underlyingToken() external view returns (address);
```

However, there's a significant discrepancy due to differing decimal places between the two tokens. The underlying token `wTAO` has **9 decimals**, whereas the staking token has **18 decimals**.

This difference leads to a misalignment in value representation:

- **Intended Pegging**: 1 `wTAO` (which is `1e9` units considering 9 decimals) should equal 1 staking token (`1e18` units with 18 decimals).
- **Actual Behavior**: When a user deposits 1 `wTAO` (`1e9` units), they receive `1e9` units of the staking token, which is only `0.000000001` of the staking token when considering its 18 decimals.

As a result, the staking token is not correctly pegged to the underlying `wTAO` token.

POC:

```
function test_deposit() public {

    uint256 amount = 1 * 10**mockUnderlyingToken.decimals();
    uint256 minStakingAmount = 1e6;
    uint16 stakingFeeBPS = 0;
    uint256 bridgingFee = 0;

    // Have the config manager set configs
    vm.startPrank(configManager);
    stakingToken.setMinStakingAmount(minStakingAmount);
    stakingToken.setStakingFeeBPS(stakingFeeBPS);
    stakingToken.setBridgingFee(bridgingFee);
    stakingToken.setProtocolVault(protocolVault);
    stakingToken.setMaxDepositPerRequest(type(uint256).max);
    stakingToken.setCap(type(uint256).max);
    vm.stopPrank();

    // Mint the amount of undelying tokens to userA
    mockUnderlyingToken.mint(userA, amount);

    // Have userA approve the staking token to pull the underlying token from
    // them
    vm.prank(userA);
    mockUnderlyingToken.approve(address(stakingToken), amount);

    // Have userA mint the amount of staking tokens
    vm.startPrank(userA);
    stakingToken.deposit(amount);
    vm.stopPrank();

    // Get the staking token balance of userA
    uint256 stakingTokenBalance = stakingToken.balanceOf(userA);

    console.log("balance = %s", stakingTokenBalance);
}
```

POC Output

```
Ran 1 test for test/Attack.t.sol:AttackTest
[PASS] test_deposit() (gas: 365732)
Logs:
  balance = 1000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.21ms
  (172.65µs CPU time)

Ran 1 test suite in 3.65ms
  (1.21ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendations

To fix the pegging issue, adjust the staking token's decimals or modify the conversion logic to account for the decimal difference:

1. **Match Decimals**: Set the staking token's decimals to match that of the underlying `wTAO` token (9 decimals). This ensures a 1:1 pegging between the two tokens.

```
function decimals() public view virtual override returns (uint8) {
    return 9; // Match the underlying wTAO token's decimals
}
```

2. **Adjust Conversion Logic**: If changing the staking token's decimals is not feasible, modify the deposit and withdrawal functions to scale the amounts appropriately.

[M-03] Users can frontrun rebase to extract profits at the expense of long-term stakers

Severity

Impact: Medium

Likelihood: Medium

Description

The `StakingToken` contract is susceptible to a frontrunning attack during the rebase process. Users who monitor on-chain events can detect when a rebase is imminent, especially if they can predict or observe an increase in the total supply in the calldata. They can exploit this by:

1. **Front-running the Rebase**: Depositing a significant amount of tokens just before the rebase occurs, thereby increasing their share of the total supply.

2. **Benefiting from the Rebase**: When the rebase increases the total supply, these users receive a disproportionate share of the newly minted tokens relative to their short-term stake.

3. **Back-running with Withdrawal**: Immediately requesting an unstake after the rebase, capturing the rebase rewards without a long-term commitment.

This behavior allows opportunistic users to extract undue profits, effectively diluting the value of long-term stakers' holdings. It undermines the fairness and intended reward distribution of the staking mechanism.

Recommendations

Enforce a minimum staking duration before users are allowed to unstake or withdraw their tokens. This prevents immediate withdrawal after benefiting from a rebase.

8.3. Low Findings

[L-01] Missing unstake request cancellation functionality can lead to locked funds

The `StakingToken` contract lacks functionality for users to cancel their pending unstake requests. Once a request is submitted via `requestUnstake()`, users cannot cancel it and must wait for the request to be processed. Users' funds remain locked in the contract until their request is processed, even if they want to cancel.

```
function requestUnstake(uint256 amount) external payable whenNotPaused {
    // Validate that the msg.value is equal to the gasFee
    if (msg.value < gasFee) {
        revert InsufficientGasFee(msg.value, gasFee);
    }

    // Validate that the amount is greater than 0
    if (amount == 0) {
        revert InvalidUnstakeAmount(amount);
    }

    // Build the unstake request
    UnstakeRequest memory request;
    request.shares = convertToShares(amount);
    request.recipient = msg.sender;
    request.requestTime = block.timestamp;

    // Add the unstake requests to the queue
    unstakeRequests[unstakeRequestsIndex + unstakeRequestsCount] = request;
    unstakeRequestsCount++;

    // Transfer the tokens to the staking contract to be locked
    _transfer(msg.sender, address(this), amount);

    // Emit the unstake request event
    emit UnstakeRequested(msg.sender, amount, request.shares);
}
```

Recommendation

Consider adding a cancellation function.

[L-02] Unrestricted rebase authority can lead to token value manipulation

The `rebase()` function allows REBASE_ROLE holders to arbitrarily modify the total token supply without any bounds or rate-limiting controls. A malicious or compromised role holder could drastically alter token values and drain user funds through excessive rebasing.

Current implementation:

```
function rebase(uint256 newTotalSupply) external onlyRole(REBASE_ROLE) {
    if (block.timestamp < lastRebaseTimestamp + rebaseDelay) {
        revert RebaseDelayNotElapsed
          (lastRebaseTimestamp, rebaseDelay, block.timestamp);
    }
    uint256 currentTotalSupply = totalSupply();
    _updateApy(newTotalSupply, currentTotalSupply);
    lastRebaseTimestamp = block.timestamp;
    lastTotalSupply = currentTotalSupply;
    _rebase(newTotalSupply);
}
```

Key vulnerabilities:

1. No maximum change limit per rebase operation
2. No validation of rebase direction (positive/negative)
3. No correlation with actual yield or protocol performance

Implement strict bounds and validation on rebase operations:

Add Rebase Limitations:

```
contract StakingToken {
    uint256 public constant MAX_REBASE_PERCENT = 1000; // 10% in BPS
    uint256 public constant MIN_REBASE_INTERVAL = 1 days;

    // Track cumulative rebase changes
    uint256 public dailyRebaseAmount;
    uint256 public lastRebaseReset;

    function rebase(uint256 newTotalSupply)
        external
        onlyRole(REBASE_ROLE)
    {
        // Validate timing
        require(
            block.timestamp >= lastRebaseTimestamp + rebaseDelay,
            "Rebase delay not met"
        );

        // Reset daily tracking if needed
        if (block.timestamp >= lastRebaseReset + 1 days) {
            dailyRebaseAmount = 0;
            lastRebaseReset = block.timestamp;
        }

        uint256 currentSupply = totalSupply();

        // Calculate absolute change
        uint256 changeAmount;
        if (newTotalSupply > currentSupply) {
            changeAmount = newTotalSupply - currentSupply;
            require(
                changeAmount * 10000 <= currentSupply * MAX_REBASE_PERCENT,
                "Rebase increase too large"
            );
        } else {
            changeAmount = currentSupply - newTotalSupply;
            require(
                changeAmount * 10000 <= currentSupply * MAX_REBASE_PERCENT,
                "Rebase decrease too large"
            );
        }

        // Track daily changes
        dailyRebaseAmount += changeAmount;
        require(
            dailyRebaseAmount * 10000 <= currentSupply * MAX_REBASE_PERCENT,
            "Daily rebase limit exceeded"
        );

        // Perform rebase
        _updateApy(newTotalSupply, currentSupply);
        _rebase(newTotalSupply);

        emit RebaseExecuted(
            currentSupply,
            newTotalSupply,
            changeAmount,
            block.timestamp
        );
    }
}
```

[L-03] `renounceRole` should be overridden and reverted

The protocol uses AccessControlUpgradeable, and sets `DEFAULT_ADMIN_ROLE`. In AccessControlUpgradeable, there is a function to renounce a role, which will revoke the role of the caller.

```
function renounceRole
    (bytes32 role, address callerConfirmation) public virtual {
        if (callerConfirmation != _msgSender()) {
            revert AccessControlBadConfirmation();
        }

        _revokeRole(role, callerConfirmation);
    }
```

Usually, this function is overridden and reverted to prevent any accidental renounce, especially the `DEFAULT_ADMIN_ROLE`.

The best practice is to override the function and revert.

[L-04] GasFee should not be zero

Users call `StakingToken.requestUnstake()` to convert their rebasing token back to the underlying token. Once `requestUnstake()` is called, the `WITHDRAWAL_ROLE` will have to approve the unstaking request. The unstaking request is also in sequential order, so a previous unstake has to be approved before the next unstake can be approved.

`requestUnstake()` has a `gasFee`, but this gasFee can be set to zero.

```
function setGasFee(uint256 gasFee_) external onlyRole(CONFIG_ROLE) {
        if (gasFee_ > 0.01 ether) {
            revert InvalidGasFee(gasFee_);
        }
        gasFee = gasFee_;
        emit GasFeeUpdated(gasFee_);
    }
```

If the `gasFee` is zero, users can spam the `requestUnstake` to temporarily DoS the unstake order.

Consider setting a minimum sum for `gasFee`, `require(gasFee_ > 0.0001 ether)` to deter anyone from spamming the unstaking sequence.

[L-05] Excess msg.value is not returned to the user

In `StakingToken.requestUnstake()`, the caller has to send a certain amount of `gasFee` in order for the function to continue execution.

```
function requestUnstake(uint256 amount) external payable whenNotPaused {
        // Validate that the msg.value is equal to the gasFee
        if (msg.value < gasFee) {
            revert InsufficientGasFee(msg.value, gasFee);
        }
```

If the caller sends the excess amount of `msg.value`, it will not be returned to the user.

Consider setting a strict equality `msg.value == gasFee` so that the user will not overpay for the `gasFee`.

[L-06] `convertToShares()` rounds up, potentially leading to insolvency issues

In the `RebasingERC20Upgradeable` contract, the `convertToShares` function rounds up when converting assets to shares. Specifically, it uses `Math.mulDiv` with `Math.Rounding.Ceil`:

```
function convertToShares(uint256 assets) public view virtual returns (uint256) {
    RebasingERC20Storage storage $ = _getRebasingERC20Storage();
    if ($._totalShares == 0 && $._totalSupply == 0) {
        return assets * (10 ** _decimalsOffset());
    } else {
        return Math.mulDiv
          (assets, $._totalShares, $._totalSupply, Math.Rounding.Ceil);
    }
}
```

This rounding method favors users when they request an unstake because it potentially grants them more shares than the exact proportional amount. Over time, especially with a high volume of transactions, these small discrepancies can accumulate, leading to the protocol distributing more assets than it actually holds. This can cause insolvency issues where the total shares redeemed by users exceed the actual assets available in the contract.

Modify the `convertToShares` function to round down instead of rounding up when calculating shares from assets. This change ensures that users receive a slightly smaller number of shares, preventing the protocol from over-allocating assets.

You can implement this by changing the rounding direction in the `Math.mulDiv` function to `Math.Rounding.Down` or by omitting the rounding parameter (since the default is to round down):

```
function convertToShares(uint256 assets) public view virtual returns (uint256) {
    RebasingERC20Storage storage $ = _getRebasingERC20Storage();
    if ($._totalShares == 0 && $._totalSupply == 0) {
        return assets * (10 ** _decimalsOffset());
    } else {
+       return Math.mulDiv(assets, $._totalShares, $._totalSupply);
+       // or explicitly specify rounding down
+       // return Math.mulDiv
+ (assets, $._totalShares, $._totalSupply, Math.Rounding.Down);
    }
}
```

By rounding down, the protocol protects itself from cumulative rounding errors that could lead to insolvency. This approach aligns with best practices, ensuring the system remains solvent and fair for all participants.