# Tanssi Security Review

## Pashov Audit Group

Conducted by: unforgiven, SpicyMeatball, zark, Oblivionis, Dimah, Shubham, ph

April 30th 2025 - May 13th 2025

# 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 **moondance-labs/tanssi-symbiotic** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

# 4. About Tanssi

Tanssi provides an infrastructure that lets to spawn substrate/EVM based blockchain on demand while guaranteeing security through stakes present in Symbiotic.

# 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 hashes*

- 5e9cf50106f4624cf4e975097e9360ce61a12e20
- f39bcaa2957642174e7201ed00df27e8e57f4b6d

*fixes review commit hashes*

- 8ec5ade1155212bcb3e714ca3b8c9298d193c73e

## Scope

The following smart contracts were in scope of the audit:

- `OSharedVaults`
- `QuickSort`
- `Middleware`
- `MiddlewareProxy`
- `MiddlewareStorage`
- `OBaseMiddlewareReader`
- `ODefaultOperatorRewards`
- `ODefaultStakerRewards`
- `ODefaultStakerRewardsFactory`
- `BeefyClient`
- `Gateway`
- `Operators`
- `Types`
- `OSubstrateTypes`
- `GatewayCoreStorage`

# 7. Executive Summary

Over the course of the security review, unforgiven, SpicyMeatball, zark, Oblivionis, Dimah, Shubham, ph engaged with Moondance Labs to review Tanssi. In this period of time a total of **24** issues were uncovered.

## Protocol Summary

| | |
|---|---|
| **Protocol Name** | Tanssi |
| **Repository** | https://github.com/moondance-labs/tanssi-symbiotic |
| **Date** | April 30th 2025 - May 13th 2025 |
| **Protocol Type** | Interaction toolkit and Bridge |

## Findings Count

| **Severity** | **Amount** |
|---|---|
| High | 1 |
| Medium | 6 |
| Low | 17 |
| **Total Findings** | **24** |

# Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-01] | Permissionless sendCurrentOperatorsKeys() | High | Resolved |
| [M-01] | Same reward can be claimed multiple times in certain cases | Medium | Resolved |
| [M-02] | Disabling collateral with setCollateralToOracle() blocks reward claims | Medium | Acknowledged |
| [M-03] | Operator key change might cause loss of access to unclaimed rewards | Medium | Acknowledged |
| [M-04] | Vault curator can force operator.claimReward() to revert with precision | Medium | Resolved |
| [M-05] | getOperatorPowerAt uses current prices for past stake calculations | Medium | Acknowledged |
| [M-06] | _getRewardsAmountPerVault might be using an outdated vault power | Medium | Acknowledged |
| [L-01] | getEpochAtTs() incorrect when timestamp matches epoch start | Low | Resolved |
| [L-02] | operatorVaultPairs array length not adjusted | Low | Resolved |
| [L-03] | ClaimAdminFee() omits epoch in ClaimAdminFee event | Low | Resolved |
| [L-04] | Middleware::slash can revert due to VetoSlasher near epoch end | Low | Acknowledged |
| [L-05] | Constructor of ODefaultOperatorRewards missing disableInitializers() call | Low | Resolved |

| | | | |
|---|---|---|---|
| [L-06] | _beforeRegisterOperator allows invalid validator registration | Low | Acknowledged |
| [L-07] | adminFeeAmount always round down, resulting in reduced revenue for the admin | Low | Acknowledged |
| [L-08] | distributeRewards() revert | Low | Resolved |
| [L-09] | Missing executeSlash() after requestSlash() for veto slashing | Low | Resolved |
| [L-10] | Middleware may send bytes32(0) as validator key | Low | Resolved |
| [L-11] | Vault rewards distribution fails without ODefaultStakerRewards | Low | Acknowledged |
| [L-12] | Missing validation for stale or invalid price data in Middleware::stakeToPower | Low | Acknowledged |
| [L-13] | Middleware treats validators with very low active power as valid | Low | Acknowledged |
| [L-14] | Pending slashes are not taken into account | Low | Acknowledged |
| [L-15] | Vaults can maliciously escaping slashing | Low | Acknowledged |
| [L-16] | Recursive QuickSort will always fail for some worst-case scenarios | Low | Acknowledged |
| [L-17] | performUpkeep() may exceed Chainlink's maxPerformDataSize | Low | Acknowledged |

# 8. Findings

## 8.1. High Findings

## [H-01] Permissionless `sendCurrentOperatorsKeys()`

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

### The First impact

The `Middleware.sendCurrentOperatorsKeys()` function is permissionless, meaning any external actor can call it. This function triggers the `Gateway.sendOperatorsData()` function on the Ethereum side of our bridge, which:

- Emits an OutboundMessageAccepted event.
- Increments the outboundNonce of the channel.
- Submits a message to the outbound channel of the bridge infrastructure.

```
// Middleware.sol
    function sendCurrentOperatorsKeys() external returns
      (bytes32[] memory sortedKeys) {
        address gateway = getGateway();
        if (gateway == address(0)) {
            revert Middleware__GatewayNotSet();
        }

        uint48 epoch = getCurrentEpoch();
        sortedKeys = IOBaseMiddlewareReader(address(this)).sortOperatorsByPower
          (epoch);
        IOGateway(gateway).sendOperatorsData(sortedKeys, epoch);
    }
```

Now, let's see how `Gateway::sendOperatorsData()` works:

```
function sendOperatorsData
      (bytes32[] calldata data, uint48 epoch) external onlyMiddleware {
        Ticket memory ticket = Operators.encodeOperatorsData(data, epoch);
        _submitOutboundToChannel(PRIMARY_GOVERNANCE_CHANNEL_ID, ticket.payload);
    }
```

The creation of the ticket for this outbound is done with 0 cost, so it will be fee-less and free to send this outbound message. As we can see in the `Operators` lib:

```
// TODO: This is a type from Snowbridge, do we want our own simplified
        // Ticket type?
        ticket.dest = ParaID.wrap(0);
        // TODO For now mock it to 0
@>        ticket.costs = Costs(0, 0);

        ticket.payload = OSubstrateTypes.EncodedOperatorsData
          (operatorsKeys, uint32(validatorsKeysLength), epoch);
```

So, as we can see, anyone will be able to permissionlessly and free of cost, spam the infrastructure nodes of the bridge and DoS them. We can expect this will cause serious problems in the bridge operations such as griefing, resource exhaustion, or race conditions.

# The Second impact

`Middleware.sendCurrentOperatorsKeys()` allows users to send the current validator set to the Tanssi consensus layer. If this function is called exactly at `epochStartTs`, it may return different results within the same block. For example, if a user `deposit`s stake, it might cause a new validator to join the queue; if a user `withdraw`s stake, it might cause a validator to leave the queue.

When a new validator set is processed through the `process_message` function of the `SymbioticMessageProcessor`, it calls `pallet_external_validators::Pallet::<T>::set_external_validators_inner` to set the new validator set.

Each time a new validator set is received, it overwrites the previously stored validator set information, and waits until the next era to be officially written into the validator set.

There is an edge case where the first `OperatorsKeys` submitted by a user within a block is included in an old era on the Tanssi chain, while the second

or subsequent OperatorsKeys are included in the next era. This can result in an incorrect epoch: the first era of the epoch uses an incorrect operator set (which does not match the one on the Ethereum side), while the remaining eras use the correct operator set. Since the slashing and reward logic on the Ethereum side relies entirely on the correct validator set from the latter eras, the first era will use incorrect values for slashing and rewards.

# Recommendations

Restrict access to `sendCurrentOperatorsKeys()` by requiring a specific role (like `FORWARDER_ROLE`), or introduce a rate-limiting mechanism based on epoch timing, or introduce a small fee.

# 8.2. Medium Findings

## [M-01] Same reward can be claimed multiple times in certain cases

### Severity

**Impact:** High

**Likelihood:** Low

### Description

The `ODefaultOperatorRewards` contract distributes rewards to operators based on their active stake. To prevent multiple claims, the contract tracks claimed amounts via:

```
claimed[eraIndex][recipient]
```

However, the recipient is derived from the `operatorKey` using:

```
address recipient = IOBaseMiddlewareReader(middlewareAddress).operatorByKey
  (abi.encode(input.operatorKey));
```

The issue arises because operator keys can be reassigned over time by the middleware. If a key that was previously associated with one operator is later reassigned to a different operator address, that new operator could claim the same reward again—since the recipient address is different, and the `claimed` mapping would not detect the duplicate claim.

### Recommendations

Track claimed rewards by `operatorKey` instead of recipient address: `claimed[eraIndex][key]`.

# [M-02] Disabling collateral with `setCollateralToOracle()` blocks reward claims

## Severity

**Impact:** High

**Likelihood:** Low

## Description

Calling `setCollateralToOracle(collateral, address(0))` on `Middleware` disables the collateral by removing its associated oracle. It introduces a critical issue: **vaults that used this collateral prevent their operators from claiming rewards**.

```
function claimRewards(
        ClaimRewardsInput calldata input
    ) external nonReentrant returns (uint256 amount) {
        --Snipped--
        _distributeRewardsToStakers(

                        eraRoot_.epoch, input.eraIndex, stakerAmount, recipient, midd
        );
    }

   function _distributeRewardsToStakers(
        uint48 epoch,
        uint48 eraIndex,
        uint256 stakerAmount,
        address operator,
        address middlewareAddress,
        address tokenAddress,
        bytes calldata data
    ) private {
        --Snipped--
        uint256[] memory amountPerVault = _getRewardsAmountPerVault(

                        operatorVaults, totalVaults, epochStartTs, operator, middlewa
        );

        _distributeRewardsPerVault(
          epoch,
          eraIndex,
          tokenAddress,
          totalVaults,
          operatorVaults,
          amountPerVault,
          data
        );
    }
```

The problem arises during `claimRewards()` execution. This function calls `_distributeRewardsToStakers()`, which relies on `stakeToPower()` to calculate vault power. `stakeToPower()` fetches the oracle address via `collateralToOracle()`. If the mapping returns `address(0)`, the function reverts, blocking the entire reward claim process.

## Recommendations

- **Disallow setting `collateralToOracle` to `address(0)`** unless a mechanism is in place to safely handle rewards for historical vaults.
- Introduce a **"retired" state for collaterals**, allowing historical rewards to be claimed while preventing new vaults from using the disabled collateral.
- Consider **caching oracle prices per epoch** to avoid depending on live price feeds for historical calculations.

Additional Notes

This issue reflects a systemic dependency on live oracle data even for historical accounting. Administrative or governance operations that affect `collateralToOracle` should ensure backward compatibility for reward calculations.

# [M-03] Operator key change might cause loss of access to unclaimed rewards

## Severity

**Impact:** High

**Likelihood:** Low

## Description

The `claimRewards()` function in the `ODefaultOperatorRewards` contract relies on `operatorByKey()` to resolve the operator's address from the provided `operatorKey`. However, this approach is fragile when operator keys are updated.

```
function claimRewards(
        ClaimRewardsInput calldata input
    ) external nonReentrant returns (uint256 amount) {
        OperatorRewardsStorage storage $ = _getOperatorRewardsStorage();
        EraRoot memory eraRoot_ = $.eraRoot[input.eraIndex];
        address tokenAddress = eraRoot_.tokenAddress;
        if (eraRoot_.root == bytes32(0)) {
            revert ODefaultOperatorRewards__RootNotSet();
        }
        --Snipped--
        // Starlight sends back only the operator key, thus we need to get back
        // the operator address
@>      address recipient = IOBaseMiddlewareReader
  (middlewareAddress).operatorByKey(abi.encode(input.operatorKey));
         --Snipped--
}
```

The `Middleware` admin can change an operator's key by calling `updateOperatorKey()`, `registerOperator()`, or `unregisterOperator()`. Internally, these functions call `KeyManager256._updateKey()`, which updates the mapping of keys to addresses. This function deletes the data for previous key and, in fact, releases it. Once a key is released, `operatorByKey()` may either:

- Return `address(0)`, or.
- Return a different operator address after being reassigned.

Consequently, an operator who has unclaimed rewards (associated with the deleted key) will no longer be able to claim them via `claimRewards()`, effectively losing access to those rewards.

# Recommendations

Make sure operator rewards are claimed before updating the operator key and deleting the key data.

# [M-04] Vault curator can force `operator.claimReward()` to revert with precision

## Severity

**Impact:** High

**Likelihood:** Low

# Description

A vault curator can allocate a **very small amount** of stake to an operator. Due to precision loss (e.g., rounding errors) in `_getRewardsAmountPerVault()`, the vault may end up with a **reward amount of zero**.

```
function _getRewardsAmountPerVault(
        address[] memory operatorVaults,
        uint256 totalVaults,
        uint48 epochStartTs,
        address operator,
        address middlewareAddress,
        uint256 stakerAmount
    ) private view returns (uint256[] memory amountPerVault) {
        --Snipped--
        for (uint256 i; i < totalVaults;) {
            uint256 amountForVault;
            // Last vault gets the remaining amount
            if (i == totalVaults - 1) {
                amountForVault = stakerAmount - distributedAmount;
            } else {
@>              amountForVault = vaultPowers[i].mulDiv
  (stakerAmount, totalPower);
            }
            amountPerVault[i] = amountForVault;
            unchecked {
                distributedAmount += amountForVault;
                ++i;
            }
        }
    }
```

When the operator later calls `claimReward()`, it triggers `vault.distributeRewards()` and then `_transferAndCheckAmount()` for each vault. If the calculated reward amount for a vault is zero, the `_transferAndCheckAmount()` function can **revert**, causing the **entire claim transaction to fail**.

As a result:

- The operator is **unable to claim their rewards**.
- Other vaults associated with the operator are also blocked from receiving their share of the rewards.

# Recommendations

Implement safeguards to:

- **Skip vaults** with zero reward amounts during distribution.
- **Set a minimum stake threshold** to avoid precision-based exploits..

# [M-05] `getOperatorPowerAt` uses current prices for past stake calculations

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

`OBaseMiddlewareReader::getOperatorPowerAt()` is trying to calculate the stake power that a specific operator had at a past timestamp based on their stake across all active vaults. This past stake power is used to sort the operator against others in `sortOperatorsByPower`, and also in `ODefaultOperatorRewards::_getRewardsAmountPerVault` to determine how the `stakerAmount` will be distributed among their active shared vaults.

```
function _getRewardsAmountPerVault(
        address[] memory operatorVaults,
        uint256 totalVaults,
        uint48 epochStartTs,
        address operator,
        address middlewareAddress,
        uint256 stakerAmount
    ) private view returns (uint256[] memory amountPerVault) {
        // ...

        uint256[] memory vaultPowers = new uint256[](totalVaults);
        uint256 totalPower;
        for (uint256 i; i < totalVaults;) {
@>            vaultPowers[i] = reader.getOperatorPowerAt
  (epochStartTs, operator, operatorVaults[i], subnetwork);
            totalPower += vaultPowers[i];
            unchecked {
                ++i;
            }
        }

        // ...
    }
```

However, `getOperatorPowerAt` fails to calculate the correct stake power of the operator at the past timestamp because it uses current token prices. Here's how

`getOperatorPowerAt` works:

```
// VaultManager.sol
    function _getOperatorPowerAt(
        uint48 timestamp,
        address operator,
        address vault,
        uint96 subnetwork
    ) internal view returns (uint256) {
        uint256 stake = _getOperatorStakeAt
          (timestamp, operator, vault, subnetwork);
        return stakeToPower(vault, stake);
    }

    function _getOperatorStakeAt(
        uint48 timestamp,
        address operator,
        address vault,
        uint96 subnetwork
    ) private view returns (uint256) {
        bytes32 subnetworkId = _NETWORK().subnetwork(subnetwork);
@>        return IBaseDelegator(IVault(vault).delegator()).stakeAt
  (subnetworkId, operator, timestamp, "");
    }
```

As we can see in `_getOperatorStakeAt`, it correctly retrieves the stake at the specified past timestamp. However, during the conversion of stake to power in `stakeToPower`, it uses the current price of the collateral token from Chainlink feeds:

```
function stakeToPower(
      addressvault,
      uint256stake
    ) public view override returns (uint256 power
        address collateral = vaultToCollateral(vault);
        address oracle = collateralToOracle(collateral);

        if (oracle == address(0)) {
            revert Middleware__NotSupportedCollateral(collateral);
        }
        (, int256 price,,,) = AggregatorV3Interface(oracle).latestRoundData();
        uint8 priceDecimals = AggregatorV3Interface(oracle).decimals();
        power = stake.mulDiv(uint256(price), 10 ** priceDecimals);
        // Normalize power to 18 decimals
        uint8 collateralDecimals = IERC20Metadata(collateral).decimals();
        if (collateralDecimals != DEFAULT_DECIMALS) {
            power = power.mulDiv
              (10 ** DEFAULT_DECIMALS, 10 ** collateralDecimals);
        }
    }
```

As a result, the computed power does not correctly reflect the operator's power at the given past timestamp. This leads to incorrect reward distributions in `ODefaultOperatorRewards`, as reward claims will be affected by recent price changes. This is inaccurate because the prices at the time of distribution

(meaning when `ODefaultOperatorRewards::distributeRewards` was called) would differ.

## Recommendations

Store and reference historical price data for each collateral token at the time of reward distribution or stake snapshots with usage of Open Zeppelin Checkpoints library.

# [M-06] `_getRewardsAmountPerVault` might be using an outdated vault power

## Severity

**Impact:** High

**Likelihood:** Low

## Description

In Symbiotic, if a vault is slashed in the current epoch, its `activeStake` will be immediately reduced:

```
//
// https://github.com/symbioticfi/core/blob/main/src/contracts/vault/Vault.sol#L237
            if (slashedAmount > 0) {
                uint256 activeSlashed = slashedAmount.mulDiv
                  (activeStake_, slashableStake);
                uint256 nextWithdrawalsSlashed = slashedAmount - activeSlashed;

                _activeStake.push(Time.timestamp
                  (), activeStake_ - activeSlashed);

                                    withdrawals[captureEpoch + 1] = nextWithdrawals - nex
            }
```

However, in the current Tanssi implementation, regardless of how many times `_distributeRewardsToStakers` is called in an epoch, the power of different vaults is still calculated based on the power weights at the beginning of the epoch:

```
//ODefaultOperatorRewards.sol::L242
        uint256[] memory amountPerVault = _getRewardsAmountPerVault(

                        operatorVaults, totalVaults, epochStartTs, operator, middlewa
        );
```

This means that if a vault is slashed during an epoch, the rewards it deserves should be reduced in the consensus layer of the Tanssi mainnet, but in Symbiotic, its rewards remain unchanged.

# Recommendations

Calculate power using the timestamp starting from the current eraIndex.

# 8.3. Low Findings

## [L-01] `getEpochAtTs()` incorrect when timestamp matches epoch start

The function `OBaseMiddlewareReader.getEpochAtTs()` returns an incorrect epoch number when the input timestamp equals the `epochStart` of a given epoch.

```
function getEpochAtTs(
        uint48 timestamp
    ) public view returns (uint48 epoch) {
        EpochCaptureStorage storage $ = _getEpochCaptureStorage();
        return (timestamp - $.startTimestamp) / $.epochDuration;
    }
```

According to the logic implemented in symbiotic epoch-handling functions (such as `getCurrentEpoch()`), the correct approach is to subtract 1 from the timestamp before performing the division:

```
function getCurrentEpoch() public view returns (uint48) {
        EpochCaptureStorage storage $ = _getEpochCaptureStorage();
        if (_now() == $.startTimestamp) {
            return 0;
        }

        return (_now() - $.startTimestamp - 1) / $.epochDuration;
    }
```

Although no usage of this function is currently identified in critical paths, this incorrect behavior could introduce subtle bugs or inconsistencies for other protocols or off-chain tools relying on accurate epoch identification.

## [L-02] `operatorVaultPairs` array length not adjusted

When fetching operator and vault data, the `getOperatorVaultPairs` function allocates the `operatorVaultPairs` array with a length equal to the total number of operators:

```
function getOperatorVaultPairs(
        uint48 epoch
    ) external view returns
      (IMiddleware.OperatorVaultPair[] memory operatorVaultPairs) {
        uint48 epochStartTs = getEpochStart(epoch);
        address[] memory operators = _activeOperatorsAt(epochStartTs);

>>      operatorVaultPairs = new IMiddleware.OperatorVaultPair[]
  (operators.length);

        uint256 valIdx = 0;
        uint256 operatorsLength_ = operators.length;
        for (uint256 i; i < operatorsLength_;) {
            address operator = operators[i];
            (uint256 vaultIdx, address[] memory _vaults) = getOperatorVaults
              (operator, epochStartTs);

            if (vaultIdx != 0) {
                operatorVaultPairs[valIdx++] = IMiddleware.OperatorVaultPair
                  (operator, _vaults);
            }
            unchecked {
                ++i;
            }
        }
    }
```

However, not all operators may have associated vaults. Data is only populated when an operator has at least one vault. As a result, uninitialized (empty) elements may remain in the array, leading to incomplete or misleading results when returned to the caller.

Consider adjusting the array length with:

```
assembly ("memory-safe") {
            mstore(operatorVaultPairs, valIdx)
        }
```

# [L-03] `ClaimAdminFee()` omits epoch in `ClaimAdminFee` event

The `ClaimAdminFee` event emitted by `claimAdminFee()` lacks the `epoch` parameter. This makes it difficult to track which epoch the claimed admin fee corresponds to, especially when multiple claims happen across epochs or tokens.

```
emit ClaimAdminFee(recipient, tokenAddress, claimableAdminFee_);
```

Consider updating `ClaimAdminFee` event to include the `epoch` value since it is a crucial parameter in the claim.

## [L-04] `Middleware::slash` can revert due to `VetoSlasher` near epoch end

If `Middleware::slash` is called near the end of a vault epoch, it may revert when forwarding the slashing request to a `VetoSlasher`. This happens because `VetoSlasher.requestSlash` includes a strict check that the `captureTimestamp` is far enough ahead to accommodate the full `vetoDuration`:

```
if (
    captureTimestamp < Time.timestamp() + vetoDuration - IVault
      (vault).epochDuration() ||
    captureTimestamp >= Time.timestamp()
) {
    revert InvalidCaptureTimestamp();
}
```

Since `Middleware::slash` is triggered by the `Gateway`, this failure propagates back and causes silent failures. Consider adding a pre-check in `Middleware` to skip slashing attempts if the veto slashing window is invalid.

## [L-05] Constructor of `ODefaultOperatorRewards` missing `disableInitializers()` call

The `ODefaultOperatorRewards` contract inherits from `UUPSUpgradeable`, which is designed for upgradeable deployments via proxies. However, the constructor of this contract does not call `_disableInitializers()`. This can pose a risk as it leaves the `initialize()` function callable by anyone and it is against the recommendation of OZ.

This pattern has been addressed in similar upgradeable contracts (like `Middleware`) where `_disableInitializers()` is correctly called in the constructor.

```
constructor(
        address network,
        address networkMiddlewareService
    ) notZeroAddress(network) notZeroAddress(networkMiddlewareService) {
        i_network = network;
        i_networkMiddlewareService = networkMiddlewareService;
    }
```

Consider adding the `_disableInitializers()` in `ODefaultOperatorRewards::constructor`.

# [L-06] `_beforeRegisterOperator` allows invalid validator registration

The current protocol allows users to register any non-zero validator key, which is then passed as-is to the Substrate chain and stored in its validator set. However, on the Ethereum side, the protocol does not verify whether this Substrate address is valid or resolvable. For example:

- Wrong SS58 checksum.
- Wrong length.
- Invalid Base58 characters.
- Incorrect network prefix.
- Points not on Ed25519 curve.

Make sure that the consensus layer can properly handle these invalid keys. If the consensus layer cannot process these keys, it may lead to a chain halt. Many other well-known protocols have encountered similar issues, such as Ethereum: https://github.com/ethereum/go-ethereum/security/advisories/GHSA-q26p-9cq4-7fc2.

# [L-07] `adminFeeAmount` always round down, resulting in reduced revenue for the admin

ODefaultStakerRewards.sol calculates the admin fee as:

```
//ODefaultStakerRewards.sol::L244
    function _updateAdminFeeAndRewards(
      uint256amount,
      uint256adminFee_,
      uint48epoch,
      addresstokenAddress
    ) private {
        // Take out the admin fee from the rewards
        uint256 adminFeeAmount = amount.mulDiv(adminFee_, ADMIN_FEE_BASE);
        // And distribute the rest to the stakers
        uint256 distributeAmount = amount - adminFeeAmount;
```

This means that during each reward distribution, any portion below `ADMIN_FEE_BASE` is directly discarded. If we consider the expected value, the admin loses an average of `ADMIN_FEE_BASE / 2` in revenue per call.

- For $USDT and $USDC, it equals 5000 / 1e6 = 0.005 USD.
- For $WBTC, it equals 5000 / 1e8 * 1e5 = 5 USD.
- For $STAR, it equals to 5000 / 1e12 = 5e-9 $STAR.

# [L-08] `distributeRewards()` revert

There is currently an edge case when a reward distribution happens to occur in the very first block of an epoch (i.e., `block.timestamp == epochTs`), and a user calls `ODefaultOperatorRewards.claimRewards()` within the same block, the contract treats the reward distribution as if it were in a future block. This causes a valid call that should have succeeded to failed.

```
//ODefaultStakerRewards.sol::L191
        // If the epoch is in the future, revert
        if (epochTs >= Time.timestamp()) {
            revert ODefaultStakerRewards__InvalidRewardTimestamp();
        }
```

As stated in the code comments, the function is supposed to block future epochs, not the current one.

# [L-09] Missing `executeSlash()` after `requestSlash()` for veto slashing

The `_slashVault()` function in the `Middleware` contract handles slashing based on the slasher type. For `SlasherType.VETO`, it correctly calls

`VetoSlasher.requestSlash()` to request a slashing. Then it should call `vetoSlasher.executeSlashe()` after veto phase to finalize the slashing action (document).

```
function _slashVault(
        uint48 timestamp,
        address vault,
        bytes32 subnetwork,
        address operator,
        uint256 amount
    ) private {
        address slasher = IVault(vault).slasher();

        if (slasher == address(0) || amount == 0) {
            return;
        }
        uint256 slasherType = IEntity(slasher).TYPE();

        uint256 response;
        if (slasherType == uint256(SlasherType.INSTANT)) {
            response = ISlasher(slasher).slash
              (subnetwork, operator, amount, timestamp, new bytes(0));
            emit VaultManager.InstantSlash(vault, subnetwork, response);
        } else if (slasherType == uint256(SlasherType.VETO)) {
            response = IVetoSlasher(slasher).requestSlash
              (subnetwork, operator, amount, timestamp, new bytes(0));
            emit VaultManager.VetoSlash(vault, subnetwork, response);
        } else {
            revert VaultManager.UnknownSlasherType();
        }
    }
```

In the current implementation, `executeSlash()` is not implemented or invoked within the `Middleware`, which leaves the slashing incomplete. This gap allows a veto-based slashing to be requested but never executed.

**Note**: In test cases, the `executeSlash()` function is called directly on behalf of the middleware.

Recommendations: Implement and integrate the `executeSlash()` function within the `Gateway` and `Middleware` to complete the veto slashing process.

# [L-10] Middleware may send `bytes32(0)` as validator key

The Middleware contract periodically sends validator data to the gateway:

```
// Decode the sorted keys and the epoch from performData
            (bytes32[] memory sortedKeys, uint48 epoch) = abi.decode
              (performData, (bytes32[], uint48));
            IOGateway(gateway).sendOperatorsData(sortedKeys, epoch);
```

Validator keys are gathered through `getValidatorSet()`, which relies on `getOperatorKeyAt()` to return the validator's key:

```
function getValidatorSet(
        uint48 epoch
    ) public view returns (IMiddleware.ValidatorData[] memory validatorSet) {
        uint48 epochStartTs = getEpochStart(epoch);
        address[] memory operators = _activeOperatorsAt(epochStartTs);
        validatorSet = new IMiddleware.ValidatorData[](operators.length);

        uint256 len = 0;
        uint256 operatorsLength_ = operators.length;
        for (uint256 i; i < operatorsLength_;) {
            address operator = operators[i];
            unchecked {
                ++i;
            }
>>          bytes32 key = abi.decode(getOperatorKeyAt(operator, epochStartTs),
  (bytes32));
            uint256 power = _getOperatorPowerAt(epochStartTs, operator);
            validatorSet[len++] = IMiddleware.ValidatorData(power, key);
        }

        // shrink array to skip unused slots
        assembly ("memory-safe") {
            mstore(validatorSet, len)
        }
    }
```

However, `getOperatorKeyAt()` can return `bytes32(0)` under certain edge cases, e.g., when an operator's key is disabled using `Operators.updateOperatorKey()` and no other active key remains. This results in a zero key being encoded and included in the validator set sent to the gateway—potentially violating assumptions downstream. Worst-case scenario: some rewards may be allocated to a zero key validator, making them unclaimable.

```
function getOperatorKeyAt
      (address operator, uint48 timestamp) public view returns (bytes memory) {
        KeyManager256Storage storage $ = _getKeyManager256Storage();
        bytes32 key = $._key[operator];
        if (key != bytes32(0) && $._keyData[key].status.wasActiveAt
          (timestamp)) {
            return abi.encode(key);
        }
        key = $._prevKey[operator];
        if (key != bytes32(0) && $._keyData[key].status.wasActiveAt
          (timestamp)) {
            return abi.encode(key);
        }
>>      return abi.encode(bytes32(0));
    }
```

Recommendations:

Filter out zero-value keys before inclusion in the validator set:

```
function getValidatorSet(
        uint48 epoch
    ) public view returns (IMiddleware.ValidatorData[] memory validatorSet) {
        uint48 epochStartTs = getEpochStart(epoch);
        address[] memory operators = _activeOperatorsAt(epochStartTs);
        validatorSet = new IMiddleware.ValidatorData[](operators.length);

        uint256 len = 0;
        uint256 operatorsLength_ = operators.length;
        for (uint256 i; i < operatorsLength_;) {
            address operator = operators[i];
            unchecked {
                ++i;
            }
            bytes32 key = abi.decode(getOperatorKeyAt(operator, epochStartTs),
              (bytes32));
            uint256 power = _getOperatorPowerAt(epochStartTs, operator);
+           if(key != bytes32(0) {
              validatorSet[len++] = IMiddleware.ValidatorData(power, key);
+           }
        }

        // shrink array to skip unused slots
        assembly ("memory-safe") {
            mstore(validatorSet, len)
        }
    }
```

# [L-11] Vault rewards distribution fails without `ODefaultStakerRewards`

In the `_distributeRewardsPerVault` function of the `ODefaultOperatorRewards` contract, the logic assumes that all vaults listed in `operatorVaults` have corresponding staker reward contracts set in the

`vaultToStakerRewardsContract` mapping. However, this assumption is incorrect for operator specific vaults that got registered through `Operators::registerOperatorVault` :

```
function _distributeRewardsPerVault(
         // ...
    ) private {
        OperatorRewardsStorage storage $ = _getOperatorRewardsStorage();
        for (uint256 i; i < totalVaults;) {
@>
              address stakerRewardsForVault = $.vaultToStakerRewardsContract[operatorV
            IERC20(tokenAddress).approve
              (stakerRewardsForVault, amountPerVault[i]);
            IODefaultStakerRewards(stakerRewardsForVault).distributeRewards(
                epoch, eraIndex, amountPerVault[i], tokenAddress, data
            );

            // ...
        }
    }
```

This is happening because `operatorVaults` array passed in the `_distributeRewardsPerVault` is retrieved using :

```
IOBaseMiddlewareReader(middlewareAddress).getOperatorVaults
  (operator, epochStartTs);
```

Which returns all the vaults of the operator (both shared vaults and specific ones). But specific ones don't have `vaultToStakerRewardsContract`. So, even if, as per sponsor message "**There might be a possibility for registerOperatorVault**, but for now we don’t have an objective for it.", it would be good to consider that registration of operator vaults will completely DoS `ODefaultOperatorRewards::claimRewards` calls.

Recommendations: Consider getting only the active shared vaults from the operator and make the reward distribution only for these vaults that actually have `vaultToStakerRewardsContract` contracts.

# [L-12] Missing validation for stale or invalid price data in `Middleware::stakeToPower`

The `stakeToPower` function fetches the latest price from a `Chainlink` oracle via `AggregatorV3Interface(oracle).latestRoundData()`, but it does not perform any validation to ensure that the returned price data is fresh, valid, and not stale. Chainlink feeds can return outdated or incomplete data under certain

conditions, such as oracle network delays or failures, which may result in incorrect or manipulated power calculations.

```
function stakeToPower(
      addressvault,
      uint256stake
    ) public view override returns (uint256 power
        address collateral = vaultToCollateral(vault);
        address oracle = collateralToOracle(collateral);

        if (oracle == address(0)) {
            revert Middleware__NotSupportedCollateral(collateral);
        }
@>        (, int256 price,,,) = AggregatorV3Interface(oracle).latestRoundData();
        uint8 priceDecimals = AggregatorV3Interface(oracle).decimals();
        power = stake.mulDiv(uint256(price), 10 ** priceDecimals);
        // Normalize power to 18 decimals
        uint8 collateralDecimals = IERC20Metadata(collateral).decimals();
        if (collateralDecimals != DEFAULT_DECIMALS) {
            power = power.mulDiv
              (10 ** DEFAULT_DECIMALS, 10 ** collateralDecimals);
        }
    }
```

Recommendations: Implement standard Chainlink security validations like :

```
(, int256 price, , uint256 updatedAt,) = AggregatorV3Interface
  (oracle).latestRoundData();

require(price > 0, "Invalid oracle price");
require(updatedAt >= block.timestamp - MAX_STALE_TIME, "Stale price data");
```

# [L-13] Middleware treats validators with very low active power as valid

Each time `middleware` sends a message to the Tanssi consensus layer via `performUpkeep` or `sendCurrentOperatorsKeys`, it iterates over all validators with active stake power and sorts them.

Tanssi does not care about the exact power of each validator—only their relative ordering. The problem is that there is currently no mechanism to ensure a validator's active power is sufficient. When a validator has very low power and appears at the bottom of the list, it can still function normally on the Substrate chain. This means that some validators in the network may not be properly staked, and any slashing applied to them would also be minimal. This undermines the security of the network from an economic perspective.

Recommendations: Add a minimum threshold in `_quickSort`, discard validators with low stake.

# [L-14] Pending slashes are not taken into account

`Middleware.sol` uses `IBaseDelegator(IVault(vault).delegator()).stakeAt()` to decide the max slashable amount:

```
uint256 vaultStake = IBaseDelegator(IVault(vault).delegator()).stakeAt(
            subnetwork, params.operator, params.epochStartTs, new bytes(0)
        );
```

When using a VetoSlasher, pending slashes are not reflected in the `stake()`/`stakeAt()` calls. This is explicitly mentioned in the Symbiotic documentation:

> "You are aware that NetworkRestakeDelegator.stakeAt()/FullRestakeDelegator.stakeAt() (the function returning a stake for an Operator in your Network) counts the whole existing money as a real stake, meaning that it doesn't take into account pending slashings from you or your 'neighbor' Networks. Hence, it should be covered by your Middleware, depending on the Vault's type and your slashings' processing logic."

This can lead to several issues:

- Since slashing is applied as a percentage, if a validator is slashed multiple times within a single epoch (which is possible because the Tanssi chain executes the validator's maximum slash from the previous era(capped at 75%) at the start of each era), and an epoch contains multiple eras, then both slashes may end up using the same stake amount. If the time between vetoSlasher.requestSlash and vetoSlasher.executeSlash spans more than one era, both slashes might be executed based on the stake from the first slash. If the combined slash percentage exceeds 50%, the second slash may fail.
- Validators may be considered opted-in and slashable when they should not be, due to pending slashings not being accounted for in the stake calculation.

Recommendations:

It's not possible/feasible to track pending slashes from external networks. This is simply a weakness of the symbiotic protocol.

# [L-15] Vaults can maliciously escaping slashing

For vaults that use burner routers, the following exploit scenario is possible: a vault can create a custom network and specify its own receiver for that network, which then transfers funds to an address it controls. If this vault also acts as a validator, it could behave maliciously. Upon detecting/confirming its own misbehavior within the Tanssi network, the vault could immediately slash itself within its custom network, transferring collateral away before the Tanssi network has a chance to enforce its own slashing — thereby avoiding any penalty from Tanssi.

Because of this, Tanssi network must fully trust all vaults integrated with `middleware`. Any vault has the ability to evade any slash, although it may not necessarily have the incentive to do so.

Recommendations:

To alleviate this issue, a feasible solution is to enforce a minimum burner router delay, allowing time for off-chain monitoring services to observe pending changes in vault state, and try to detect malicious behavior. Failed slashing attempts should also be monitored off-chain, and if a vault is found to be maliciously escaping slashing, they should be immediately deregistered, hopefully prior to enacting the malicious behavior with multiple validators.

# [L-16] Recursive QuickSort will always fail for some worst-case scenarios

```
File: contracts/libraries/QuickSort.sol
34:     function _quickSort(
  IMiddleware.ValidatorData[]memoryarr,
  int256left,
  int256right
) public pure {
[...]
48:         if (left < j) {
49:             _quickSort(arr, left, j);
50:         }
51:         if (i < right) {
52:             _quickSort(arr, i, right);
53:         }
```

The protocol implements a recursive version of QuickSort algorithm. While, the recursion depth for the average case is estimated to `log(n)`, for some edge cases, the depth will be close to `n`. Those cases occurs when pivot is always the smallest or largest element (e.g., when the array is already sorted or reverse sorted). The Solidity language is limited by the maximal stack depth of 1024 - which implies, that it won't be possible to call `_quickSort()` function on some lists which has more than 1024 elements.

To avoid the described issues and still remain the `O(n*log(n))` time complexity, it's recommended to switch to the iterative QuickSort algorithm.

This issue was evaluated as Low - as it's very unlikely not to hit a gas limit while sorting an array with more than 1024 elements. The code which utilized this function:

```
/**

            * @dev Sorts operators by their total power in descending order, after 500
[...]
        validatorSet = validatorSet.quickSort(0, int256
          (validatorSet.length - 1));
```

implies, that the developers are aware of this limitations. However, out of gas and max. stack depth limitations are not the same root-cause, thus this issue had been separately reported for the record.

# [L-17] `performUpkeep()` may exceed Chainlink's `maxPerformDataSize`

According to Chainlink documentation the maximum size in bytes that can be sent to `performUpkeep` function is `2000` bytes.

Within the `performUpkeep()` implementation, `performData` contains the sorted keys and the epoch from `performData`.

```
File: contracts/middleware/Middleware.sol
316:     function performUpkeep(
317:         bytes calldata performData
318:     ) external override checkAccess {
319:         StorageMiddleware storage $ = _getMiddlewareStorage();
320:         address gateway = $.gateway;
321:         if (gateway == address(0)) {
322:             revert Middleware__GatewayNotSet();
323:         }
324:
325:         uint48 currentTimestamp = Time.timestamp();
326:         if ((currentTimestamp - $.lastTimestamp) > $.interval) {
327:             $.lastTimestamp = currentTimestamp;
328:
329:             // Decode the sorted keys and the epoch from performData
330:             (bytes32[] memory sortedKeys, uint48 epoch) = abi.decode
  (performData, (bytes32[], uint48));
331:
332:             IOGateway(gateway).sendOperatorsData(sortedKeys, epoch);
333:         }
334:     }
```

Chainlink Keepers are also constrained by a strict maximum gas usage, which impacts their ability to execute high-cost operations such as `sortOperatorsByPower()`. The development team is aware of the significant gas consumption associated with this function, as noted in the code comments:

```
File: contracts/middleware/OBaseMiddlewareReader.sol
521:     /**
522:
       * @dev Sorts operators by their total power in descending order, after 500 it
523:       * @param epoch The epoch number
524:
       * @return sortedKeys Array of sorted operators keys based on their power
525:       */
526:
```

However, there is an additional constraint that appears to be overlooked. Chainlink enforces a `maxPerformDataSize` limit, which imposes a stricter cap on the number of operators that can be handled. Since each operator key is represented as a `bytes32` value, the total number of keys that can be included is limited to: 2000 / 32 = 62 keys.

This results in a practical upper limit of 62 operators, which is significantly lower than the 500-operator estimate referenced in the comments.

The Middleware does not implement any mechanism which limits the number of operators, thus when that number becomes too big (there would be too many operators) - the Chainlink's `maxPerformDataSize` will be reached.

**Recommendations**

Implement a mechanism which limits the number of operators.