#Hashlock.

Algem (dApp)

SMART CONTRACT

Security Audit

Performed on Contracts:

Sio2Adapter.sol
MD5 Hash: c35c00ddbbb1984b35373c9d2aea01ca
Sio2AdapterAssetManager.sol
MD5 Hash: 21e733a3995f93daaf68006e36425379
Github Commit Hash: bd12f0c74b95970f49d28484ade26a9bcdf3d3ef

Platform

ASTR

hashlock.com.au

AUGUST 2023

Table of Contents

Executive Summary

The Algem team partnered with Hashlock to conduct a security audit of their Sio2Adapter.sol smart contract. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

Algem is a DeFi dApp built on Astar Network that allows you to stay liquid while staking your ASTR. Staying liquid means you can double-dip with your Astar tokens by staking while yield farming.

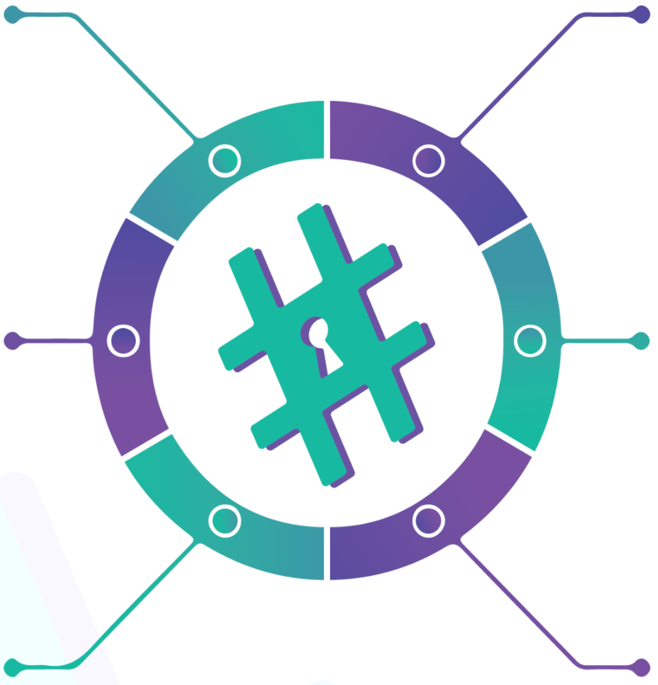
Simply put, you don't have to choose between staking and yield farming with your Astar tokens. You can do both.

Project Name: Algem
Compiler Version: ^0.8.4
Website: https://www.algem.io/

Logo:

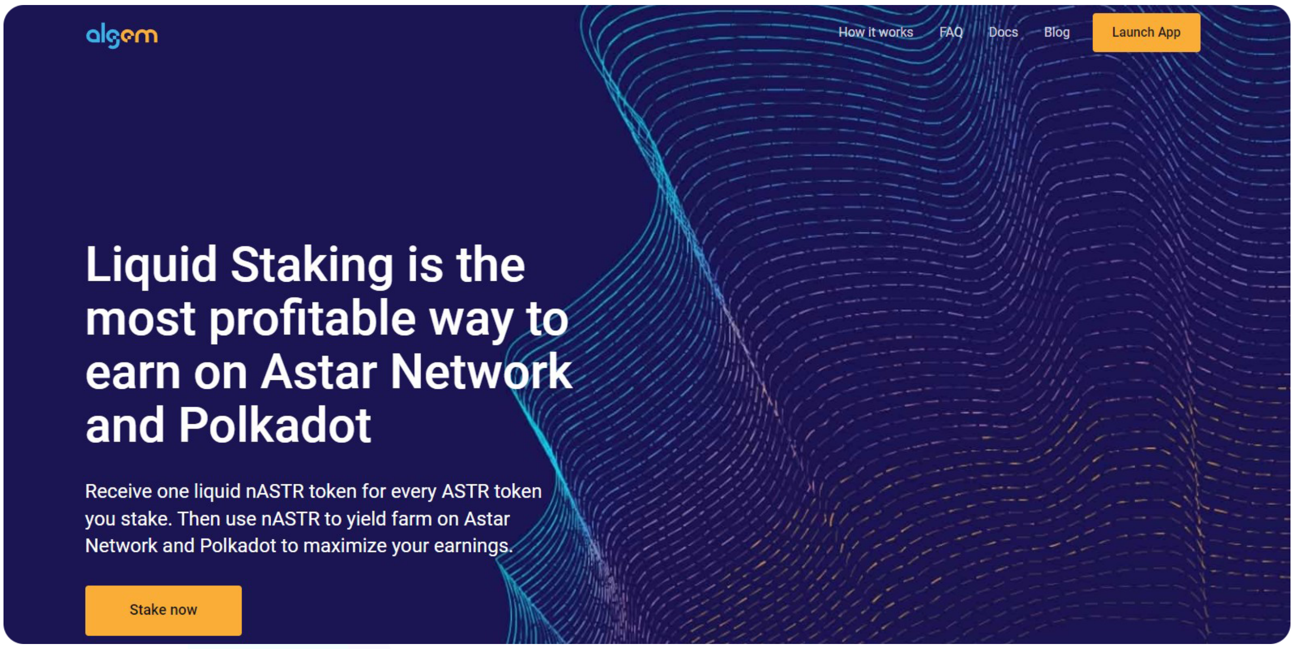
algem

Visualised Context:



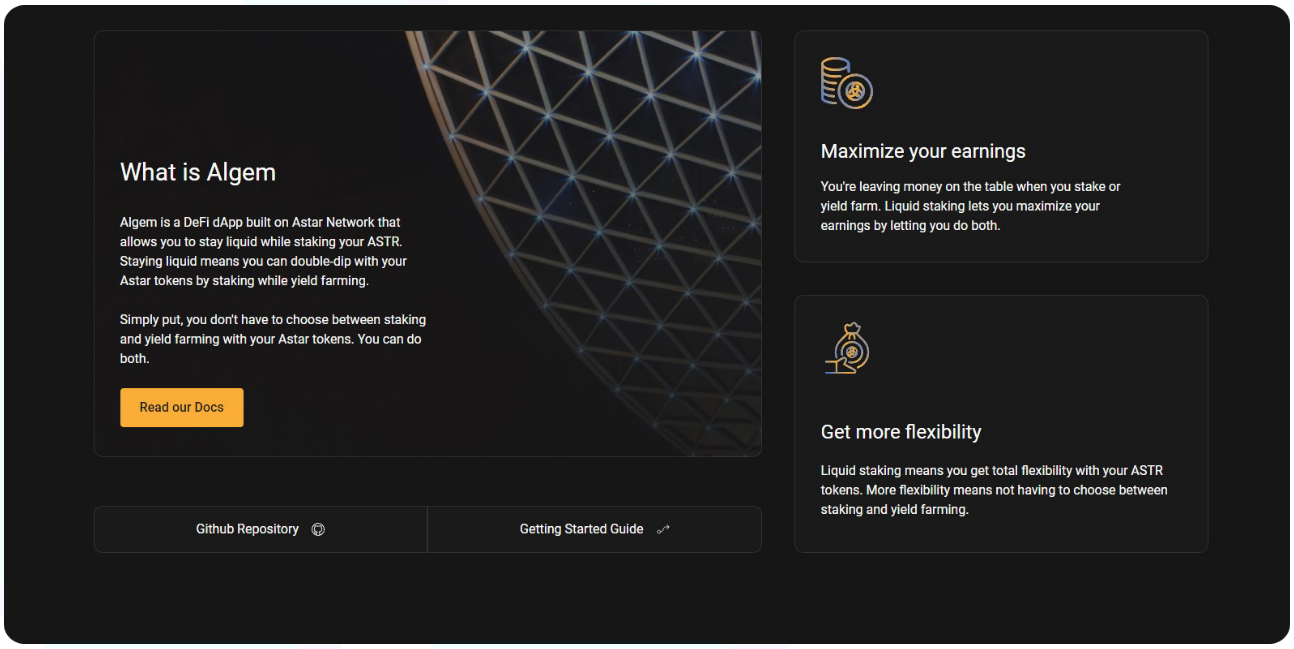
Project Name	Algem
Launch Date	TBA
Compiler Version	v^0.8.4
Language	Solidity
Network	Astar Network
Token Ticker	(nASTR)

Project Visuals:



Liquid Staking is the most profitable way to earn on Astar Network and Polkadot

Receive one liquid nASTR token for every ASTR token you stake. Then use nASTR to yield farm on Astar Network and Polkadot to maximize your earnings.



What is Algem

Algem is a DeFi dApp built on Astar Network that allows you to stay liquid while staking your ASTR. Staying liquid means you can double-dip with your Astar tokens by staking while yield farming.

Simply put, you don't have to choose between staking and yield farming with your Astar tokens. You can do both.

Maximize your earnings

You're leaving money on the table when you stake or yield farm. Liquid staking lets you maximize your earnings by letting you do both.

Get more flexibility

Liquid staking means you get total flexibility with your ASTR tokens. More flexibility means not having to choose between staking and yield farming.

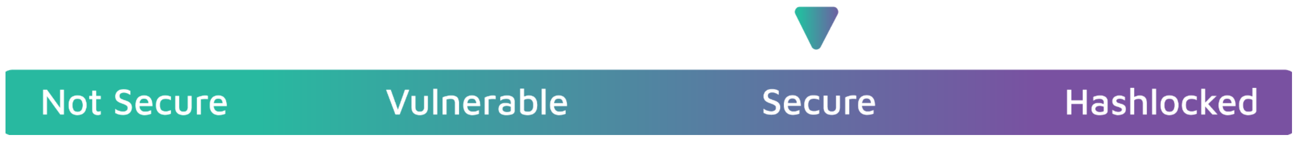
Audit scope

We at Hashlock audited the solidity code within the Algem Project, the scope of works 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	**Project Review and Security Analysis Report for Algem Protocol Smart Contracts and other factors.**
Platform	**Ethereum / Solidity**
Audit Date	**August, 2023**
Contract 1	Sio2Adapter.sol
Contract 1 MD5 Hash	d215f44138a69b185b020a4044f194c0
Contract 2	Sio2AdapterAssetManager.sol
Contract 2 MD5 Hash	ac6fab05d2158611e2b6df145ddb85ac

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. We initially identified some significant vulnerabilities that have since been addressed.



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 overview section. General overview is presented in the Function list section and all identified issues can be found in the Audit overview section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

5 High severity vulnerabilities

7 Medium severity vulnerabilities

12 Low severity vulnerabilities

3 Gas Optimisations

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.*

Standardised Checks

Main Category	Subcategory	Result
General Code Checks	Solidity/compiler version stated	Passed
	Consistent pragma version across each contract	Passed
	Outdated Solidity Version	Reviewed
	Overflow/underflow	Passed
	Correct checks, effects, interaction order	Reviewed
	Lack of check on input parameters	Reviewed
	Function input parameters check bypass	Passed
	Correct Access control	Reviewed
	Built in emergency features	Reviewed
	Correct event logs	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Reviewed
	Features claimed	Passed
	delegatecall() vulnerabilities	Passed
	Other programming issues	Reviewed
Code Specification	Correctly declared function visibility	Passed
	Correctly declared variable storage location	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Reviewed
Gas Optimization	“Out of Gas” Issue	Reviewed
	High consumption ‘for/while’ loop	Reviewed

	High consumption 'storage' storage	Reviewed
	Assert() misuse	Passed
Tokenomics Risk	The maximum limit for mintage not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Initial Audit Result: **VULNERABLE**

Revised Audit Result: **PASSED**

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
Sio2Adapter.sol - Allows users to: - Supply tokens for loan liquidity - Take out debt to borrow tokens against their collateral - Repay debts fully or partially - Withdraw their initial collateral - Add STokens - Propose a liquidationCall - Claim rewards for protocol interaction - Allows admins to: - Withdraw revenue earned by the protocol - Set set the maximum amount to borrow	**Contract achieves this functionality.**
Sio2AdapterAssetManager.sol - Periphery contract used to: - Add and remove assets that can be borrowed and used as collateral - Keep track of details about these assets, including the total amount borrowed and the amount of borrow rewards per share - Calculate health factor, available amount of collateral to borrow with or withdraw - Conversions of denominations from and to native decimals to standardised wei units	**Contract achieves this functionality.**

Code Quality

This audit scope involves the solidity smart contracts of the Algem 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 is required.

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

Audit Resources

We were given the Algem Protocol's 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 understanding the overall architecture of the protocol.

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

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

Audit Findings

High

[H-01] Sio2Adapter#_harvestRewards - Precision loss when calculating for `accCollateralRewardsPerShare` causes depositors to miss out on rewards

Description

Depending on the total amount of nASTR supplied, and the number of blocks or time between harvesting rewards, all users who have supplied nASTR can be missing out on rewards due to precision loss when calculating `accCollateralRewardsPerShare`.

Vulnerability Details

The `_harvestRewards` function updates the `accCollateralRewardsPerShare` state variable, which is used to calculate the accrued rewards for each user that has supplied nASTR to the adapter.

```
function _harvestRewards(uint256 _pendingRewards) private {
    ...
    uint256 rewardsToDistribute = receivedRewards - comissionPart;
    ...
    // set accumulated rewards per share for collateral asset
    uint256 nastrShare = (snastrToken.balanceOf(address(this)) *
        SHARES_PRECISION) / snastrToken.totalSupply();
    uint256 collateralRewards = (rewardsToDistribute *
        nastrShare *
        COLLATERAL_REWARDS_WEIGHT) / sumOfAssetShares;
    accCollateralRewardsPerShare +=
        (collateralRewards * REWARDS_PRECISION) /
        totalSupply;

    emit HarvestRewards(msg.sender, _pendingRewards);
}
```

Since the value added to `accCollateralRewardsPerShare` is calculated by dividing against `totalSupply`, there is a considerable amount of precision loss if `collateralRewards` is small and `totalSupply` is large. This is very much the case when users are interacting with the contract frequently, causing `collateralRewards` to be small.

Proof of Concept

An example of a test that simulates precision loss is shown below.

```
function testHarvestRewardsPrecisionLoss() public {
    address user2 = makeAddr("user2");
    nastr.mint(user2, 1e72);


    uint256 depositAmount = 1_000_000 ether;
    vm.prank(user);
    adapter.supply(depositAmount);
    uint256 colRewardsPerShareBefore = adapter.accCollateralRewardsPerShare();


    //test harvest rewards
    for (uint256 i = 0; i < 20; i++) {
        vm.roll(block.number + 2);
        vm.startPrank(user2);
        nastr.approve(address(adapter), 1);
        adapter.supply(1);
        vm.stopPrank();
    }
    uint256 colRewardsPerShareAfter = adapter.accCollateralRewardsPerShare();
    assertGt(colRewardsPerShareAfter, colRewardsPerShareBefore);
}
```

Impact

Users that supply nASTR tokens will lose out on a considerable amount of rewards or all rewards.

Recommendation

Increase the number of decimals of the `REWARDS_PRECISION` constant to `1e36`.

Status

Resolved

[H-02] Sio2Adapter#_updateUserRewards - `increaseAssetsTotalBorrowed` is only called per-user interaction, resulting in an outdated `asset.totalBorrowed` value

Description

`Sio2Adapter._updateUserRewards` increases the `asset.totalBorrowed` value each time a user updates their own rewards and debts. This results in an outdated `asset.totalBorrowed` value, as borrowers who have not interacted with the `Sio2Adapter` contract will not have their accrued debts reflected in `asset.totalBorrowed`.

Vulnerability Details

`Sio2Adapter._updateUserRewards` calls `assetManager.increaseAssetsTotalBorrowed` to increase the `asset.totalBorrowed` value each time a user updates their own rewards, collateral and debts.

```
function _updateUserRewards(address _user) private {
    ...
    // update total amount of total borrowed for current asset
    assetManager.increaseAssetsTotalBorrowed(assetName, debtToHarvest);
    ...
}
```

Since this happens on a per-user interaction basis, a user has to interact with the `Sio2Adapter` contract to have their accrued debts reflected in `asset.totalBorrowed`. This results in an outdated `asset.totalBorrowed` value that's lower than the actual value.

The `asset.totalBorrowed` value is used to calculate the increases in `asset.accBorrowedRewardsPerShare` and `asset.accBTokensPerShare`.

In `Sio2AdapterAssetManager`:

```
function increaseAccBorrowedRewardsPerShare(string memory _assetName, uint256
_assetRewards) external onlyAdapter {
    Asset storage asset = assetInfo[_assetName];
    asset.accBorrowedRewardsPerShare += _assetRewards * rewardsPrecision /
asset.totalBorrowed;
}

function increaseAccBTokensPerShare(string memory _assetName, uint256 _income)
external onlyAdapter {
    Asset storage asset = assetInfo[_assetName];
    asset.accBTokensPerShare += _income * rewardsPrecision / asset.totalBorrowed;
}
```

This means that the user will accrue more borrowed rewards and debt than intended.

Impact

The user will accrue a lot more borrowed rewards and debt than intended. This vulnerability is combined together with [H-03] to amplify the impact.

Recommendation

Instead of a per-user interaction basis, the `asset.totalBorrowed` value should be updated on a global basis inside `_updatePools`. This can simply be done with the following code snippet:

```
function _updatePools() private {
    ...
    // update bToken debts
    for (uint256 i; i < assetsLen; ) {
        ( , , ,
            address assetBTokenAddress, ,
            uint256 assetLastBTokenBalance,
            uint256 assetTotalBorrowed, , ,
        ) = assetManager.assetInfo(assets[i]); // prettier-ignore

        if (assetTotalBorrowed > 0) {
            uint256 bTokenBalance = IERC20Upgradeable(assetBTokenAddress)
                .balanceOf(address(this));

            // add missing zeros for correct calculations
            bTokenBalance = assetManager.to18DecFormat(
                assetBTokenAddress,
                bTokenBalance
            );

            uint256 income;

            if (bTokenBalance > assetLastBTokenBalance) {
                income = bTokenBalance - assetLastBTokenBalance;
+               assetManager.increaseAssetsTotalBorrowed(assets[i], income);
                assetManager.increaseAccBTokensPerShare(assets[i], income);
                assetManager.updateLastBTokenBalance(assets[i]);
            }
        }

        unchecked {
            ++i;
        }
    }
    emit UpdatePools(msg.sender);
}
```

Remove the call from `_updateUserRewards`:

```
function _updateUserRewards(address _user) private {
    User storage user = userInfo[_user];
    uint256 userBAssetsLen = user.borrowedAssets.length;

    // moving by borrowing assets for current user
    for (uint256 i; i < userBAssetsLen; ) {
        ( , string memory assetName, , , , , , ,
            uint256 assetAccBTokensPerShare,
            uint256 assetAccBorrowedRewardsPerShare // @audit make sure that this
variable is also adjusted to be 1e36
        ) = assetManager.assetInfo(user.borrowedAssets[i]); // prettier-ignore

        // update bToken debt
        uint256 debtToHarvest = (debts[_user][assetName] *
            assetAccBTokensPerShare) /
            rewardsPrecision -
            userBTokensIncomeDebt[_user][assetName];
        debts[_user][assetName] += debtToHarvest;
        userBTokensIncomeDebt[_user][assetName] =
            (debts[_user][assetName] * assetAccBTokensPerShare) /
            rewardsPrecision;

        // harvest sio2 rewards amount for each borrowed asset
        user.rewards +=
            (debts[_user][assetName] * assetAccBorrowedRewardsPerShare) /
            rewardsPrecision -
            userBorrowedRewardDebt[_user][assetName];

        userBorrowedRewardDebt[_user][assetName] =
            (debts[_user][assetName] * assetAccBorrowedRewardsPerShare) /
            rewardsPrecision;

-           // update total amount of total borrowed for current asset
-           assetManager.increaseAssetsTotalBorrowed(assetName, debtToHarvest);

        unchecked {
            ++i;
        }
    }
    ...
}
```

Status

Resolved

[H-03] Sio2Adapter#_updates - Only one user's rewards, collateral and debts are updated per block

Description

`Sio2Adapter._updates` only updates the rewards and debts of one user per block. This results in a delay in the accrual of rewards and debts for other users, and even potentially an indefinite DoS if that user's transactions are being front-run.

Vulnerability Details

The `_updates` function checks if the current block number is greater than the last updated block number. If it is, it updates the rewards, collateral and debts of the caller, and then updates `lastUpdatedBlock`.

```
function _updates(address _user) private {
    ...
    if (block.number > lastUpdatedBlock) {
        // update collateral and debt accumulated rewards per share
        _updatePools();

        // update user's rewards, collateral and debt
        _updateUserRewards(_user);

        lastUpdatedBlock = block.number;
    }
    ...
}
```

`lastUpdatedBlock` is a global state variable, such that all users who interact with `Sio2Adapter` check and update the same `lastUpdatedBlock` value. This means that within a block, if another user has already updated their rewards, collateral and debts, the other users will not be able to update their rewards, collateral and debts until the next block.

Impact

Only one user can update their rewards, collateral and debts per block. This results in a very drastic delay in the accrual of rewards and debts for users. If the user's transactions are being targeted via front-running, then a DoS can occur such that the user is never able to update their rewards and debts. As a result of this, there are multiple impacts:

1. Pending rewards cannot be claimed until they've been updated. Hence, users will be unable to claim their rewards unless they're the first user in the block to interact with `Sio2Adapter`. This is very unlikely to happen for normal users, as MEV bots will bid to front-run and interact with `Sio2Adapter` first.

2. The `totalSupply` state variable that tracks the total amount of collateral in the `Sio2Adapter` contract will be heavily inaccurate and much lower than its actual value. See [H-04] for the impacts of this.

Recommendation

Inside the `User` struct, add a `lastUpdatedBlock` variable that stores the last updated block number for that user.

```
struct User {
    ...
    uint256 lastUpdatedBlock;
    ...
}
```

Check and update the `lastUpdatedBlock` value of the caller instead of the global `lastUpdatedBlock` value before `_updateUserRewards(_user)` is called.

```
function _updates(address _user) private {
    // check sio2 rewards
    uint256 pendingRewards = incentivesController.getUserUnclaimedRewards(
        address(this)
    );

    // claim sio2 rewards if there is some
    if (pendingRewards > 0) _harvestRewards(pendingRewards);

    if (block.number > lastUpdatedBlock) {
        // update collateral and debt accumulated rewards per share
        _updatePools();

        lastUpdatedBlock = block.number;
    }

    if (block.number > userInfo[_user].lastUpdatedBlock) {
        // update user's rewards, collateral and debt
        _updateUserRewards(_user);

        userInfo[_user].lastUpdatedBlock = block.number;
    }

    emit Updates(msg.sender, _user);
}
```

Status

Resolved

[H-04] Sio2AdapterAssetManager#calcEstimateUserCollateralUSD - Incorrect `estAccSTokensPerShare` calculation results in an incorrect collateral estimate and `availableCollateralUSD` calculation

> This finding is related to [L-11], [H-03].

Description

The `estAccSTokensPerShare` calculation in `Sio2AdapterAssetManager.calcEstimateUserCollateralUSD` divides by `adapter.totalSupply` instead of the last updated sToken balance. This results in the estimated collateral being underestimated and the `availableCollateralUSD` calculation being overestimated.

This issue is also in `Sio2Adapter._updatePools` and `Sio2Adapter._harvestRewards`.

Vulnerability Details

The issue is present in three places.

1. To calculate the estimated accumulated sTokens per share, the change in snASTR balance is divided by the current total supply as opposed to the last updated sToken balance.

```
function calcEstimateUserCollateralUSD(
    address _userAddr
) public view returns (uint256 coll) {
    Sio2Adapter.User memory user = adapter.getUser(_userAddr);
    // get est collateral accRPS
    uint256 estAccSTokensPerShare = adapter.accSTokensPerShare();
    uint256 estUserCollateral = user.collateralAmount;

    IERC20Upgradeable snastr = IERC20Upgradeable(adapter.snastrToken());
    if (snastr.balanceOf(address(this)) > adapter.lastSTokenBalance()) {
        estAccSTokensPerShare +=
            ((snastr.balanceOf(address(this)) - adapter.lastSTokenBalance()) *
                rewardsPrecision) /
            adapter.totalSupply(); // @audit division by `adapter.totalSupply()`
instead of `adapter.lastSTokenBalance()`
    }

    estUserCollateral +=
        (estUserCollateral * estAccSTokensPerShare) /
        rewardsPrecision -
        user.sTokensIncomeDebt;

    coll = adapter.toUSD(address(adapter.nastr()), estUserCollateral);
}
```

Outside of `deposit` and `_withdraw`, the `totalSupply` value is only updated whenever a user calls `_updateUserReward` inside `Sio2Adapter`. This will overestimate the value of `estAccSTokensPerShare`, as pending sTokens accrued from collateral rewards that users have not updated are not taken into account (`totalSupply < lastSTokenBalance`).

2. The same issue is also in `Sio2Adapter._updatePools`. The `accSTokensPerShare` variable is updated by dividing the change in sToken balance by the `totalSupply` state variable.

```
function _updatePools() private {
    uint256 currentSTokenBalance = snastrToken.balanceOf(address(this));
    string[] memory assets = assetManager.getAssetsNames();
    uint256 assetsLen = assets.length;

    // if sToken balance was changed, lastSTokenBalance updates
    if (currentSTokenBalance > lastSTokenBalance) {
        accSTokensPerShare +=
            ((currentSTokenBalance - lastSTokenBalance) *
                rewardsPrecision) /
            totalSupply; // @audit `totalSupply` is used instead of
`lastSTokenBalance`
    }
    ...
}
```

As a result, `accSTokensPerShare` will be a lot higher than its actual value, which means that users will accrue a lot more collateral than intended.

3. The same issue is also in `Sio2Adapter._harvestRewards`. The `accCollateralRewardsPerShare` variable is updated by dividing the amount of collateral rewards harvested by the `totalSupply` state variable.

```
function _harvestRewards(uint256 _pendingRewards) private {
    ...

    // set accumulated rewards per share for collateral asset
    uint256 nastrShare = (snastrToken.balanceOf(address(this)) *
        SHARES_PRECISION) / snastrToken.totalSupply();
    uint256 collateralRewards = (rewardsToDistribute *
        nastrShare *
        collateralRewardsWeight) / sumOfAssetShares;
    accCollateralRewardsPerShare +=
        (collateralRewards * rewardsPrecision) /
        totalSupply;

    emit HarvestRewards(msg.sender, _pendingRewards);
}
```

As a result, the `accCollateralRewardsPerShare` will be a lot higher than its actual value, which means that users will accrue more collateral rewards than intended.

Impact

The return values of `availableCollateralUSD` will be overestimated, which will result in users being able to borrow and withdraw more than they should be able to. This exposes the `Sio2Adapter` contract to a lot more risk.

Users will accrue a lot more collateral rewards and collateral than intended.

[H-03] is combined with this vulnerability to amplify the impact further, as `totalSupply` is only updated per-user interaction per-block. This results in `totalSupply << lastSTokenBalance` (`totalSupply` is much less than `lastSTokenBalance`).

Proof of Concept

To demonstrate the third issue with more collateral rewards being accrued than intended, run the following test inside `Sio2Adapter.t.sol`:

```
function testClaimRewardsTwice() public {
    vm.startPrank(user);
    adapter.supply(10000 ether);
    adapter.borrow("BUSD", 10 ether);
    vm.stopPrank();

    uint256 adapterSTokenBal = snastr.balanceOf(address(adapter));
    // Simulate collateral interest being accrued
    deal(address(snastr), address(adapter), adapterSTokenBal + 1 ether, true);

    vm.startPrank(user);

    vm.roll(100);
    adapter.claimRewards();

    vm.roll(101);
    // This will revert due to underflow.
    // This is because `rewardPool < rewardsToClaim`
    // The reason why is because the user has accrued more rewards than it's entitled
to.
    adapter.claimRewards();

    vm.stopPrank();
}
```

Recommendation

1. To calculate `estAccSTokensPerShare`, divide by `adapter.lastSTokenBalance()` instead of `adapter.totalSupply`.

```
function calcEstimateUserCollateralUSD(
    address _userAddr
) public view returns (uint256 coll) {
    Sio2Adapter.User memory user = adapter.getUser(_userAddr);
    // get est collateral accRPS
    uint256 estAccSTokensPerShare = adapter.accSTokensPerShare();
    uint256 estUserCollateral = user.collateralAmount;

    IERC20Upgradeable snastr = IERC20Upgradeable(adapter.snastrToken());
    if (snastr.balanceOf(address(this)) > adapter.lastSTokenBalance()) {
        estAccSTokensPerShare +=
            ((snastr.balanceOf(address(this)) - adapter.lastSTokenBalance()) *
                rewardsPrecision) /
-               adapter.totalSupply();
+               adapter.lastSTokenBalance();
    }
    ...
}
```

2. To calculate `accSTokensPerShare`, divide by `lastSTokenBalance` instead of `totalSupply`.

```
function _updatePools() private {
    uint256 currentSTokenBalance = snastrToken.balanceOf(address(this));
    string[] memory assets = assetManager.getAssetsNames();
    uint256 assetsLen = assets.length;

    // if sToken balance was changed, lastSTokenBalance updates
    if (currentSTokenBalance > lastSTokenBalance) {
        accSTokensPerShare +=
            ((currentSTokenBalance - lastSTokenBalance) *
                rewardsPrecision) /
-               totalSupply;
+               lastSTokenBalance;
    }
    ...
}
```

3. To update `accCollateralRewardsPerShare`, divide by the current sToken balance instead of `totalSupply`.

```
function _harvestRewards(uint256 _pendingRewards) private {
    ...

    // set accumulated rewards per share for collateral asset
    uint256 nastrShare = (snastrToken.balanceOf(address(this)) *
        SHARES_PRECISION) / snastrToken.totalSupply();
    uint256 collateralRewards = (rewardsToDistribute *
        nastrShare *
        collateralRewardsWeight) / sumOfAssetShares;
    accCollateralRewardsPerShare +=
        (collateralRewards * rewardsPrecision) /
-           totalSupply;
+           snastrToken.balanceOf(address(this));

    emit HarvestRewards(msg.sender, _pendingRewards);
}
```

Status

Resolved

[H-05] Sio2Adapter - Underwater positions cannot be liquidated due to `ltFactor > 10000`

Description

Increasing the liquidation threshold by setting `ltFactor = 12000` results in the health factor of borrow positions to be greater than intended. Underwater positions inside SiO2 Finance's lending pools cannot be liquidated inside `Sio2Adapter` due to the health factor being greater than 1.

Vulnerability Details

The `ltFactor` variable is used to adjust the liquidation threshold relative to SiO2 Finance. Setting an `ltFactor > 10000` will cause the liquidation threshold to be greater than the value set by SiO2 Finance's lending pools.

```
    function getLT() public view returns (uint256) {
        return collateralLT * ltFactor / 10_000;
    }
```

The liquidation threshold is used to determine the health factor of an open position. A higher liquidation threshold results in a higher health factor.

```
    function getLiquidationParameters(
        address _user
    ) public update(_user) returns (uint256 hf, uint256 debtUSD) {
        debtUSD = assetManager.calcEstimateUserDebtUSD(_user);
        require(debtUSD > 0, "User has no debts");
        uint256 collateralUSD = toUSD(
            address(nastr),
            userInfo[_user].collateralAmount
        );
        hf =
            (collateralUSD * getLT() * 1e18) /
            RISK_PARAMS_PRECISION /
            debtUSD;
    }
```

An open borrow position can only be liquidated if its `health factor < 1`. Setting `ltFactor > 10000` will result in the health factor being greater than 1, which means that underwater positions cannot be liquidated.

```
    function liquidationCall(
        string memory _debtAsset,
        address _user,
        uint256 _debtToCover
    ) external returns (uint256) {
        ...
        // check user HF, debtUSD and update state
        (uint256 hf, uint256 userTotalDebtInUSD) = getLiquidationParameters(
            _user
        );
        require(hf < 1e18, "User has healthy enough position");
        ...
    }
```

Proof of Concept

Add this test to `Sio2Adapter.t.sol`:

```
    function testLTAndLTV() public {
        uint256 depositAmount = 10_000 ether;
        uint256 depositAmountInUsd = adapter.toUSD(address(nastr), depositAmount);
        (uint256 collateralLT, uint256 liquidationPenalty, uint256 collateralLTV) =
            assetManager.getAssetParameters(address(nastr));
        console.log("Sio2 LTV:", collateralLTV);
        console.log("Sio2 LT:", collateralLT);
        console.log("Sio2 LP:", liquidationPenalty);
        console.log("LTV:", adapter.getLTV());
        console.log("LT:", adapter.getLT());
        vm.startPrank(user);
        adapter.supply(depositAmount);

        // Available collateral to borrow and withdraw in USD
        // Mock Price of NASTR is 5340158 ($0.05340158)
        uint256 estCollateralInUsd = assetManager.calcEstimateUserCollateralUSD(user);
        console.log("estCollateralInUsd:", estCollateralInUsd, estCollateralInUsd /
1e18, "USD");
        assertEq(depositAmountInUsd, estCollateralInUsd, "Collateral values don't
match");

        // Available collateral to borrow and withdraw in USD
        (uint256 availBorrowUSD1, uint256 availWithdrawUSD1) =
assetManager.availableCollateralUSD(user);
        console.log("availBorrowUSD before borrow:", availBorrowUSD1, availBorrowUSD1
/ 1e18, "USD");
```

```
        console.log("availWithdrawUSD before borrow:", availWithdrawUSD1,
availWithdrawUSD1 / 1e18, "USD");
        assertEq((depositAmountInUsd * adapter.getLTV()) / 1e4, availBorrowUSD1,
"Available borrow values don't match");

        // Borrow all available borrow amount
        adapter.borrow("BUSD", availBorrowUSD1);
        vm.stopPrank();

        (uint256 availBorrowUSD2, uint256 availWithdrawUSD2) =
assetManager.availableCollateralUSD(user);
        console.log("availBorrowUSD after borrow 2:", availBorrowUSD2, availBorrowUSD2
/ 1e18, "USD");
        console.log("availWithdrawUSD after borrow 2:", availWithdrawUSD2,
availWithdrawUSD2 / 1e18, "USD");
        assertEq(availBorrowUSD2, 0, "Available borrow amount should be 0");
        assertEq(availWithdrawUSD2, 0, "Available withdraw amount should be 0");

        // Get liquidation parameters with ltvFactor = 8000, ltFactor = 12000
        (uint256 hf1,) = adapter.getLiquidationParameters(user);
        console.log("Health factor with ltFactor = 12000:", hf1);

        // Get liquidation parameters with ltvFactor = 8000, ltFactor = 10000
        adapter.setParamsFactors(80000, 10000);
        (uint256 hf2,) = adapter.getLiquidationParameters(user);
        console.log("Health factor with ltFactor = 10000:", hf2);

        assertLt(hf1, hf2, "ltFactor = 12000 results in higher health factor!");
    }
```

The test should fail.

```
[FAIL. Reason: Assertion failed.] testLTAndLTV() (gas: 972688)
Logs:
  Sio2 LTV: 8000
  Sio2 LT: 8500
  Sio2 LP: 10250
  LTV: 6400
  LT: 10200
  estCollateralInUsd: 534015800000000000000 534 USD
  availBorrowUSD before borrow: 341770112000000000000 341 USD
  availWithdrawUSD before borrow: 534015800000000000000 534 USD
  availBorrowUSD after borrow 2: 0 0 USD
  availWithdrawUSD after borrow 2: 0 0 USD
  Health factor with ltFactor = 12000: 1593750000000000000
  Health factor with ltFactor = 10000: 1328125000000000000
  Error: ltFactor = 12000 results in higher health factor!
  Error: a < b not satisfied [uint]
    Value a: 1593750000000000000
    Value b: 1328125000000000000
```

Impact

Underwater loan positions cannot be liquidated through `Sio2Adapter`. These positions will be liquidated through SiO2 Finance, resulting in potential insolvency of the `Sio2Adapter` contract as these liquidated positions will not update the liquidated user's collateral balance. Furthermore, since the user's debt amount isn't updated, this will result in incorrect collateral and debt values for the user.

Recommendation

Reduce liquidation factor to 80% (`8000`).

Status

Resolved

Medium

[M-01] Sio2Adapter#_harvestRewards - Unbounded loops when iterating over assets from the asset manager could cause out of gas error

Description

Because the `Sio2AssetManager` allows for an unlimited number of BTokens to be added, when the `Sio2Adapter` contract iterates over these two data structures during rewards harvest, the contract may revert when a user tries to interact with the contract.

Vulnerability Details

The `_harvestRewards` function loops over all BTokens twice to calculate how rewards are to be distributed:

```
for (uint256 i; i < assetsLen; i++) {
    ...
}

// set accumulated rewards per share for each borrowed asset
// needed for sio2 rewards distribution
for (uint256 i; i < assetsLen; ) {
    ...
}
```

Depending on the amount of BTokens that the contract supports, calls to `_harvestReward` may run out of gas, resulting in a denial of service of all external-facing functions.

Impact

All external-facing functions that require rewards to be harvested and updated may revert due to running out of gas.

Recommendation

It's recommended that the `Sio2Adapter` allows for the specification of an offset and length so that rewards can be collected in batches, or even rewards that the user is eligible for specifically.

Status

Resolved

[M-02] Sio2Adapter#repayFull/repayPart - Users can immediately lose funds if they attempt to repay their debt with native tokens in conjunction with ERC20 borrowed

Description

Users can attempt to repay their debt by supplying tokens however, the `repayFull` and `repayPart` function includes the payable keyword with no logic to stipulate if WASTR was actually borrowed or not.

Vulnerability Details

The `repayFull` and `repayPart` functions are `payable` and call `_repay`. However, `_repay` does not contain any logic to check that WASTR was actually borrowed for cases where `msg.value > 0`.

```
function _repay(
    string memory _assetName,
    uint256 _amount,
    address _user
) private {
    ...
    if (assetAddress != address(WASTR)) {
        userBal = asset.balanceOf(msg.sender);

        // add missing zeros for correct calculations if needed
        userBal = assetManager.to18DecFormat(assetAddress, userBal);

        require(userBal >= _amount, "Not enough wallet balance to repay");
    }
    ...
}
```

A user can lose funds if they send ASTR inside their repay transaction, but don't intend to repay their WASTR loan.

Impact

Should users attempt to pay their ERC20 token debt with a native token, those tokens may be locked in the contract forever.

Recommendation

It's recommended that the repay functions include some logic to determine whether a users debt is to be repaid using ERC20 tokens or Native tokens by supplying the asset name which correlates to `address(0)` and using `msg.value` in conjunction with the desired amount the user wishes to pay back.

Status

Resolved

[M-03] Sio2Adapter#borrow - Debt accounting and real amount borrowed are inconsistent due to precision loss

Description

Due to the precision loss from converting the amount to borrow from 18 decimal format to the decimals of the borrowed asset, the user's debt increase differs from the actual amount borrowed.

Vulnerability Details

The `borrow` function increases the user's debt and total borrowed of the asset by `_amount`, but actually borrows `nativeAmount`.

```
function borrow(
    string memory _assetName,
    uint256 _amount
) external update(msg.sender) nonReentrant {
    ...
    debts[msg.sender][_assetName] += _amount;
    assetManager.increaseAssetsTotalBorrowed(_assetName, _amount);
    ...
    uint256 nativeAmount = assetManager.toNativeDecFormat(
        assetAddr,
        _amount
    );
    pool.borrow(assetAddr, nativeAmount, 2, 0, address(this));
    ...
}
```

`nativeAmount` is calculated by reducing `_amount` to the asset's native decimal format, which results in precision loss as a result of integer division.

```
function toNativeDecFormat(
    address _tokenAddress,
    uint256 _amount
) external view returns (uint256) {
    if (ERC20Upgradeable(_tokenAddress).decimals() < 18) {
        return
            _amount /
            10 ** (18 - ERC20Upgradeable(_tokenAddress).decimals());
    }
    return _amount;
}
```

Impact

Since `nativeAmount <= _amount`, there are instances where a user can receive less in borrowed assets than they're meant to receive.

Recommendation

Increase the user's debts and total borrowed for the assets by `to18DecFormat(nativeAmount)`, as opposed to just using `_amount`.

Status

Resolved

[M-04] Sio2Adapter#initialize - `Sio2Adapter` cannot be initialized due to circular dependency with `Sio2AdapterAssetManager`

Description

`Sio2Adapter.initialize` will revert, as `Sio2AdapterAssetManager` has not set the `adapter` state variable.

Vulnerability Details

`Sio2Adapter.initialize` calls the `Sio2AdapterAssetManager.getAssetWeights` function to assign a value to `collateralRewardsWeight`.

```
function initialize(
    ...
) external initializer {
    ...
    collateralRewardsWeight = assetManager.getAssetWeight(address(nastr));
    ...
}
```

This in turn calls `adapter.incentivesController` inside `Sio2AdapterAssetManager`.

```
function getAssetWeight(address asset) external view returns (uint256) {
    ISio2IncentivesController ic =
ISio2IncentivesController(adapter.incentivesController());
    ...
}
```

However, the `adapter` state variable inside `Sio2AdapterAssetManager` has not been set, since `Sio2Adapter` is still being initialized. This results in an EVM revert.

Impact

The `Sio2Adapter` contract cannot be initialized.

Recommendation

Set the `collateralRewardsWeight` value in a separate function that's called after `initialize` and `Sio2AdapterAssetManager.setAdapter`. Make sure that all of these calls happen inside one transaction to prevent any potential front-running attacks or unintended behavior.

Status

Resolved

> The `getAssetWeight` function now takes the address of the incentives controller as input instead of calling `Sio2Adapter`.

[M-05] `updateParams` can be called more than once

Description

`Sio2AdapterAssetManager.updateParams` can be by the owner more than once, which would result in 'per-share' variables being scaled up by 24 decimal places more than once.

Vulnerability Details

The `updateParams` function in both `Sio2Adapter` and `Sio2AdapterAssetManager` can be called more than once by the owner, which would result in the 'per-share' variables being scaled up by 24 decimal places more than once.

Impact

The 'per-share' variables will be scaled up by 24 decimal places more than once, which would result in abnormally higher rewards and debts for users.

Recommendation

Add a `paramsUpdated` boolean variable to both `Sio2Adapter` and `Sio2AdapterAssetManager` to prevent `updateParams` from being called more than once.

```
...
bool private _paramsUpdated;
...
function updateParams() public onlyOwner {
    require(!_paramsUpdated, "Params already updated");
    ...
    paramsUpdated = true;
}
```

Status

Resolved

[M-06] Sio2AdapterAssetManager#removeAsset - A user with an outstanding debt in a removed asset will lose pending rewards and will not be able to withdraw their collateral or borrow any asset

Description

A removed asset will have its Asset struct reset to its default values. This results in a loss of pending rewards/debts and a DoS when a user with an outstanding debt tries to borrow or withdraw their collateral.

Vulnerability Details

In `Sio2Adapter`, when a user borrows a new asset, the `assetName` gets added into their `borrowedAssets` array inside their User struct. This results in the following vulnerabilities:

1. The `borrowedAssets` array is used whenever a user triggers an update on their user rewards.

```
function _updateUserRewards(address _user) private {
    User storage user = userInfo[_user];
    uint256 userBAssetsLen = user.borrowedAssets.length;

    // moving by borrowing assets for current user
    for (uint256 i; i < userBAssetsLen; ) {
        ( , string memory assetName, , , , , ,
            uint256 assetAccBTokensPerShare,
            uint256 assetAccBorrowedRewardsPerShare
        ) = assetManager.assetInfo(user.borrowedAssets[i]); // prettier-ignore

        // update bToken debt
        uint256 debtToHarvest = (debts[_user][assetName] *
            assetAccBTokensPerShare) /
            rewardsPrecision -
            userBTokensIncomeDebt[_user][assetName];
        debts[_user][assetName] += debtToHarvest;
        userBTokensIncomeDebt[_user][assetName] =
            (debts[_user][assetName] * assetAccBTokensPerShare) /
            rewardsPrecision;

        // harvest sio2 rewards amount for each borrowed asset
        user.rewards +=
            (debts[_user][assetName] * assetAccBorrowedRewardsPerShare) /
            rewardsPrecision -
            userBorrowedRewardDebt[_user][assetName];

        userBorrowedRewardDebt[_user][assetName] =
            (debts[_user][assetName] * assetAccBorrowedRewardsPerShare) /
            rewardsPrecision;

        // update total amount of total borrowed for current asset
        assetManager.increaseAssetsTotalBorrowed(assetName, debtToHarvest);

        unchecked {
            ++i;
        }
    }
    ...
}
```

The function will attempt to fetch the asset info of the removed asset, which have been reset to their default values.

```
    ( , string memory assetName, , , , , ,
        uint256 assetAccBTokensPerShare,
        uint256 assetAccBorrowedRewardsPerShare
    ) = assetManager.assetInfo(user.borrowedAssets[i]);
```

Since `assetName = ""`, `assetAccBTokensPerShare = 0` and `assetAccBorrowedRewardsPerShare = 0`, `debtToHarvest = 0` and the user will not be able to accrue any pending debt or rewards for the removed asset.

2. A user with a removed asset in their `borrowedAssets` array will not be able to withdraw or borrow, as calls to `Sio2AdapterAssetManager.estimateDebtInAsset` will revert, causing `Sio2AdapterAssetManager.calcEstimateUserDebtUSD` and hence `Sio2AdapterAssetManager.availableCollateralUSD` to also revert.

```
function estimateDebtInAsset(address _userAddr, string memory _assetName) public view
returns (uint256) {
    Asset memory asset = assetInfo[_assetName];
    ...
    // @audit `asset.bTokenAddress = address(0)` => Calls to zero address will revert
    uint256 curBBal =
ERC20Upgradeable(asset.bTokenAddress).balanceOf(address(adapter));
    ...
}
```

3. To remove the removed asset from the user's `borrowedAssets` array, the user will need to repay back their outstanding debt. However, this cannot be done, as the returned `assetAddress` from `assetManager.assetInfo` inside `_repay` will be the zero address.

```
function _repay(
    string memory _assetName,
    uint256 _amount,
    address _user
) private whenNotPaused {
    // @audit `assetAddress = address(0)`
    (, , address assetAddress, , , , , , ) = assetManager.assetInfo(
        _assetName
    );
    IERC20Upgradeable asset = IERC20Upgradeable(assetAddress);

    uint256 userBal;
    if (assetAddress != address(WASTR)) {
        // @audit this will revert if the asset is removed
        userBal = asset.balanceOf(msg.sender);
        ...
    }
    ...
}
```

Impact

1. Any pending rewards or debt that a user has in a removed asset will be lost.
2. The user will not be able to borrow any assets or withdraw any of their collateral, regardless of their health factor. Their collateral is frozen.
3. The user cannot repay their outstanding debt on the removed asset.

Recommendation

Inside `Sio2AdapterAssetManager`, keep track of removed assets through the following mapping:

```
mapping(string => Asset) public removedAssetInfo;
```

The `removeAsset` function can add to this mapping before the key-value pair is deleted from `assetInfo`:

Inside `removeAsset`:

```
function removeAsset(string memory assetName) external onlyOwner {
    ...
    removedAssetInfo[assetName] = asset;
    // Delete id since removed asset is no longer in `assets` array
    delete removedAssetInfo[assetName].id;

    delete assetInfo[assetName];
    ...
}
```

External calls to `Sio2AdapterAssetManager.assetInfo` (except in `borrow`) and internal calls to the `assetInfo` mapping used for debt/reward calculations inside `Sio2AdapterAssetManager` can instead be routed through a new function that checks if the asset exists in `assetInfo` or `removedAssetInfo`:

Inside `Sio2AdapterAssetManager`:

```
function getAssetInfo(string memory assetName) public view returns (Asset memory) {
    Asset memory asset = assetInfo[assetName];
    if (asset.addr == address(0)) {
        // Asset has been removed
        asset = removedAssetInfo[assetName];
    }
    return asset;
}
```

These modifications will achieve the following:

1. This will allow users to update their pending rewards and debts for removed assets, and allow users to pay off their outstanding debts on the removed asset.
2. Once all outstanding debts for the removed asset have been paid off, `_removeAssetFromUser` will be called, removing the asset from the user's `borrowedAssets` array. This will allow the user to call `withdraw` and `borrow` again.
3. Since `_repay` will now be able to fetch the asset info of the removed asset, the user will be able to repay their outstanding debt on the removed asset.

Status

Resolved

[M-07] Sio2AdapterAssetManager - Decimal conversion functions do not account for ERC-20 tokens with more than 18 decimals

Description

The `to18DecFormat` and `toNativeDecFormat` functions do not account for ERC-20 tokens with more than 18 decimals. This results in an accounting error where the user gets less borrowed assets than they intended but still incur the same debt.

Vulnerability Details

`to18DecFormat` and `toNativeDecFormat` return the original `_amount` if the token has 18 **or more** decimals.

```
    function to18DecFormat(address _tokenAddress, uint256 _amount) public view returns
(uint256) {
        if (ERC20Upgradeable(_tokenAddress).decimals() < 18) {
            return _amount * 10 ** (18 - ERC20Upgradeable(_tokenAddress).decimals());
        }
        return _amount;
    }

    function toNativeDecFormat(
        address _tokenAddress,
        uint256 _amount
    ) external view returns (uint256) {
        if (ERC20Upgradeable(_tokenAddress).decimals() < 18) {
            return
                _amount /
                10 ** (18 - ERC20Upgradeable(_tokenAddress).decimals());
        }
        return _amount;
    }
```

`Sio2Adapter.borrow` uses these functions to convert the input `_amount` which is reflected in 18 decimals to the token's native decimals.

```
    function borrow(
        string memory _assetName,
        uint256 _amount
    ) external update(msg.sender) nonReentrant whenNotPaused {
        ...

        uint256 nativeAmount = assetManager.toNativeDecFormat(assetAddr, _amount);
        uint256 roundedAmount = assetManager.to18DecFormat(assetAddr, nativeAmount);

        debts[msg.sender][_assetName] += roundedAmount;
        assetManager.increaseAssetsTotalBorrowed(_assetName, roundedAmount);

        ...

        pool.borrow(assetAddr, nativeAmount, 2, 0, address(this));

        ...
    }
```

However, if the token has more than 18 decimals, `_amount` remains unchanged. This means that the user borrows less tokens than specified, even though the debt recorded is the same as the amount they intended to borrow.

`to18DecFormat` is also used inside `estimateDebtInAsset`, resulting in the user's debt for assets with more than 18 decimals to be higher than intended. The amount of debt ends up being equivalent to the amount that was intended to be borrowed, instead of the amount actually borrowed.

Proof of Concept

Let's assume that BUSD has 20 decimals.

A user that intends to borrow 1000 BUSD will call `borrow("BUSD", 1000e18)`. The `borrow` function will not convert `1000e18` into BUSD's `nativeAmount` which is `1000e20` since BUSD has more than 18 decimals. The amount that `Sio2Adapter` ends up borrowing and giving to the user ends up being `1000e18 / 1e20 = 10 BUSD`, even though the user's debts are recorded as `1000e18` which is $1000 USD.

The scenario above is demonstrated through the test below. Add this test to `Sio2Adapter.t.sol`:

```
    function testAssetMoreThan18Dec() public {
        uint8 decimals = 20;

        // Set BUSD to 20 decimals
        busd.setDecimals(decimals);
        vdbusd.setDecimals(decimals);

        // Start supplying and borrowing
        vm.startPrank(user);
        adapter.supply(10000000 ether); // supply 10000000 nASTR
        adapter.borrow("BUSD", 1000 ether); // borrow 1000 BUSD

        uint256 debt = assetManager.calcEstimateUserDebtUSD(user);
        console.log("debt in USD:", debt / 1 ether);
        console.log("BUSD borrowed:", busd.balanceOf(user) / 10 ** decimals);

        // @audit this fails
        assertEq(busd.balanceOf(user), 1000 * 10 ** decimals, "user should have 1000
BUSD");

        // Repay debt
        busd.approve(address(adapter), type(uint256).max);
        adapter.repayFull("BUSD");
        assertEq(busd.balanceOf(user), 0, "user should have no BUSD left");
        uint256 debt2 = assetManager.calcEstimateUserDebtUSD(user);
        assertEq(debt2, 0, "user should have no debt");
        vm.stopPrank();
    }
```

Running the unit test shows that the user intends to borrow 1000 BUSD, but only gets 10. However, their outstanding debt is still $1000 USD.

```
[FAIL. Reason: Assertion failed.] testAssetMoreThan18Dec() (gas: 843436)
Logs:
  debt in USD: 1000
  BUSD borrowed: 10
  Error: user should have 1000 BUSD
  Error: a == b not satisfied [uint]
    Expected: 100000000000000000000000
      Actual: 1000000000000000000000
```

Impact

Users who borrow assets with more than 18 decimals receive less assets than intended by a factor of `10 ** (assetDecimals - 18)`. However, the amount of debt recorded remains the same.

Recommendation

Account for assets with more than 18 decimals by changing the functions to the following:

```
    function to18DecFormat(address _tokenAddress, uint256 _amount) public view returns
(uint256) {
        if (ERC20Upgradeable(_tokenAddress).decimals() < 18) {
            return _amount * 10 ** (18 - ERC20Upgradeable(_tokenAddress).decimals());
        }

+       if (ERC20Upgradeable(_tokenAddress).decimals() > 18) {
+           return _amount / 10 ** (ERC20Upgradeable(_tokenAddress).decimals() - 18);
+       }

        return _amount;
    }

    function toNativeDecFormat(
        address _tokenAddress,
        uint256 _amount
    ) external view returns (uint256) {
        if (ERC20Upgradeable(_tokenAddress).decimals() < 18) {
            return
                _amount /
                10 ** (18 - ERC20Upgradeable(_tokenAddress).decimals());
        }

+       if (ERC20Upgradeable(_tokenAddress).decimals() > 18) {
+           return
+               _amount * 10 ** (ERC20Upgradeable(_tokenAddress).decimals() - 18);
+       }

        return _amount;
    }
```

The POC unit test should pass.

```
[PASS] testAssetMoreThan18Dec() (gas: 846028)
Logs:
  debt in USD: 1000
  BUSD borrowed: 1000
```

Status

Resolved

Low

[L-01] Outdated version of Solidity

Description

The project uses `pragma solidity ^0.8.4`. 0.8.4 is an outdated version of Solidity

Recommendation

Use `pragma solidity 0.8.18`.

Status

Acknowledged

> Since the contracts have already been deployed, it would be impractical to upgrade the version. To avoid potential incompatibility issues with newer versions of Solidity, it was decided along with the Hashock team that it would be better to keep the contracts on version 0.8.4.

[L-02] Unnecessary indentation

Description

There are inconsistent and unnecessary indentations in the code that damages its readability.

```
mapping(address => User) public userInfo;
mapping(address => mapping(string => uint256)) public debts;
mapping(address => mapping(string => uint256)) public userBorrowedAssetID;
mapping(address => mapping(string => uint256)) public userBTokensIncomeDebt;
mapping(address => mapping(string => uint256))
    public userBorrowedRewardDebt;
```

Recommendation

Fix these inconsistencies. Write the mapping line above in one line.

Status

Resolved

[L-03] Lack of checks-effects-interactions pattern

Description

Some functions lack the Checks-Effects-Interaction (CEI) pattern. This is best practice to avoid potential reentrancy attacks.

```
function supply(uint256 _amount) external update(msg.sender) nonReentrant {
    require(_amount > 0, "Should be greater than zero");
    require(
        nastr.balanceOf(msg.sender) >= _amount,
        "Not enough nASTR tokens on the user balance"
    );

    ...

    // take nastr from user
    nastr.safeTransferFrom(msg.sender, address(this), _amount);

    // deposit nastr to lending pool
    nastr.approve(address(pool), _amount);
    pool.deposit(address(nastr), _amount, address(this), 0);

    user.collateralAmount += _amount;
    totalSupply += _amount;

    assetManager.updateBalanceInAdaptersDistributor(msg.sender);

    _updateUserCollateralIncomeDebts(user);

    emit Supply(msg.sender, _amount);
}
```

Recommendation

Perform external calls to the end of the function.

Status

Resolved

[L-04] Spelling mistake in comments

Description

There are some spelling mistakes in comments, which reduce the quality and readability of the code.

```
function _updateUserRewards(address _user) private {
    ...
    // uncrease total collateral amount by received user's collateral
    totalSupply += collateralToHarvest;
    ...
}
```

Recommendation

Fix these spelling mistakes.

Status

Resolved

[L-05] Sio2Adapter - Don't hard-code values that aren't guaranteed to be constant

Description

The contract uses hard-coded values, and sometimes even stores important state variables from external contracts as constants.

```
uint256 private constant PRICE_PRECISION = 1e8;
...
uint256 private constant COLLATERAL_REWARDS_WEIGHT = 5;
```

Recommendation

If the value comes from a state variable from an external contract, get the value using an external call.

If the value is a config for the contract and it is unknown whether it would change in the future, make it a state variable that can be changed.

Status

Resolved

> The Algem team is confident that the `PRICE_PRECISION` constant is static, as the value that it corresponds to comes from a contract that is not upgradeable and contains no logic to change it. `COLLATERAL_REWARDS_WEIGHT` was changed to be a state variable that's calculated on initialization.

[L-06] Incorrect use of Natspec comments

Description

The contract declares Natspec comments with 2 slashes instead of 3.

```
// @notice Collect accumulated b-tokens and s-tokens
```

Recommendation

Change the comments to 3 slashes.

Status

Resolved

[L-07] Sio2Adapter - repay function Natspec comments do not indicate that `_amount` param is in 18 decimals

Description

The Natspec comment for the `_amount` parameter inside the `repayPart` function does not indicate that it's formatted in 18 decimals.

```
// @dev when user calls repay(), _user and msg.sender are the same
//      and there is difference when liquidator calling function
// @param _assetName Asset name
// @param _amount Amount of tokens
function repayPart(
    string memory _assetName,
    uint256 _amount
) external payable update(msg.sender) nonReentrant {
    _repay(_assetName, _amount, msg.sender);
}
```

Recommendation

Change the comment to indicate that it's formatted in 18 decimals.

```
// @param _amount Amount of tokens in 18 decimals
```

Status

Resolved

[L-08] Sio2AdapterAssetManager#getAssetWeight - `assets` memory address array shadows the state variable `assets`

Description

`assets` is a string array in storage that stores all the names of the added assets, which is shadowed by a declared address array in `getAssetWeight` of the same name.

In storage:

```
string[] public assets;
```

In `getAssetWeight`:

```
address[] memory assets = pool.getReservesList();
```

Recommendation

Use a different name for the memory array. We recommend `assetsInPool`.

Status

Resolved

[L-09] Sio2AdapterAssetManager#removeAsset - Incomplete Natspec comment

Description

The Natspec comment for `removeAsset` is incomplete.

```
/// @notice Removes an asset and
```

Recommendation

Complete the Natspec comment.

Status

Resolved

[L-10] Sio2AdapterAssetManager#getAssetWeight - Unnecessary casting of contract types

Description

`getAssetWeight` casts the result of `adapter.incentivesController()` (which returns the contract type `ISio2IncentivesController`) to `ISio2IncentivesController`. This is unnecessary.

```
ISio2IncentivesController controller =
ISio2IncentivesController(adapter.incentivesController());
```

Recommendation

Remove the casting.

```
ISio2IncentivesController controller = adapter.incentivesController();
```

Status

Resolved

[L-11] Sio2AdapterAssetManager#estimateDebtInAsset - Incorrect `estAccBTokens` calculation results in an incorrect debt estimate and `availableCollateralUSD` calculation

> This finding is related to [H-04]

Description

The `estAccBTokens` calculation in `Sio2AdapterAssetManager.estimateDebtInAsset` divides by the current bToken balance instead of the last updated bToken balance. This results in the estimated debt being underestimated and the `availableCollateralUSD` calculation being overestimated.

Vulnerability Details

To calculate the estimated accumulated bTokens per share, the income is divided by the current bToken balance as opposed to the last updated bToken balance.

```
function estimateDebtInAsset(address _userAddr, string memory _assetName) public view
returns (uint256) {
    Asset memory asset = assetInfo[_assetName];

    uint256 bIncomeDebt = adapter.userBTokensIncomeDebt(_userAddr, _assetName);
    uint256 estDebt = adapter.debts(_userAddr, _assetName);
    uint256 estAccBTokens = asset.accBTokensPerShare;

    uint256 income;
    uint256 curBBal =
ERC20Upgradeable(asset.bTokenAddress).balanceOf(address(adapter));
    uint256 curBBal18Dec = to18DecFormat(asset.bTokenAddress, curBBal);
    if (curBBal18Dec > asset.lastBTokenBalance) {
        income = curBBal18Dec - asset.lastBTokenBalance;
    }
    if (curBBal18Dec > 0 && income > 0) {
        // @audit division by `curBBal18Dec` instead of `asset.lastBTokenBalance`
        estAccBTokens += income * rewardsPrecision / curBBal18Dec;
        estDebt += estDebt * estAccBTokens / rewardsPrecision - bIncomeDebt;
    }

    return estDebt;
}
```

This approach underestimates the value of the accrued debt in situations where `curBBal18Dec < asset.lastBTokenBalance`, which in turn results in an overestimation of the `availableCollateralUSD` calculation.

```
function availableCollateralUSD(
    address _userAddr
) public view returns (uint256 toBorrow, uint256 toWithdraw) {
    Sio2Adapter.User memory user = adapter.getUser(_userAddr);
    if (user.collateralAmount == 0) return (0, 0);
    uint256 debt = calcEstimateUserDebtUSD(_userAddr);
    uint256 userCollateral = calcEstimateUserCollateralUSD(_userAddr);
    uint256 collateralAfterLTV = (userCollateral * adapter.collateralLTV()) /
        1e4; // 1e4 is RISK_PARAMS_PRECISION
    if (collateralAfterLTV > debt) toBorrow = collateralAfterLTV - debt;
    uint256 debtAfterLTV = (debt * 1e4) / adapter.collateralLTV();
    if (userCollateral > debtAfterLTV)
        toWithdraw = userCollateral - debtAfterLTV;
}
```

Impact

External contracts that call `Sio2AdapterAssetManager.estimateDebtInAsset` or `Sio2AdapterAssetManager.availableCollateralUSD` will receive an incorrect debt estimate and available collateral calculation.

> **Auditor's Note**: A similar bug was found in [H-04], which is a high severity vulnerability. However, the impact of this bug is much lower, as the `update` modifier is called before `borrow` or `withdraw` which updates `asset.lastBTokenBalance`. This means that the `Sio2Adapter` contract will not be affected by this bug.

Recommendation

To calculate `estAccBTokens`, divide by the last bToken balance instead of the current one.

In `Sio2AdapterAssetManager.estimateDebtInAsset`:

```
function estimateDebtInAsset(address _userAddr, string memory _assetName) public view
returns (uint256) {
    Asset memory asset = assetInfo[_assetName];

    uint256 bIncomeDebt = adapter.userBTokensIncomeDebt(_userAddr, _assetName);
    uint256 estDebt = adapter.debts(_userAddr, _assetName);
    uint256 estAccBTokens = asset.accBTokensPerShare;

    uint256 income;
    uint256 curBBal =
ERC20Upgradeable(asset.bTokenAddress).balanceOf(address(adapter));
    uint256 curBBal18Dec = to18DecFormat(asset.bTokenAddress, curBBal);
    if (curBBal18Dec > asset.lastBTokenBalance) {
        income = curBBal18Dec - asset.lastBTokenBalance;
    }

    if (curBBal18Dec > 0 && income > 0) {
-           estAccBTokens += income * rewardsPrecision / curBBal18Dec;
+           estAccBTokens += income * rewardsPrecision / asset.lastBTokenBalance;
        estDebt += estDebt * estAccBTokens / rewardsPrecision - bIncomeDebt;
    }

    return estDebt;
}
```

Status

Resolved

[L-12] Sio2Adapter#_updateUserRewards - Leftover pending rewards after function is called due to order of operations

Description

The `Sio2Adapter._updateUserRewards` function distributes pending rewards to the user then updates the user's `user.collateralAmount`. Since pending rewards are calculated using `user.collateralAmount`, this order of operations results in leftover pending rewards that are not distributed to the user.

```
    function _updateUserRewards(address _user) private {
        ...
        // harvest sio2 rewards for user's collateral
        user.rewards +=
            (user.collateralAmount * accCollateralRewardsPerShare) /
            rewardsPrecision -
            user.collateralRewardDebt;
        user.collateralRewardDebt =
            (user.collateralAmount * accCollateralRewardsPerShare) /
            rewardsPrecision;

        // user collateral update
        uint256 collateralToHarvest = (user.collateralAmount *
            accSTokensPerShare) /
            rewardsPrecision -
            user.sTokensIncomeDebt;
        user.collateralAmount += collateralToHarvest;
        user.sTokensIncomeDebt =
            (user.collateralAmount * accSTokensPerShare) /
            rewardsPrecision;
        ...
    }
```

Recommendation

The order of operations should be reversed. Update the user's collateral before distributing rewards to the user.

```
    function _updateUserRewards(address _user) private {
        ...
        // user collateral update
        uint256 collateralToHarvest = (user.collateralAmount *
            accSTokensPerShare) /
            rewardsPrecision -
            user.sTokensIncomeDebt;
        user.collateralAmount += collateralToHarvest;
        user.sTokensIncomeDebt =
            (user.collateralAmount * accSTokensPerShare) /
            rewardsPrecision;

        // harvest sio2 rewards for user's collateral
        user.rewards +=
            (user.collateralAmount * accCollateralRewardsPerShare) /
            rewardsPrecision -
            user.collateralRewardDebt;
        user.collateralRewardDebt =
            (user.collateralAmount * accCollateralRewardsPerShare) /
            rewardsPrecision;
        ...
    }
```

Status

Resolved

Gas

[G-01] Sio2Adapter - Storing a user’s address in `User` struct is redundant

Description

The `User` struct stores the address of the user as one of the fields when it isn’t required since the `userInfo` mapping’s key is also the address.

Recommendation

Remove the `User.addr` field from the struct, and replace its references in the code with just the `msg.sender`.

In the case of the new user check, you can check for a non-zero user ID:

```
// check for new user. And add to arr if there is no such
if (userInfo[msg.sender].id == 0) {
    user.id = users.length;
    user.addr = msg.sender;
    users.push(msg.sender);
}
```

Status

Resolved

> Since the contract is already deployed, the struct cannot be edited. The logic was edited so that all storage entries related to the `addr` record have been removed and a new check has been added when adding a new user. In addition to the check for 'id = 0', checks have been added to ensure that the user has a zero 'collateralAmount' and zero 'rewards', as there exists a user with index 0.

[G-02] Sio2Adapter - Cache array length before looping

Description

Before iterating through an array with a for-loop, the length of the array can be cached in memory so that its value does not need to be accessed on each iteration of the loop.

Recommendation

Cache the array length by assigning it to a variable in memory before looping through the array.

Status

Resolved

[G-03] Sio2AdapterAssetManager - `bTokenExist` and `assetNameExist` mappings are unnecessary

Description

`bTokenExist` and `assetNameExist` are mappings that store whether a bToken or asset name exists. They are used to check if a particular asset has already been added. However, these mappings are unnecessary, as the `assetInfo` mapping already stores all the added assets by their `assetName`.

Recommendation

Deprecate the use of the `bTokenExist` and `assetNameExist` mappings. To check if a particular asset has already been added, only the following require statement is necessary:

```
require(keccak256(abi.encodePacked(_assetName)) != keccak256(""), "Empty asset name");
```

> **Auditor's Note:** The optimal solution would be to change `assetInfo` into a mapping from `address` to `Asset` and the `assets` array from `string` to `address` accordingly. However, since the contracts are already deployed, this change is not possible. The next best solution is to deprecate the use of `bTokenExist` and `assetNameExist` and only use `assetInfo` to check if an asset has already been added. If more assurance that an asset has not been added is required, the following mapping can be added:
>
> ```
> mapping(address => bool) public assetAddressExist;
> ```
>
> This mapping can be used to check if an asset has already been added by its address.

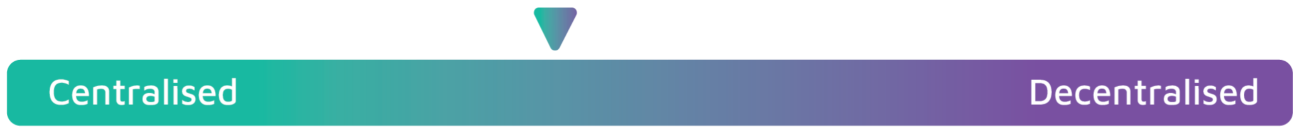
Status

Resolved

Centralisation

The project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.



Conclusion

After Hashlocks analysis, the Algem project seems to have a sound and well tested code base, however our findings need to be resolved in order to achieve security. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented 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 are 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 whitebox 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, 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 contracts 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 are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree 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 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

Hashlock.

Hashlock.

Hashlock Pty Ltd