Satoshi Protocol v2 Security Review

Reviewer

billh (@hibillh)

Draft Version: January 5, 2025
Final Version: February 12, 2025

1 Executive Summary

Over the course of 13 days in total, Satoshi Protocol engaged with billh to review Satoshi Protocol v2.

A total of 25 issues have been found with Satoshi Protocol v2.

Repository	Commit
satoshi-staking-vault	9bce79582b642f2a57cfb30bace2e755758adb9d
satoshi-v2	853c27c48a33b1ac3c8890d91dad4bec2dd6b00e

Summary

Type of Project	CDP Protocol
Timeline	Dec 24, 2024 - Jan 4, 2025
Methods	Manual Review

Total Issues

High Risk	5
Medium Risk	7
Low Risk	4
Informational	9

Contents

Disclaimer: This security review is not a guarantee against hacking. This is the result of a review within a time range based on specific commits.

2 Findings

2.1 High Risk

2.1.1 Unauthorized Access in NexusYieldManagerFacet Functions

Severity: High

Context: NexusYieldManagerFacet.sol#L69, NexusYieldManagerFacet.sol#L343, NexusYieldManagerFacet.sol#L357, NexusYieldManagerFacet.sol#L371

Description:

Multiple functions in `NexusYieldManagerFacet` lack proper permission checks, potentially allowing unauthorized access to critical functionality. This includes the following functions:

- sunsetAsset
- pause
- resume
- setPrivileged

Recommendation:

Add appropriate access control checks to these sensitive functions.

Status:

Fixed in PR#5.

2.1.2 Unreturned Debt Tokens in `SatoshiPeriphery` Redemption

Severity: High

Context: SatoshiPeriphery.sol#L233-L263

Description:

In the `SatoshiPeriphery` contract's `redeemCollateral` function, there's a potential issue where debt tokens may not be fully consumed during redemption.

When a user redeems collateral, they provide a specific amount of debt tokens (`_debtAmount`), but the actual redemption might use less than this amount.

However, the function doesn't handle the return of unused debt tokens to the user. After calling `troveManager.redeemCollateral()`, the function only processes the received collateral through `_afterWithdrawColl()` but doesn't check or return any remaining debt tokens. This could lead to users losing their unused debt tokens.

```
        troveManager.redeemCollateral(
            _debtAmount,
            _firstRedemptionHint,
            _upperPartialRedemptionHint,
            _lowerPartialRedemptionHint,
            _partialRedemptionHintNICR,
            _maxIterations,
            _maxFeePercentage
        );

        uint256 collTokenBalanceAfter =
         ↪  collateralToken.balanceOf(address(this));
        uint256 userCollAmount = collTokenBalanceAfter -
         ↪  collTokenBalanceBefore;

        _afterWithdrawColl(collateralToken, userCollAmount);
    }
```

Recommendation:

Track and return any unused debt tokens to the user.

Status:

The function is removed.

2.1.3 Incorrect Collateral Management in `TroveManager`'s Send Function

Severity: High

Context: TroveManager.sol#L1150-L1172

Description:

The `_sendCollateral` function in `TroveManager` contains two calculation errors that could disrupt protocol operations:

1. The boundary calculation uses the wrong base amount. It should use

`totalActiveCollateral - _amount` to calculate the new boundary after withdrawal. This incorrect calculation could cause the subsequent refill amount calculation (`_amount + target - remainColl`) to result in a negative value and revert, preventing legitimate collateral withdrawals.

2. The refill amount calculations are inconsistent between the two conditions. In the first condition, `SatoshiMath._min(target, collateralOutput)` refills `target` amount, while in the second condition, `_amount + target - remainColl` calculates how much needs to be refilled up to the `target`. This inconsistency in logic could lead to incorrect refill amounts and suboptimal collateral management.

These issues could prevent users from liquidating or adjusting their positions when the contract's collateral is deployed in farming strategies, potentially putting user funds at risk during market volatility.

Recommendation:

Modify the function to use consistent calculations and proper base amounts:

```
    function _sendCollateral(address _account, uint256 _amount) private {
        uint256 newTotal = totalActiveCollateral - _amount;
        uint256 boundary = newTotal * farmingParams.retainPercentage /
         ↪  FARMING_PRECISION;
        uint256 remainColl = totalActiveCollateral - collateralOutput;
        uint256 target = newTotal * farmingParams.refillPercentage /
         ↪  FARMING_PRECISION;

        // remain collateral is not enough, must refill
        if (_amount > remainColl) {
            vaultManager.exitStrategyByTroveManager(_amount - remainColl);
            // refill to target
            uint256 refillAmount = SatoshiMath._min(target + _amount -
             ↪  remainColl, collateralOutput);
            if (refillAmount != 0)
             ↪  vaultManager.exitStrategyByTroveManager(refillAmount);
        } else if (remainColl - _amount < boundary) {
            uint256 refillAmount = _amount + target - remainColl;
            vaultManager.exitStrategyByTroveManager(refillAmount);
        }

                ...
    }
```

Status:

Fixed in PR#5.

2.1.4 Potential Revert in CDPVaults Due to Excessive Withdrawal Amount

Severity: High

Context: VaultManager.sol#L81-L82

Description:

The `exitStrategy` function in `AvalonVault` and `PellVault` may revert when the requested `amount` exceeds the available balance. This is problematic because the `VaultManager.exitStrategyByTroveManager` function passes the full `withdrawAmount` to each vault without checking if the vault has sufficient funds. The issue could prevent users from liquidating or adjusting their positions when the contract's collateral is deployed in farming strategies, potentially putting user funds at risk during market volatility.

```
function exitStrategyByTroveManager(uint256 amount) external {
    require(msg.sender == troveManager, "VaultManager: Caller is not
    ↪  TroveManager");
    if (amount == 0) return;

    // assign a value to balanceAfter to prevent the priority being empty
    uint256 balanceAfter = collateralToken.balanceOf(address(this));
    uint256 withdrawAmount = amount;
    for (uint256 i; i < priority.length; i++) {
        if (balanceAfter >= amount) break;
        INYMVault vault = priority[i];
        bytes memory data =
        ↪  vault.constructExitStrategyData(withdrawAmount);
        uint256 exitAmount = vault.exitStrategy(data);
        collateralAmounts[address(vault)] -= exitAmount;
        withdrawAmount -= exitAmount;
        balanceAfter = collateralToken.balanceOf(address(this));

        emit ExitStrategy(address(vault), exitAmount);
    }
    ...
}
```

Recommendation:

Modify CDP vaults to handle partial withdrawals by checking the available balance first.

Status:

Fixed in PR#5.

2.1.5 Incomplete EigenLayer Integration in PellVault

Severity: High

Context: PellVault.sol#L30-L53

Description:

The `PellVault` contract has several issues related to its integration with the EigenLayer fork:

1. **Missing Delegation Setup**: In `executeStrategy`, tokens are deposited without setting up delegation, which is a core component for EigenLayer-style protocols. Without delegation, the deposited tokens may not participate in the protocol's staking mechanism properly.
2. **Incomplete Withdrawal Process**: The `exitStrategy` function only queues withdrawals through `queueWithdrawals` but doesn't complete them with `completeQueuedWithdrawal`. This means tokens will remain locked in the withdrawal queue and cannot be accessed by the protocol.
3. **Inaccurate Position Reporting**: The `getPosition` function uses `userUnderlyingView` which doesn't account for queued withdrawals. This could lead to incorrect reporting of available positions, especially when withdrawals are pending in the queue.
4. **Incorrect Share Calculation**: The withdrawal process uses the collateral amount directly as share amount without proper conversion, which could lead to incorrect withdrawal amounts since EigenLayer protocols use shares to represent positions.

```
    function executeStrategy(bytes calldata data) external override
     ↪  onlyVaultManager {
        uint256 amount = _decodeExecuteData(data);
        IERC20(TOKEN_ADDRESS).approve(strategyAddr, amount);
        // deposit token to lending

         ↪  IStrategyManager(strategyAddr).depositIntoStrategy(IStrategy(pellStrategy),
         ↪  IERC20(TOKEN_ADDRESS), amount);
    }

    function exitStrategy(bytes calldata data) external override
     ↪  onlyVaultManager returns (uint256) {
        uint256 amount = _decodeExitData(data);
        IStrategy[] memory strategies = new IStrategy[](1);
        strategies[0] = IStrategy(pellStrategy);

        uint256[] memory shares = new uint256[](1);
        shares[0] = amount;

        QueuedWithdrawalParams[] memory queuedWithdrawal = new
         ↪  QueuedWithdrawalParams[](1);
        queuedWithdrawal[0] =
            QueuedWithdrawalParams({strategies: strategies, shares: shares,
             ↪  withdrawer: address(this)});

        // withdraw token from lending

         ↪  IDelegationManager(0x230B442c0802fE83DAf3d2656aaDFD16ca1E1F66).queueWithdrawal

        return amount;
    }
```

Recommendation:

1. Implement proper delegation setup in `executeStrategy`.
2. Complete the withdrawal process by adding `completeQueuedWithdrawal` call after the queue delay period.
3. Modify `getPosition` to include both active and queued positions. For queued positions, `sharesToUnderlyingView` can be used.
4. Convert between shares and underlying amounts correctly using the strategy's conversion functions.
5. Consider implementing a withdrawal queue tracking system to manage pending withdrawals.

Status:

Fixed in PR#5.

2.2 Medium Risk

2.2.1 Trapped Yield in VaultManager's Accounting System

Severity: Medium

Context: VaultManager.sol#L48-L96

Description:

The `VaultManager`'s accounting system tracks deposits and withdrawals using the `collateralAmounts` mapping, which is updated based on the exact amounts deposited and withdrawn. However, this creates an issue where any yield or rewards generated by the strategy cannot be withdrawn because withdrawals are limited by the originally deposited amount. This is particularly problematic for protocols like Eigenlayer where strategy shares might not be transferable and become effectively locked due to the accounting system. The yield remains trapped in the vault, unable to be accessed through normal withdrawal mechanisms.

```
    function executeStrategy(address vault, uint256 amount) external onlyOwner
     ↪  {
        _checkWhitelistedVault(vault);

        collateralAmounts[vault] += amount;
                ...
    }

    function exitStrategy(address vault, uint256 amount) external onlyOwner {
        _checkWhitelistedVault(vault);

        bytes memory data = INYMVault(vault).constructExitStrategyData(amount);
        INYMVault(vault).exitStrategy(data);

        collateralAmounts[vault] -= amount;

        emit ExitStrategy(vault, amount);
    }
```

Recommendation:

Consider tracking strategy shares or positions instead of raw amounts, or implement a separate accounting system for yield that allows the protocol to withdraw both principal and generated yields.

Status:

Fixed in PR#5.

2.2.2 Parameter Mismatch and Incorrect Documentation in NexusYieldManagerFacet's Swap Functions

Severity: Medium

Context: NexusYieldManagerFacet.sol#L241-L283, uniswapV2Vault.sol#L46-L51

Description:

There is a parameter type mismatch between `NexusYieldManagerFacet`'s `swapOutPrivileged` function and its usage in `UniswapV2Vault`. The function expects a debt token amount, but receives an asset amount instead. Additionally, the NatSpec documentation for this function is misleading as it incorrectly describes the `amount` parameter's purpose and type.

```
        uint256 previewAmount =
         ↪  INexusYieldManagerFacet(nymAddr).convertDebtTokenToAssetAmount(
            STABLE_TOKEN_ADDRESS, IERC20(SAT_ADDRESS).balanceOf(address(this))
        );
        uint256 swapOutAmount =

            ↪  INexusYieldManagerFacet(nymAddr).swapOutPrivileged(STABLE_TOKEN_ADDRESS,
            ↪  address(this), previewAmount);
```

This dual issue could lead to two problems:

1. Only partial debt tokens are swapped for stable tokens.
2. The transaction might revert due to insufficient balance.

Recommendation:

1. Update the NatSpec documentation to clearly specify that the `amount` parameter represents debt token amount.
2. Modify the `UniswapV2Vault` implementation to use debt token amount.

Status:

The contract is removed.

2.2.3 Insufficient Surplus Balances Handling in `TroveManager`

Severity: Medium

Context: TroveManager.sol#L750-L751, TroveManager.sol#L774-L785, TroveManager.s
L1071

Description:

`TroveManager` has an oversight in how `surplusBalances` is handled. While `claimCollateral` checks if a user has claimable collateral through `surplusBalances`, it doesn't verify if the contract actually has sufficient collateral tokens to fulfill the claim. This is problematic because some collateral might be stored in the `VaultManager`, making it temporarily unavailable in the `TroveManager` contract.

```
    function claimCollateral(address _receiver) external {
        uint256 claimableColl = surplusBalances[msg.sender];
        require(claimableColl > 0, "No collateral available to claim");

        surplusBalances[msg.sender] = 0;

        collateralToken.safeTransfer(_receiver, claimableColl);

        if (interestPayable > 0) {
            collectInterests();
        }
    }
```

Recommendation:

Implement similar logic as `_sendCollateral` when adding surplus balance for a user.

Status:

Fixed in PR#5.

2.2.4 Incorrect Reward Integration Update in SatUSDVault

Severity: Medium

Context: SatUSDVault.sol#L265-L273

Description:

The `rewardIntegral` is updated with new values in the `_updateRewardIntegral` function, but rewardParamsMap is updated using stale data. This inconsistency in reward calculations can lead to incorrect reward distributions and potential economic losses for users.

```
    function _setRewardParams(address token, uint256 rewardRate, uint256
     ↪  rewardIntegral, uint32 claimStartTime)
        internal
    {
        _updateRewardIntegral(token, totalSupply());
        rewardParamsMap[token] =
            RewardParams({rewardRate: rewardRate, rewardIntegral:
             ↪  rewardIntegral, claimStartTime: claimStartTime});

        emit RewardParamsSet(token, rewardRate, rewardIntegral,
         ↪  claimStartTime);
    }
```

Recommendation:

Update rewardParamsMap using the newly calculated rewardIntegral value instead of the old one.

Status:

Fixed.

2.2.5 Double Withdrawal Vulnerability in RewardVault

Severity: Medium

Context: RewardVault.sol#L45-L53, RewardVault.sol#L65-L74

Description:

In the RewardVault contract, there is a issue where allocated users or contracts can potentially withdraw twice their allocated amount through the interaction of collectUnderlying and transferAllocatedTokens functions. The collected amount is increased in collectUnderlying, but this conflicts with the usage in transferAllocatedTokens, allowing a user to withdraw up to double their entitled amount. This is particularly concerning if the allocation target is an uncontrolled contract.

```
    function transferAllocatedTokens(address asset, address receiver, uint256
     ↪  amount) external nonReentrant {
        if (amount > 0) {
            require(collected[asset][msg.sender] >= amount, "Reward Vault:
             ↪  Insufficient balance");
            collected[asset][msg.sender] -= amount;
            IERC20(asset).safeTransfer(receiver, amount);

            emit TransferAllocatedTokens(asset, receiver, amount);
        }
    }

        function collectUnderlying(address asset, uint256 amount) external
         ↪  nonReentrant {
        if (amount > 0) {
            require(allocated[asset][msg.sender] >= amount, "Reward Vault:
             ↪  Insufficient balance");
            allocated[asset][msg.sender] -= amount;
            collected[asset][msg.sender] += amount;
            IERC20(asset).safeTransfer(msg.sender, amount);

            emit CollectUnderlying(asset, msg.sender, amount);
        }
    }
```

Recommendation:

Remove the `collected` increment in `collectUnderlying` to prevent double withdrawal possibility.

Status:

Fixed.

2.2.6 Incorrect Exchange Rate Calculation in Withdraw/Redeem

Severity: Medium

Context: SatUSDVault.sol#L120-L145

Description:

In the `SatUSDVault` contract, the transfer amount in the withdraw/redeem functions is calculated before updating integrals. This sequence leads to an incorrect exchange rate calculation as it doesn't account for unclaimed underlying tokens. The could result in users receiving less amounts during withdrawals or redemptions.

```
    function redeem(uint256 shares, address receiver, address owner)
        public
        override(ERC4626Upgradeable)
        whenNotPaused
        returns (uint256)
    {
        uint256 balance = balanceOf(msg.sender);
        uint256 supply = totalSupply();
        uint256 amount = super.redeem(shares, receiver, owner);
        _updateIntegrals(msg.sender, balance, supply);
        return amount;
    }
```

Recommendation:

Update the integrals before calculating the transfer amount to ensure all unclaimed underlying tokens are properly accounted for in the exchange rate calculation.

Status:

Fixed.

2.2.7 Short-Calldata Zero-Padding in Diamond Proxy Allows Unexpected Fallback Execution

Severity: Medium

Context: DiamondBase.sol#L18-L33

Description:

A subtle edge case exists in SolidState's Diamond proxy implementation (an EIP-2535 implementation) where any `msg.data.length < 4` call leads to zero-padding of `msg.sig`. For example, calldata of just `0xff` becomes `0xff000000`. If a facet with that selector (`0xff000000`) is added (e.g., via governance), and its fallback function contains certain logic, it could be unintentionally or maliciously triggered—potentially resulting in unexpected state changes.

Additionally, if `DiamondFallback` is set with a fallback contract address, the expected fallback logic would not be executed under above circumstances.

```
    function _getImplementation()
        internal
        view
        virtual
        override
        returns (address implementation)
    {
        // inline storage layout retrieval uses less gas
        DiamondBaseStorage.Layout storage l;
        bytes32 slot = DiamondBaseStorage.STORAGE_SLOT;
        assembly {
            l.slot := slot
        }

        implementation = address(bytes20(l.selectorInfo[msg.sig]));
    }
```

```
contract TestFacet {
    // The function selector is 0xff000000
    function BlazingIt6886408584() public payable {}

    fallback() external {
        // Potentially malicious or unexpected logic
    }
}

...

// Sending only one byte ("0xff") triggers the delegatecall to TestFacet
// and fallback would be executed
address(satoshiXApp).call(hex"ff");
```

Recommendation:

You could mitigate the issue by overwriting `_getImplementation()` with additional `msg.data.length < 4` check and default the implementation address to `address(0)` in those cases. Alternatively, remain aware of this risk and ensure that any newly added facet does not contain a fallback function that could be unintentionally triggered.

Status:

Fixed in PR#5.

2.3 Low Risk

2.3.1 Excessive Withdrawal Amount in VaultManager

Severity: Low

Context: VaultManager.sol#L71-L96

Description:

In `exitStrategyByTroveManager`, the initial `withdrawAmount` is set to the full `amount` requested without considering the existing balance in the contract. This means if the contract already has some tokens, it will still try to withdraw the full amount from strategies, potentially withdrawing more tokens than necessary. For example, if 100 tokens are requested and the contract already has 30 tokens, it should only need to withdraw 70 tokens from strategies, but currently it tries to withdraw all 100.

Additionally, the `withdrawAmount` calculation could potentially underflow if a strategy's `exitStrategy` returns more than requested due to rounding or implementation specifics, which could happen with certain yield-bearing tokens or strategies that handle decimals differently.

```
    function exitStrategyByTroveManager(uint256 amount) external {
        require(msg.sender == troveManager, "VaultManager: Caller is not
         ↪  TroveManager");
        if (amount == 0) return;

        // assign a value to balanceAfter to prevent the priority being empty
        uint256 balanceAfter = collateralToken.balanceOf(address(this));
        uint256 withdrawAmount = amount;
        for (uint256 i; i < priority.length; i++) {
            if (balanceAfter >= amount) break;
            INYMVault vault = priority[i];
            bytes memory data =
             ↪  vault.constructExitStrategyData(withdrawAmount);
            uint256 exitAmount = vault.exitStrategy(data);
            collateralAmounts[address(vault)] -= exitAmount;
            withdrawAmount -= exitAmount;
            balanceAfter = collateralToken.balanceOf(address(this));

            emit ExitStrategy(address(vault), exitAmount);
        }

        // if the balance is still not enough
        uint256 actualTransferAmount = balanceAfter >= amount ? amount :
         ↪  balanceAfter;

        // transfer token to TroveManager
        collateralToken.approve(troveManager, actualTransferAmount);

         ↪  ITroveManager(troveManager).receiveCollFromPrivilegedVault(actualTransferAmoun
    }
```

Recommendation:

1. Initialize `withdrawAmount` as `max(amount - balanceAfter, 0)` to only withdraw what's actually needed.
2. Add underflow protection when updating `withdrawAmount`.

Status:

Fixed in PR#5.

2.3.2 Incorrect Token Address in AvalonVault Position Query

Severity: Low

Context: avalonVault.sol#L58-L60

Description:

In `AvalonVault`'s `getPosition` function, there's an assumption that the aToken (interest-bearing token) address is the same as the lending pool address (`strategyAddr`). However, in Aave V3 and its forks, the aToken address is different from the pool address. This means the function is querying the balance from the wrong contract address, which will likely revert.

```
    function getPosition() external view override returns (address, uint256) {
        return (TOKEN_ADDRESS, IERC20(strategyAddr).balanceOf(address(this)));
    }
```

Recommendation:

Using the pool's `getReserveData` function to get the correct aToken address.

Status:

Fixed in PR#5.

2.3.3 DOS Vulnerability in UniswapV2Vault Balance Check

Severity: Low

Context: uniswapV2Vault.sol#L29

Description:

The `UniswapV2Vault` contract uses a strict equality check (==) to verify the SAT token balance. This creates a potential denial-of-service (DOS) vulnerability as malicious actors could prevent legitimate transactions by continuously sending a small amount of SAT tokens (e.g., 1 wei) directly to the contract. When this happens, the balance check will fail because the actual balance will be higher than the expected `amountB`.

```
    function executeStrategy(bytes calldata data) external override onlyOwner {
        (uint256 amountA, uint256 amountB, uint256 minA, uint256 minB) =
        ↪  _decodeExecuteData(data);
        // swap stable to sat in nym
        IERC20(STABLE_TOKEN_ADDRESS).approve(nymAddr, amountA);
        INexusYieldManagerFacet(nymAddr).swapInPrivileged(STABLE_TOKEN_ADDRESS,
        ↪  address(this), amountA);
        require(IERC20(SAT_ADDRESS).balanceOf(address(this)) == amountB,
        ↪  "balance not match");

                ...
    }
```

Recommendation:

Replace the strict equality check with a greater-than-or-equal check.

Status:

The contract is removed.

2.3.4 Data Type Mismatch in UniswapV2Vault Strategy Construction

Severity: Low

Context: uniswapV2Vault.sol#L56-L58, uniswapV2Vault.sol#L24-L25

Description:

In the `UniswapV2Vault` contract there's a mismatch between the data types used in `executeStrategy` and `constructExecuteStrategyData`. The `executeStrategy` function expects four parameters (`amountA`, `amountB`, `minA`, `minB`) for Uniswap V2 liquidity provision, but `constructExecuteStrategyData` only encodes a single `amount` parameter. This inconsistency will cause the `_decodeExecuteData` function to revert when trying to decode the incorrectly formatted data.

```
    function executeStrategy(bytes calldata data) external override onlyOwner {
        (uint256 amountA, uint256 amountB, uint256 minA, uint256 minB) =
         ↪   _decodeExecuteData(data);

        ...
        }

        function constructExecuteStrategyData(uint256 amount) external pure
         ↪   override returns (bytes memory) {
        return abi.encode(amount);
    }
```

Recommendation:

Since `constructExecuteStrategyData` is defined in the interface with a single parameter, create a new function for proper data construction and make the original function revert:

```
function constructExecuteStrategyData(uint256 amount) external pure override
 ↪   returns (bytes memory) {
    revert("Use constructUniswapV2StrategyData instead");
}

function constructUniswapV2StrategyData(
    uint256 amountA,
    uint256 amountB,
    uint256 minA,
    uint256 minB
) external pure returns (bytes memory) {
    return abi.encode(amountA, amountB, minA, minB);
}
```

Also add documentation to clearly indicate that `constructUniswapV2StrategyData` should be used instead of the interface function.

Status:

The contract is removed.

2.4 Informational

2.4.1 Incompatible with weird ERC20

Severity: Informational

Context: NexusYieldManagerFacet.sol#L87-L87, VaultCore.sol#L52-L52

Description:

Although `safeERC20` library is imported, the `NexusYieldManagerFacet` contract and `VaultCore` contract still use the unsafe `transfer` method instead of OpenZeppelin's `safeTransfer` for ERC20 token transfers. Some ERC20 tokens (like USDT) don't follow the standard strictly and may return `false` on failure instead of reverting.

```
    function transferTokenToPrivilegedVault(address token, address vault,
     ↪  uint256 amount) external onlyRole(Config.OWNER_ROLE) {
        AppStorage.Layout storage s = AppStorage.layout();
        if (!s.isPrivileged[vault]) {
            revert NotPrivileged(vault);
        }
        IERC20(token).transfer(vault, amount);
        emit TokenTransferred(token, vault, amount);
    }
```

Also, the `UniswapV2Vault` contract directly sets token approvals without first resetting them to zero. Some ERC20 tokens (like USDT) implement additional safety measures against approval race conditions that require setting the allowance to 0 before changing it to a new value. For these tokens, the current approval pattern will fail, making the vault incompatible with such tokens.

```
    function executeStrategy(bytes calldata data) external override onlyOwner {
        (uint256 amountA, uint256 amountB, uint256 minA, uint256 minB) =
         ↪  _decodeExecuteData(data);
        // swap stable to sat in nym
        IERC20(STABLE_TOKEN_ADDRESS).approve(nymAddr, amountA);
        INexusYieldManagerFacet(nymAddr).swapInPrivileged(STABLE_TOKEN_ADDRESS,
         ↪  address(this), amountA);
        require(IERC20(SAT_ADDRESS).balanceOf(address(this)) == amountB,
         ↪  "balance not match");

        IERC20(STABLE_TOKEN_ADDRESS).approve(strategyAddr, amountA);
        IERC20(SAT_ADDRESS).approve(strategyAddr, amountB);

        ...
```

Recommendation:

Use OpenZeppelin's `safeTransfer` and `forceApprove` if those tokens will be integrated.

Status:

Partially fixed in PR#5.

2.4.2 Unreachable receive() Function in NexusYieldManagerFacet

Severity: Informational

Context: NexusYieldManagerFacet.sol#L655-L655

Description:

The `receive()` function in `NexusYieldManagerFacet` is unreachable and redundant. This is because the contract is a facet of a Diamond proxy pattern, where all calls are delegated through the Diamond proxy contract which already implements its own `receive()` function.

Recommendation:

Remove the unreachable `receive()` function from `NexusYieldManagerFacet`.

Status:

Fixed in PR#5.

2.4.3 Incorrect Borrowing Fee Event Parameters in BorrowerOperationsFacet

Severity: Informational

Context: BorrowerOperationsFacet.sol#L372-L372

Description:

In `BorrowerOperationsFacet`'s `_triggerBorrowingFee` function, the wrong address is being used for the borrowing fee event during trove adjustments. The function passes `msg.sender` instead of the actual borrower's address (`account`) when calling the function, which leads to incorrect event emission. When the `BorrowingFeePaid` event is emitted, it will show the contract caller's address instead of the actual borrower's address.

```
        if (_isDebtIncrease) {
            require(_debtChange != 0, "BorrowerOps: Debt increase requires
             ↪  non-zero debtChange");

             ↪  BorrowerOperationsLib._requireValidMaxFeePercentage(_maxFeePercentage);

            vars.netDebtChange +=
                _triggerBorrowingFee(s, troveManager, collateralToken,
                 ↪  msg.sender, _maxFeePercentage, _debtChange);
        }
```

Recommendation:

Use `account` instead of `msg.sender`.

Status:

Fixed in PR#5.

2.4.4 Incorrect Error Message in `DebtToken` Transfer Validation

Severity: Informational

Context: DebtToken.sol#L234-L237

Description:

The error message in `DebtToken`'s `_requireValidRecipient` function is inaccurate. It mentions "StabilityPool" and "BorrowerOps" in the error message even though the check only validates against `TroveManager` addresses.

```
    function _requireValidRecipient(address _recipient) internal view {
        require(
            _recipient != address(0) && _recipient != address(this),
            "Debt: Cannot transfer tokens directly to the Debt token contract
             ↪  or the zero address"
        );
        require(
            !troveManager[ITroveManager(_recipient)],
            "Debt: Cannot transfer tokens directly to the StabilityPool,
             ↪  TroveManager or BorrowerOps"
        );
    }
```

Recommendation:

Update the error message to accurately reflect what is being checked.

Status:

Fixed in PR#5.

2.4.5 Unused vault Variable in transferCollToPrivilegedVault function

Severity: Informational

Context: TroveManager.sol#L1359-L1371

Description:

In the `transferCollToPrivilegedVault` function of `TroveManager`, there's a mismatch between the intended recipient and the code logic. While the function takes a `vault` parameter and emits an event indicating the transfer is to this vault, the actual `transfer` call sends the collateral to the `vaultManager` address instead.

```
function transferCollToPrivilegedVault(address vault, uint256 amount)
 ↪  external onlyOwner {
    // check the output amount does not exceed the limit
    require(
        collateralOutput + amount
            <= getEntireSystemColl() * (FARMING_PRECISION -
             ↪  farmingParams.retainPercentage) / FARMING_PRECISION,
        "TroveManager: Exceed the collateral transfer limit"
    );

    // record the collateral output
    collateralOutput += amount;
    collateralToken.transfer(address(vaultManager), amount);
    emit CollateralTransferred(vault, amount);
}
```

Recommendation:

Update the function and event to reflect that transfers go through the `VaultManager`.

Status:

Fixed in PR#5.

2.4.6 Function Name Typo in RewardManager

Severity: Informational

Context: RewardManager.sol#L379-L379

Description:

The function `_isVaildCaller` in `RewardManager` contains a spelling error in its name. The word "Valid" is misspelled as "Vaild", which could cause confusion when reading or searching through the codebase.

Recommendation:

Rename the function to `_isValidCaller` to fix the spelling error and maintain code clarity.

Status:

Fixed in PR#5.

2.4.7 Incorrect Interface Type in VaultManager's Priority Array

Severity: Informational

Context: VaultManager.sol#L24-L24

Description:

The `priority` array in `VaultManager` uses the `INYMVault` interface, but it should be using `ICDPVault` instead. This mismatch in interface types could lead to missing function definitions or incorrect assumptions about available methods when interacting with vault contracts in future upgrades.

```
    // priority / rule
    INYMVault[] public priority;
```

Recommendation:

Update the priority array declaration to use the correct interface.

Status:

Fixed in PR#5.

2.4.8 Unnecessary Deadline Extension in UniswapV2Vault

Severity: Informational

Context: uniswapV2Vault.sol#L34-L36, uniswapV2Vault.sol#L43-L45

Description:

In both `executeStrategy` and `exitStrategy` functions, the timestamp deadline is set to `block.timestamp + 100` for Uniswap operations. This extension is unnecessary since all operations (swapping and liquidity provision/removal) occur within the same transaction.

```
        IUniswapV2Router01(strategyAddr).addLiquidity(
            STABLE_TOKEN_ADDRESS, SAT_ADDRESS, amountA, amountB, minA, minB,
             ↪  address(this), block.timestamp + 100
        );
```

Recommendation:

Simply use `block.timestamp` as the deadline to avoid confusion.

Status:

The contract is removed.

2.4.9 Inefficient Iteration in Reward Claiming Process in SatUSDVault

Severity: Informational

Context: uniswapV2Vault.sol#L34-L36, uniswapV2Vault.sol#L43-L45

Description:

The reward claiming process in `SatUSDVault` contains a nested iteration inefficiency. The `claimReward` function iterates through the `emissionList` once, and for each token, it calls `_claimReward` which triggers `_updateIntegrals`. The `_updateIntegrals` function then iterates through the `emissionList` again for each token being claimed. This creates an O(n^2) complexity where n is the length of `emissionList`.

```
    function claimReward() external whenNotPaused {
        for (uint256 i; i < emissionList.length; i++) {
            address token = emissionList[i];
            if (!isClaimStart(token) || token == address(UNDERLYING)) continue;
            uint256 amount = _claimReward(token, msg.sender);
            if (amount != 0) {
                rewardVault.transferAllocatedTokens(token, msg.sender, amount);
            }
            emit RewardClaimed(token, msg.sender, amount);
        }
    }

    function _updateIntegrals(address account, uint256 balance, uint256 supply)
     ↪  internal {
        for (uint256 i = 0; i < emissionList.length; i++) {
            uint256 integral = _updateRewardIntegral(emissionList[i], supply);
            _updateIntegralForAccount(emissionList[i], account, balance,
             ↪  integral);
        }
        lastUpdate = uint32(block.timestamp);
    }
```

Recommendation:

Consider restructuring the reward calculation logic to avoid the nested iteration. One approach would be to update all integrals in a single pass through the emission list before processing individual token claims.

Status:

Fixed.