hashlock.

Security Audit

Rubicon (Digital Asset)

October 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

The LSA team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure. All issues identified during the initial review were fully remediated prior to publication. The final rating is Secure.

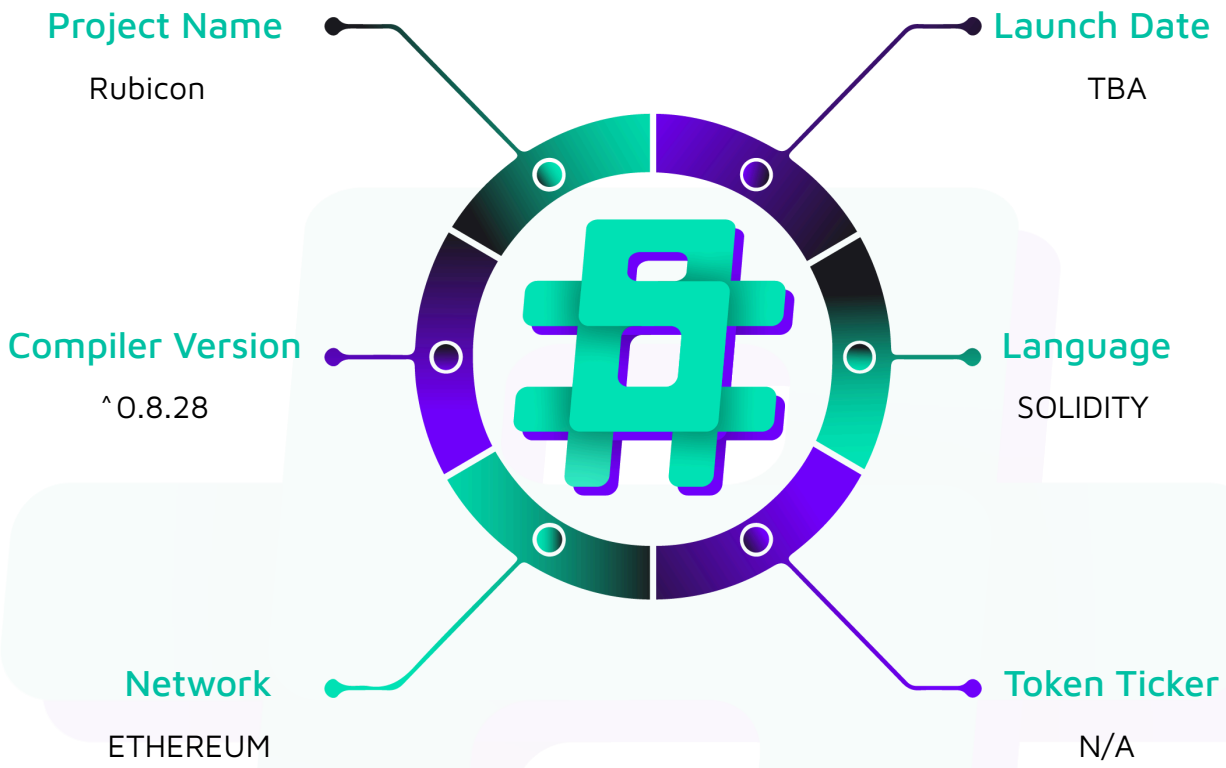
Project Context

Project Rubicon is a novel cross-chain staking infrastructure purpose-built for the Bittensor ecosystem. It enables users to stake subnet-native tokens (Alpha) to earn validator rewards while simultaneously unlocking liquidity via xAlpha tokens (representing staked Alpha).
This project is spearheaded by General Tao Ventures in conjunction with Chainlink, who intend for the Rubicon to become the canonical staking solution for the TAO ecosystem.

Project Name: Rubicon
Project Type: Digital Asset, RWA, Security Tokens, RWAfi, TradFi, DeFi
Website: https://www.rubiconbridge.io/
Compiler Version: ^0.8.28
Logo:

PROJECT
RUBICON

Visualised Context:



Audit Scope

We at Hashlock audited the solidity code within the Rubicon project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Rubicon Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**October, 2025**
Contract 1	Blake2b.sol
Contract 2	LiquidStakedV2.sol
Contract 3	MetaGraphInterface.sol
Contract 4	BalanceTransfer.sol
Contract 5	Constants.sol
Contract 6	StakingV2Interface.sol
Contract 7	LiquidStakedV3.sol
Audited GitHub Commit Hash	ed25b1897976f9cf47382f8acaf8379354f73010
Fix Review GitHub Commit Hash	0a3ae61b6622b501df8062f7ae7089fbdd771472

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

3 High severity vulnerabilities

5 Medium severity vulnerabilities

2 Low severity vulnerability

10 QAs

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
Blake2b.sol - Provides BLAKE2b cryptographic hash function implementation for converting Ethereum addresses to SS58 public keys used in the Bittensor network.	**Contract achieves this functionality.**
LiquidStakedV2.sol - Enhanced version of LiquidStaked with emergency withdrawal mechanisms, community governance features, and improved security controls, including timelock protections. Allows anyone to: - Stake TAO: Convert TAO to LSA tokens via stakeWithSlippage() - Redeem LSA: Convert LSA tokens back to TAO via redeemWithSlippage() - Collect Yield: Trigger yield distribution via collectYield() - View Functions: Query exchange rates, TVL, fees, and other contract information - Update TVL: Manually refresh TVL data via updateTVL() Allows the owner to: - Fee Management: Set treasury fees, yield fees, and network fees - Treasury Configuration: Set treasury EVM address for fee collection	**Contract achieves this functionality.**

- Emergency Controls: Pause/unpause contract operations - Emergency Withdrawals: Initiate and execute emergency alpha/TAO withdrawals with 48 48-hour timelock - Community Representative Management: Initiate and execute community representative changes with a timelock	
LiquidStakedV3.sol Extends LiquidStakedV2 to introduce architectural changes: - Upgradeable contract using UUPS proxy pattern - Fee mechanism redesign: Fees accumulated as alpha (not minted as LSA) - Role-based fee claiming - No on-chain Blake2b hashing	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Rubicon project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Rubicon project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] LiquidStakedV2#stakeWithSlippage - User loses leftover wei from msg.value conversion

Description

The contract function that accepts native value payments uses division to convert `msg.value` from wei to a 9 decimal format, causing users to lose any remainder wei that doesn't divide evenly by `1e9`.

Vulnerability Details

In `stakeWithSlippage()`, the code performs `uint256 amountIn9Decimals = msg.value / 1e9;` to convert the incoming `msg.value` from wei to a 9 decimal format. This integer division operation discards any remainder wei that doesn't divide evenly by `1e9`.

For example, if a user sends `1,500,000,000` wei, the division results in `amountIn9Decimals = 1`, and the remaining `500,000,000` wei is permanently lost to the contract.

Impact

Users permanently lose any remainder wei when the `msg.value` doesn't divide evenly by `1e9`, resulting in direct loss of funds.

Recommendation

Consider adding a check to ensure `msg.value` is divisible by `1e9`, `require(msg.value % 1e9 == 0, "Amount must be divisible by 1e9")` or return the remainder to the user.

Status

Resolved

[H-02] LiquidStakedV2#stakeWithSlippage, LiquidStakedV2#redeemWithSlippage - Yield fee not collected before operations that affect yield calculation variables

Description

The `_calculateAndCollectYield()` function is not called before stake and redeem operations, causing accumulated yield fees to be lost when `_updateRecordedAlphaBalance()` resets the `lastRecordedAlphaBalance` variable used for yield calculation.

Vulnerability Details

The yield collection mechanism relies on comparing the current alpha balance `_getTotalContractStake()` with the last recorded balance `lastRecordedAlphaBalance` to calculate accumulated yield and yield fees. However, the stake and redeem functions call `_updateRecordedAlphaBalance()` at the end of their operations, which updates `lastRecordedAlphaBalance` to the current balance.

When these functions execute, any accumulated yield fees since the last collection are permanently lost because:

1. `_calculateAndCollectYield()` is not called before the operation
2. `_updateRecordedAlphaBalance()` resets `lastRecordedAlphaBalance` to the current balance
3. Future yield calculations will use the new balance as a baseline, losing the previous yield

Impact

Accumulated yield fees are permanently lost when users stake or redeem, resulting in direct loss of funds for the treasury.

Recommendation

Call `_calculateAndCollectYield()` at the beginning of all stake and redeem functions, before any operations that could affect the alpha balance or exchange rate. This

ensures the accumulated yield is properly collected and distributed before the balance tracking variables are updated.

Status

Resolved

[H-03] LiquidStakedV2#executeEmergencyWithdrawAlpha - Emergency withdrawal can cause unfair losses and withdrawal failure due to time gap and supply imbalance

Description

The `initiateEmergencyWithdrawAlpha` and `executeEmergencyWithdrawAlpha` functions allow the owner to withdraw Alpha tokens from the contract after a timelock. However, this process introduces an exploitable time gap during which users can anticipate the impending Alpha withdrawal, exit early, and offload losses onto remaining participants. Furthermore, if users withdraw before the timelock expires, the available Alpha balance may become insufficient, causing the emergency withdrawal execution to fail.

Vulnerability Details

The vulnerability arises from the two-step withdrawal flow combined with the public EmergencyWithdrawalInitiated event. When the owner calls `initiateEmergencyWithdrawAlpha`, the contract records the withdrawal amount and sets `emergencyWithdrawPending = true`. This action emits an event that publicly signals the future withdrawal of Alpha from the staking pool. Users monitoring the blockchain can detect this event and immediately redeem their LSA tokens or withdraw liquidity before the timelock expires, preserving their value while diluting the remaining users' share. By the time `executeEmergencyWithdrawAlpha` is called, the total Alpha stake (`_getTotalContractStake`) may have changed significantly, either lowering available Alpha or invalidating the stored `emergencyWithdrawAmount`. In this case, the transaction will revert, preventing the emergency withdrawal from succeeding.

```
function executeEmergencyWithdrawAlpha() external onlyOwner {
    require(emergencyWithdrawPending, "No emergency withdrawal pending");
    require(
        block.timestamp >= emergencyWithdrawInitiated + EMERGENCY_TIMELOCK,
        "Timelock period not yet passed" );
    uint256 withdrawAmount = emergencyWithdrawAmount;
    uint256 totalStaked = _getTotalContractStake();
    require(totalStaked > 0, "No alpha staked to withdraw");
    require(withdrawAmount <= totalStaked, "Cannot withdraw more than currently staked");
    emergencyWithdrawPending = false;
```

```
    emergencyWithdrawInitiated = 0;
    emergencyWithdrawAmount = 0;
    _removeStake(withdrawAmount);
    _updateRecordedAlphaBalance();}
```

Because `_removeStake` depends on the contract's available Alpha, any user-triggered exits between initiation and execution can cause `_removeStake` to revert, making the emergency process unreliable.

Impact

The issue enables a predictable and unfair redistribution of value among users: those who monitor the EmergencyWithdrawalInitiated event can exit early and avoid losses, while others who remain in the pool bear a disproportionate reduction in LSA token value. In severe cases, early withdrawals can reduce the Alpha balance below the scheduled emergency withdrawal amount, causing executeEmergencyWithdrawAlpha to revert entirely and locking the contract in a semi-broken state. This creates both economic damage (value leakage and inequality) and operational risk (failed emergency recovery).

Recommendation

The most robust fix is to remove the open timelock mechanism and replace it with a direct, governance-controlled emergency withdrawal process. If a timelock must remain, the withdrawal amount should be recalculated at execution based on the latest `_getTotalContractStake()` instead of a stored value.

Additionally, these emergency withdrawal functions should be restricted with the whenPaused modifier to ensure they can only be executed when the protocol is paused. This enforces the intended emergency-only behavior and prevents staking or redemption activity from affecting accounting between the initiation and execution phases.

Status

Resolved

Medium

[M-01] LiquidStakedV2#redeemWithSlippage - Functions do not burn the full lsaAmount creating dust amounts and inconsistent behavior

Description

The `redeemWithSlippage()` function does not burn the complete `lsaAmount` parameter as expected, leaving dust amounts of LSA tokens in the user's balance. This creates inconsistent behavior where users cannot fully redeem their intended amount and are left with small, potentially unredeemable dust amounts.

Vulnerability Details

The `redeemWithSlippage()` function calculates the amount to burn as follows:

```
// Initial burn calculation
uint256 redeemLsaAmount = lsaAmount - feeLsaAmount - ((lsaAmount * networkFee) / FEE_SCALED);
_burn(msg.sender, feeLsaAmount + redeemLsaAmount);

// Additional burn logic (limited by networkFeeAmount)
if (lsaEquivalent > redeemLsaAmount) {
    uint256 additionalBurn = lsaEquivalent - redeemLsaAmount;
    uint256 networkFeeAmount = (lsaAmount * networkFee) / FEE_SCALED;
    if (additionalBurn > networkFeeAmount)
        additionalBurn = networkFeeAmount; // This limits the additional burn
    if (additionalBurn > 0) {
        _burn(msg.sender, additionalBurn);
        totalBurned += additionalBurn;
    }
```

```
}
```

In case the `redeemLsaAmount` is the same as the `lsaEquivalent` or the difference is less than the `networkFeeAmount`, the additional burn logic is not executed. This leaves dust amounts in the user's balance that may be too small to redeem later due to minimum amount requirements or gas costs, and is inconsistent with the user's expectations given by the interface.

The additional burn logic may be unnecessary because:

1. Network Fee Buffer: The protocol already reduces the `redeemLsaAmount` by charging a network fee (typically 0.06%), providing a buffer against small roundings or minor discrepancies
2. Deterministic Precompile: The Subtensor `removeStake` precompile should remove exactly the amount requested, not more.

Impact

This creates poor user experience and economic inefficiency as users are left with dust amounts of LSA tokens that may be uneconomical to redeem and is inconsistent with the interface where `lsaAmount` should represent the exact amount to be redeemed.

Recommendation

Consider two potential approaches to fix this issue.

Option 1: Simplified Approach (recommended), burn the complete `lsaAmount` and remove the additional burn logic entirely, and trust the network fee buffer:

```
// Calculate amounts and execute burn

uint256 feeLsaAmount = (fee > 0 && treasuryEvm != address(0)) ? (lsaAmount * fee) / FEE_SCALED : 0;

uint256 netLsaAmount = lsaAmount - feeLsaAmount;

uint256 networkFeeAmount = (netLsaAmount * networkFee) / FEE_SCALED;

uint256 redeemLsaAmount = netLsaAmount - networkFeeAmount;

// Burn the full lsaAmount as expected by users
```

```
_burn(msg.sender, lsaAmount);

// Remove stake using the calculated amount

_removeStake(((redeemLsaAmount * rate) / 1e18) / 1e9);
```

Option 2: Full Burn with TAO Adjustment, burn the complete `lsaAmount` and handle any excess alpha redemption by adjusting the TAO transfer:

```
// Always burn the full lsaAmount

_burn(msg.sender, lsaAmount);

// Calculate expected alpha to remove based on lsaAmount

uint256 expectedAlphaToRemove = ((lsaAmount * rate) / 1e18) / 1e9;

uint256 actualAlphaRemoved = beforeStake - _getTotalContractStake();

// If more alpha was removed than expected, adjust the TAO transfer

uint256 taoReceived = address(this).balance - beforeBalance;

if (actualAlphaRemoved > expectedAlphaToRemove) {

    // Calculate excess alpha value in TAO and reduce the transfer

    uint256 excessAlpha = actualAlphaRemoved - expectedAlphaToRemove;

    uint256 excessTaoValue = _calculateTaoFromAlpha(excessAlpha);

    taoReceived = taoReceived - excessTaoValue;

}

// Send adjusted TAO amount to user

(bool sent, ) = msg.sender.call{value: taoReceived}("");

require(sent, "Transfer Failed");
```

However, this second option would also require handling the remaining tao not returned to the user.

Status

Resolved

[M-02] LiquidStakedV2 - Incorrect pragma version

Description

The contract uses a floating pragma `^0.8.28`, which can cause compatibility issues with Bittensor EVM precompiles. According to the Bittensor EVM documentation, the Solidity version should be fixed to `0.8.24` or below to avoid precompile-related errors.

Vulnerability Details

The contract currently uses `pragma solidity ^0.8.28`

According to the Bittensor EVM troubleshooting documentation, this can cause issues `InvalidCode(Opcode(94)) when calling a precompile from another contract`.

And recommends using `cancun` as the EVM version and `0.8.24` as the Solidity version.

The floating pragma `^0.8.28` allows the compiler to use any version from `0.8.28`, which includes versions that are incompatible with Bittensor EVM precompiles.

Impact

Using a floating pragma with versions above `0.8.24` can cause precompile calls to fail.

Recommendation

Fix the pragma to use a specific version that is compatible with Bittensor EVM `pragma solidity 0.8.24`, and ensure the build system is configured to use `cancun` as evm version.

Status

Resolved

[M-03] LiquidStakedV2#setYieldFee - Yield fee change applies retroactively

Description

The `setYieldFee()` functions change the yield fee rate without first calling `_calculateAndCollectYield()` to collect accumulated yield using the current fee rate, causing the new fee rate to be applied retroactively to past yield.

Vulnerability Details

The `setYieldFee()` function immediately updates the `yieldFee` variable without collecting any accumulated yield that was earned under the previous fee rate. This creates a retroactive application problem where:

1. Yield accumulates over time under the current fee rate (e.g., 10%)
2. Owner calls `setYieldFee()` to change to a new rate (e.g., 20%)
3. The new fee rate is immediately applied to all accumulated yield
4. When `_calculateAndCollectYield()` is eventually called, it uses the new fee rate on the yield that was earned under the old rate

Impact

Yield fee changes are applied retroactively to past yield, resulting in incorrect fee collection

Recommendation

Call `_calculateAndCollectYield()` at the beginning of `setYieldFee()` function before updating the `yieldFee` variable. This ensures accumulated yield is collected using the current fee rate before applying the new rate to future yield.

Status

Resolved

[M-04] LiquidStakedV2 - Fee estimation functions ignore treasuryEvm presence, causing inaccurate estimates and UX / slippage discrepancies

Description

Multiple view estimation functions (e.g., `estimatedMintingFee`, `estimatedLsaToMint`, `estimatedMintingFeeInTao`, `estimatedTaoFromLsa`, `estimatedRedemptionFees`, `estimatedRedemptionFeesInTao`) calculate and return treasury fee values based solely on `fee > 0` without checking whether `treasuryEvm` is set. In practice, the protocol only charges/mints treasury fees when `treasuryEvm != address(0)`. This mismatch means on-chain read-only estimates can report non-zero fees even when no fee will be applied at execution, leading to wrong UX, slippage protection failures, and user confusion.

Vulnerability Details

Root cause: inconsistent guard logic between mutative flows and read-only estimators. Mutative functions (e.g., `stake`, `_calculateAndCollectYield`, `redeem`) only mint treasury fees when both `fee > 0 && treasuryEvm != address(0)`. Many view functions, however, compute and return fees whenever `fee > 0` and skip the `treasuryEvm` check:

```
function estimatedMintingFee(uint256 amount) external view returns (uint256) {
    // ...
    if (fee > 0) {
        uint256 feeLsaAmount = (totalLstAmount * fee) / FEE_SCALED;
        return feeLsaAmount; // Returns fee even if treasuryEvm == address(0)
    }
    return 0;
}
```

Because `treasuryEvm == address(0)` prevents actual fee minting in state-changing functions, the view-returned feeLsaAmount may be non-zero while the real transaction will not mint or collect any fee. This inconsistency can cause frontends to show incorrect expected outputs, cause stakeWithSlippage or redeemWithSlippage checks to fail or be overly conservative, and mislead users on cost estimates.

Impact

View functions returning non-zero treasury fee estimates while the implementation will not mint fees (because `treasuryEvm` is unset) creates diverging on-chain expectations versus actual behavior; this can cause front-end UI to display higher fees than those actually charged, break slippage protection (users may set `minLsaAmount` or `minTaoAmount` based on inflated estimates and transactions may revert or be rejected), undermine user trust, and lead to incorrect off-chain analytics or accounting that assume fees were collected when they were not, increasing risk of failed UX and erroneous decision-making by users and integrators.

Recommendation

Ensure all estimator/view functions apply the same guard used in state-changing flows: require `treasuryEvm != address(0)` in addition to `fee > 0` when computing/presenting treasury fees. Minimal patch example — apply the treasuryEvm check to each estimator:

```
-    if (fee > 0) {
+    if (fee > 0 && treasuryEvm != address(0)) {
         uint256 feeLsaAmount = (totalLstAmount * fee) / FEE_SCALED;
         return feeLsaAmount; // Return actual fee amount charged
     }
```

Apply the same pattern to `estimatedLsaToMint`, `estimatedMintingFeeInTao`, `estimatedTaoFromLsa`, `estimatedRedemptionFees`, and `estimatedRedemptionFeesInTao`.

Status

Resolved

[M-05] LiquidStakedV2#estimatedRedemptionFeesInTao - Incorrect network fee Application in redemption estimations

Description

The `estimatedRedemptionFeesInTao` view function applies the `networkFee` on the original `lsaAmount` instead of the net LSA after the treasury fee. This causes the estimated redemption amounts to understate the actual TAO users receive in the redeem function. The issue occurs whenever both `fee > 0 and networkFee > 0`.

The LiquidStakedV2::estimatedTaoFromLsa function has this patched.

Vulnerability Details

The inconsistency arises because the actual redeem function calculates the network fee on the net LSA amount (after treasury fee):

```
uint256 feeLsaAmount = 0;
if (fee > 0 && treasuryEvm != address(0)) {
    feeLsaAmount = (lsaAmount * fee) / FEE_SCALED;
}


uint256 redeemLsaAmount = lsaAmount -
    feeLsaAmount -
    ((lsaAmount - feeLsaAmount) * networkFee) / FEE_SCALED;
```

This mismatch results in underestimated redemption values and fee buffers, misleading users and external strategies relying on accurate estimations.

Impact

Users and integrators relying on estimation functions to calculate redemption outcomes will receive inaccurate, underestimated TAO values, potentially causing suboptimal strategy execution, poor liquidity decisions, or reduced trust in the protocol's fee transparency. While funds are not directly at risk, operational inefficiencies and strategic miscalculations can arise, especially in automated DeFi interactions.

Recommendation

Update estimation functions to apply the network fee on the net LSA amount after the treasury fee, aligning with the redemption logic:

```
- uint256 networkFeeBufferLsa = (lsaAmount * networkFee) / FEE_SCALED;
+ uint256 networkFeeBufferLsa = (netLsaAmount * networkFee) / FEE_SCALED;
```

Status

Resolved

[M-06] LiquidStakedV3#grantFeeClaimerRole/revokeFeeClaimerRole - Owner and default admin role conflict

Description

The contract uses both `OwnableUpgradeable` and `AccessControlUpgradeable`, but the fee claimer role management functions use the `onlyOwner` modifier while calling `grantRole()` and `revokeRole()` which require `DEFAULT_ADMIN_ROLE`. This creates a conflict where these functions will fail if the owner and admin are different addresses.

Vulnerability Details

The `LiquidStakedV3` contract inherits from both `OwnableUpgradeable` and `AccessControlUpgradeable`. During initialization, the deployer is set as both the owner and granted the `DEFAULT_ADMIN_ROLE`.

The functions `grantFeeClaimerRole()` and `revokeFeeClaimerRole()` both use the `onlyOwner` modifier to restrict access, but then call `grantRole()` and `revokeRole()` public functions, respectively. These `AccessControl` functions require that the caller has the `DEFAULT_ADMIN_ROLE` (or the admin role for that specific role) to grant or revoke roles.

```
// LiquidStakedV3
function grantFeeClaimerRole(address account) external onlyOwner {
    if (account == address(0)) revert InvalidAddress();
    grantRole(FEE_CLAIMER_ROLE, account);
    emit FeeClaimerRoleGranted(account, msg.sender, block.timestamp);
}


// LiquidStakedV3
function revokeFeeClaimerRole(address account) external onlyOwner {
    revokeRole(FEE_CLAIMER_ROLE, account);
    emit FeeClaimerRoleRevoked(account, msg.sender, block.timestamp);
}


// AccessControlUpgradeable
function grantRole(bytes32 role, address account) public virtual
onlyRole(getRoleAdmin(role)) {
```

```
    _grantRole(role, account);
}


// AccessControlUpgradeable
function revokeRole(bytes32 role, address account) public virtual
onlyRole(getRoleAdmin(role)) {
    _revokeRole(role, account);
}
```

The deployment script `deploy-liquidstakedv3-upgradeable.js` includes functionality to transfer ownership to a new address. However, the script only calls `transferOwnership()` and does not transfer the `DEFAULT_ADMIN_ROLE`. This means that after deployment, ownership will likely be transferred to a separate address, while the `DEFAULT_ADMIN_ROLE` remains with the deployer's private key.

This creates a problematic scenario where:

1. Ownership is transferred to a new address via `transferOwnership()` (from Ownable)
2. The `DEFAULT_ADMIN_ROLE` remains with the original deployer's private key
3. The new owner calls `grantFeeClaimerRole()` or `revokeFeeClaimerRole()` but the function fails because the owner lacks `DEFAULT_ADMIN_ROLE`
4. Meanwhile, the deployer key (which still holds `DEFAULT_ADMIN_ROLE`) can directly call `grantRole()` and `revokeRole()` to add or remove `FEE_CLAIMER_ROLE` members independently of the owner

Impact

If ownership is transferred while the `DEFAULT_ADMIN_ROLE` is not, the fee claimer role management functions will fail for the owner, breaking expected contract functionality. Additionally, the `DEFAULT_ADMIN_ROLE` retains the ability to directly manage `FEE_CLAIMER_ROLE` members, creating a parallel access control system that bypasses the intended owner-controlled role management.

Recommendation

Consider using `AccessControlUpgradeable` exclusively for all access control needs. This will require removing the redundant wrapper functions `grantFeeClaimerRole()`, `revokeFeeClaimerRole()`, and the view functions associated with it, removing `OwnableUpgradeable` and replacing all the `onlyOwner` modifiers with `onlyRole(DEFAULT_ADMIN_ROLE)`.

Alternatively, consider keeping both `OwnableUpgradeable` and `AccessControlUpgradeable` and removing the redundant wrapper functions `grantFeeClaimerRole()`, `revokeFeeClaimerRole()`, and the view functions associated with it, and use AccessControlUpgradeable's standard `grantRole()` and `revokeRole()` functions directly to manage the `FEE_CLAIMER_ROLE`, calling them from the `DEFAULT_ADMIN_ROLE` role. Also, make sure to include functionality in the deployment script to transfer the `DEFAULT_ADMIN_ROLE` to a secure new owner address.

Status

Resolved

Low

[L-01] LiquidStakedv2#_calculateAndCollectYield - Missing Update for accumulatedYieldFees

Description

The variable `accumulatedYieldFees` is defined to track the total yield fees collected across all yield processing cycles. However, the contract never updates this variable. As a result, the protocol state does not correctly reflect the actual accumulated fees, which may lead to incorrect accounting, off-chain reporting errors, and future logic inconsistencies in any functions depending on this variable.

Vulnerability Details

In LiquidStakedV2, the internal function `_calculateAndCollectYield` is responsible for calculating and distributing yield between the treasury and holders. During this process, the function computes `treasuryYieldAlpha` — the fee portion that represents yield collected by the treasury. However, the function never increments `accumulatedYieldFees` by this amount. This omission prevents the protocol from accurately tracking total collected yield fees.

```
uint256 treasuryYieldAlpha = (yieldGenerated * yieldFee) / FEE_SCALED;


// Convert only the treasury portion to LSA
if (treasuryYieldAlpha > 0 && treasuryEvm != address(0)) {
    uint256 rateBefore = _exchangeRate();
    uint256 totalSupplyBefore = totalSupply();
    uint256 treasuryYieldIn18Decimals = treasuryYieldAlpha * 1e9;
    uint256 treasuryLsaAmount = (treasuryYieldIn18Decimals * 1e18) / rateBefore;


    if (treasuryLsaAmount > 0) {
        _mint(treasuryEvm, treasuryLsaAmount);
        // Missing accumulation of yield fees here
        // accumulatedYieldFees += treasuryYieldAlpha;
```

```
    }
}
```

Impact

This issue causes incorrect accounting of collected yield fees across the protocol. Although funds are correctly distributed to the treasury, the `accumulatedYieldFees` state variable — which is expected to represent the cumulative fee total — remains outdated or zero. This can lead to misleading on-chain data for analytics, faulty assumptions in future upgrades, and inconsistencies if the variable is later used to enforce limits, perform audits, or trigger treasury withdrawals.

Recommendation

If on-chain tracking of yield fees is desired, the `accumulatedYieldFees` variable should be updated whenever yield is distributed, as shown below:

```
   if (treasuryLsaAmount > 0) {
        _mint(treasuryEvm, treasuryLsaAmount);
+       accumulatedYieldFees += treasuryYieldAlpha;
```

Otherwise, if the project intends to rely solely on off-chain tracking through the YieldDistributed event, the accumulatedYieldFees variable can be safely removed to avoid confusion and maintain cleaner code.

Status

Resolved

[L-02] LiquidStakedV2#redeemAsAlphaWithSlippage - Redundant Signature Validation Tied to msg.sender

Description

The function verifies an EIP-712 signature where the signer is `msg.sender` and then performs all actions for that same `msg.sender` (balance check, burn, nonce update). Because the caller must be the user, this provides no delegation/meta-transaction capability and adds unnecessary gas and complexity.

Recommendation

Choose one:

- Option A (simplify): Remove v, r, s, and the verification block, relying on `msg.sender` for direct execution.
- Option B (support delegation): Add address `fromUser`, verify a signature over {fromUser, to, nonce, deadline}, use nonces[fromUser], and burn from fromUser (with allowance or EIP-3009-style auth), enabling relayer execution.

Status

Resolved

[L-03] LiquidStakedV3 - Centralization risk

Description

Multiple critical functions are restricted to the contract owner, creating centralization risks where the owner has significant control over the protocol.

Vulnerability Details

The LiquidStakedV3 contracts contain several functions that are restricted to the owner only, including pausing the contract, updating parameters, emergency withdrawal and upgrading the contract implementation.

Impact

A compromised or malicious owner can cause denial of service, loss of funds, or manipulation of the protocol's core functionality by upgrading the contract implementation to a malicious implementation.

Recommendation

Consider using a multisig wallet for the address controlling the owner role. This would require multiple approvals for critical actions.

Status

Resolved

QA

[Q-01] LiquidStakedV2 - Unused code

Description

The `LiquidStakedV2` contract contains several unused functions and variables that should be removed to enhance code maintainability.

Vulnerability Details

The following unused code has been identified in the contracts.

In `LiquidStakedV2`:

- `delegateRemoveStake()` function
- `DOMAIN_TYPEHASH` constant
- `REDEMPTION_TYPEHASH` constant
- `DECIMAL_SCALE_FACTOR` variable
- `nonces` mapping
- `balanceTransfer` variable

Impact

Unused code makes the codebase harder to maintain.

Recommendation

Remove all unused functions and variables not used in the contract logic.

Status

Resolved

[Q-02] LiquidStakedV2 - Magic numbers should be replaced with named constants

Description

The contracts use magic numbers like `1e9`, `1e18`, and `1e27` throughout the codebase instead of defining them as named constants.

Vulnerability Details

The contracts use magic numbers in multiple places for decimal conversions and calculations:

- `1e9` - Used for converting between 9 decimal (rao) and 18 decimal (wei) formats
- `1e18` - Used for 18-decimal precision calculations and conversions
- `1e27` - Used for complex decimal conversions in exchange rate calculations

Impact

Magic numbers make the code harder to understand, audit, and maintain, increasing the likelihood of bugs and making future modifications more error-prone.

Recommendation

Define named constants at the contract level and replace all magic numbers.

Status

Resolved

[Q-03] LiquidStakedV2 - Variables set in the constructor should be declared as immutable

Description

The `NETUID`, `VALIDATOR_UID`, `blake2bInstance`, and `contractSS58Pub` variables in both contracts are only assigned in the constructor and never modified afterwards, but they are declared as regular state variables instead of `immutable`. This results in unnecessary gas costs and doesn't make their immutability explicit.

Vulnerability Details

The `LiquidStakedV2` contract declares these variables as regular state variables:

```
uint256 public NETUID;

uint256 public VALIDATOR_UID;

BLAKE2b private blake2bInstance;

bytes32 public contractSS58Pub;
```

These variables are only assigned in the constructor and never modified afterwards. They are used frequently throughout the contracts for precompile calls and validator operations. Regular state variables require expensive storage reads (SLOAD operations), while immutable variables are inlined at compile time, eliminating storage reads and reducing gas costs.

Impact

It results in unnecessary gas costs for contract users.

Recommendation

Declare these variables as immutable to improve gas efficiency and make their immutability explicit.

Status

Resolved

[Q-04] LiquidStakedV2 - FEE_SCALED should be declared as constant

Description

The `FEE_SCALED` variable is hardcoded to `10000` and never changes, but it is declared as a regular state variable instead of a constant. This results in unnecessary gas costs since it's used extensively throughout the contracts for fee calculations.

Vulnerability Details

The contract declares `FEE_SCALED` as a regular state variable:

```
uint256 public FEE_SCALED = 10000; // Basis points: 10000 = 100%, 100 = 1%, 1 = 0.01%
```

This variable is never modified and is used and named using the convention of capital letters. Making it clear that the intention is to be a constant.

Impact

It results in unnecessary gas costs for contract users since `FEE_SCALED` is accessed frequently in fee calculations throughout the contracts.

Recommendation

Declare `FEE_SCALED` as a constant.

Status

Resolved

[Q-05] Constants - Unused file

Description

The `Constants.sol` file contains constant addresses that are not used by the contracts and creates confusion with the actual addresses defined in the interface files.

Vulnerability Details

The `Constants.sol` file defines constants that are never imported or used by the contracts. However, the contracts actually use constants defined in the interface files, which have different addresses. This creates confusion about which addresses are actually being used and makes the codebase harder to maintain.

Impact

The unused `Constants.sol` file creates confusion about which addresses are actually used by the contracts and makes the codebase harder to maintain and audit.

Recommendation

Remove the `Constants.sol` file.

Status

Resolved

[Q-06] LiquidStakedV2#_updateTVL - Missing metrics calculation

Description

The `_updateTVL()` function never assigns values to `currentBackingRatio` and `currentUtilizationRate`, unlike `LiquidStaked`. This results in these values always being `0` in the emitted `TVLUpdated` event.

Vulnerability Details

In `LiquidStakedV2`, the `_updateTVL()` function initializes `currentBackingRatio` and `currentUtilizationRate` to `0` but never assigns them values, unlike `LiquidStaked` which properly calculates these metrics through `getPortfolioMetrics()`.

Impact

The `TVLUpdated` event in `LiquidStakedV2` always emits `0` for `currentBackingRatio` and `currentUtilizationRate`, providing incomplete information to event listeners and making the events less useful for monitoring and analytics.

Recommendation

Add the missing calculations to assign `currentBackingRatio` and `currentUtilizationRate`, remove those variables from the event or document why they are not being calculated.

Status

Resolved

[Q-07] blake2b, LiquidStakedV2 - Missing NatSpec documentation in some functions

Description

The contracts `blake2b`, and `LiquidStakedV2` lack NatSpec documentation in some of the functions that explain their intended functionality. This makes the code harder to understand, maintain, and audit.

Vulnerability Details

All contracts have NatSpec documentation for them, but it is missing some functions explaining function purposes, parameters, return values, and implementation details.

Impact

Lack of documentation makes the code harder to understand, maintain, and integrate with, while also reducing audit efficiency.

Recommendation

Add comprehensive NatSpec documentation to all missing functions.

Status

Resolved

[Q-08] LiquidStakedV2#redeemAsAlphaWithSlippage - Incorrect Revert Error for Zero Transfer

Description

redeemAsAlphaWithSlippage reverts with `NoStake` when `actualAlpha == 0` after attempting `_transferStake`. This error name suggests a zero-stake input scenario rather than a failed or zero-result transfer during redemption, which can confuse monitoring/analytics and mislead operators.

Recommendation

Introduce a dedicated error for this condition, e.g., error `NoAlphaTransferred`; and use it in place of `NoStake` when `actualAlpha == 0`. Alternatively, reuse `TransferFailed` if the failure mode corresponds to the underlying transfer operation failing rather than rounding to zero.

Status

Resolved

[Q-09] LiquidStakedV2#verifySignature - Internal Function Should Use Underscore Prefix

Description

The internal helper `verifySignature` does not follow the common Solidity convention of prefixing internal/private helpers with an underscore. This creates inconsistency with typical code style expectations and reduces greppability of externally callable vs. internal-only functions.

Recommendation

Rename `verifySignature` to `_verifySignature` and update all call sites accordingly.

Status

Resolved

[Q-10] LiquidStakedV2#depositAlphaWithSlippage - Misaligned Indentation Reduces Readability

Description

The mid-block indentation in `depositAlphaWithSlippage` is inconsistent with surrounding code, making the control flow harder to visually parse. This is particularly evident around the conversion and rate-check logic, where statements are left-shifted relative to their enclosing scope. While not changing compiled behavior, inconsistent formatting increases maintenance risk and can obscure subtle logic changes. Reference: contracts/fixed.sol:428 and contracts/fixed.sol:463–468.

Recommendation

Adopt a consistent indentation standard (e.g., 4 spaces) and re-align the statements under their proper block scope. Enforce formatting via Prettier with the Solidity plugin, Foundry's fmt, or Solhint in CI to prevent regressions.

Status

Resolved

Centralisation

The protocol includes administrative functions limited to authorised governance addresses, balancing decentralisation with security oversight.

Conclusion

After Hashlock's analysis, the Rubicon project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code.This disclaimer is standard across all smart-contract audits and does not imply any additional risk specific to Rubicon. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au