hashlock.

Security Audit

Aqualis Code Review (DeFi)

December 2024

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	10
Audit Resources	10
Dependencies	10
Severity Definitions	11
Status Definitions	12
Audit Findings	13
Centralisation	29
Conclusion	30
Our Methodology	31
Disclaimers	33
About Hashlock	34

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

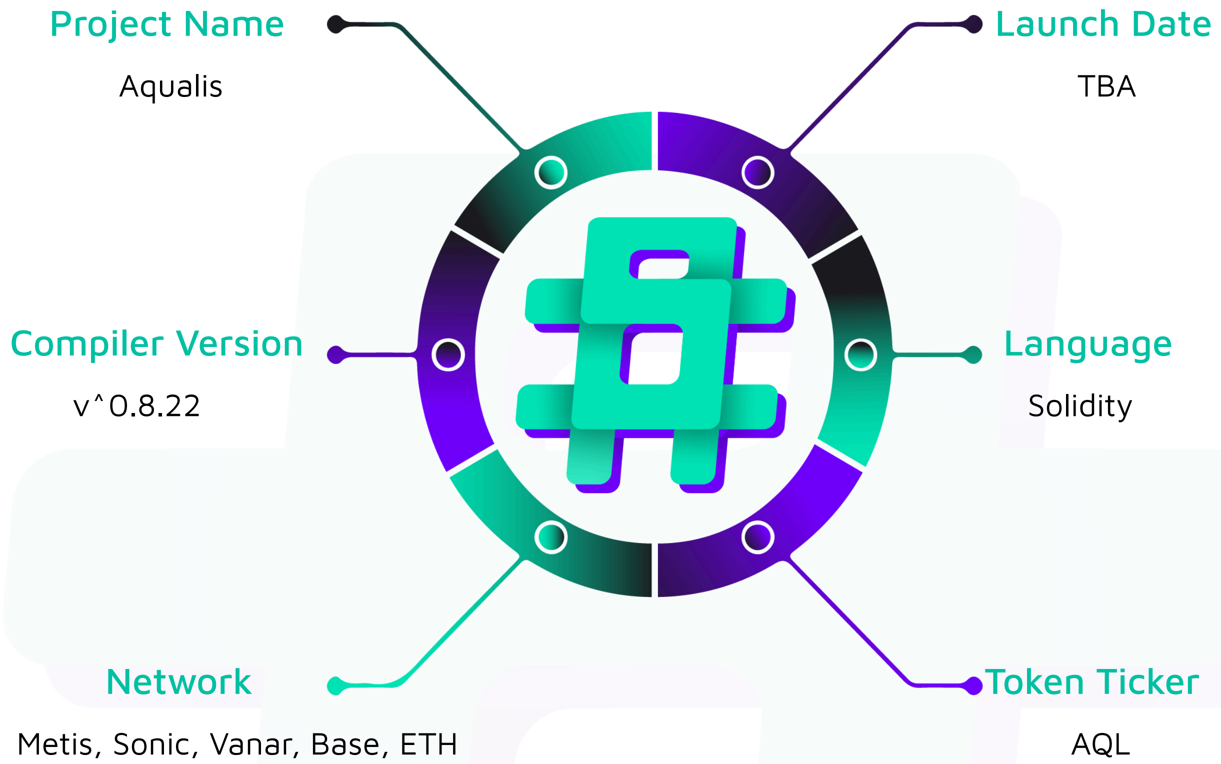
The Aqualis team partnered with Hashlock to conduct a security audit of their AqualisSP.sol and only the implemented fixes in AqualisSPV1.sol smart contracts. A new version of the Aqualis smart contract (AqualisSPv1.sol) represents a refined and fixed iteration of the initial version (AqualisSP.sol). This update addresses issues identified during the initial audit conducted by Hashlock. 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

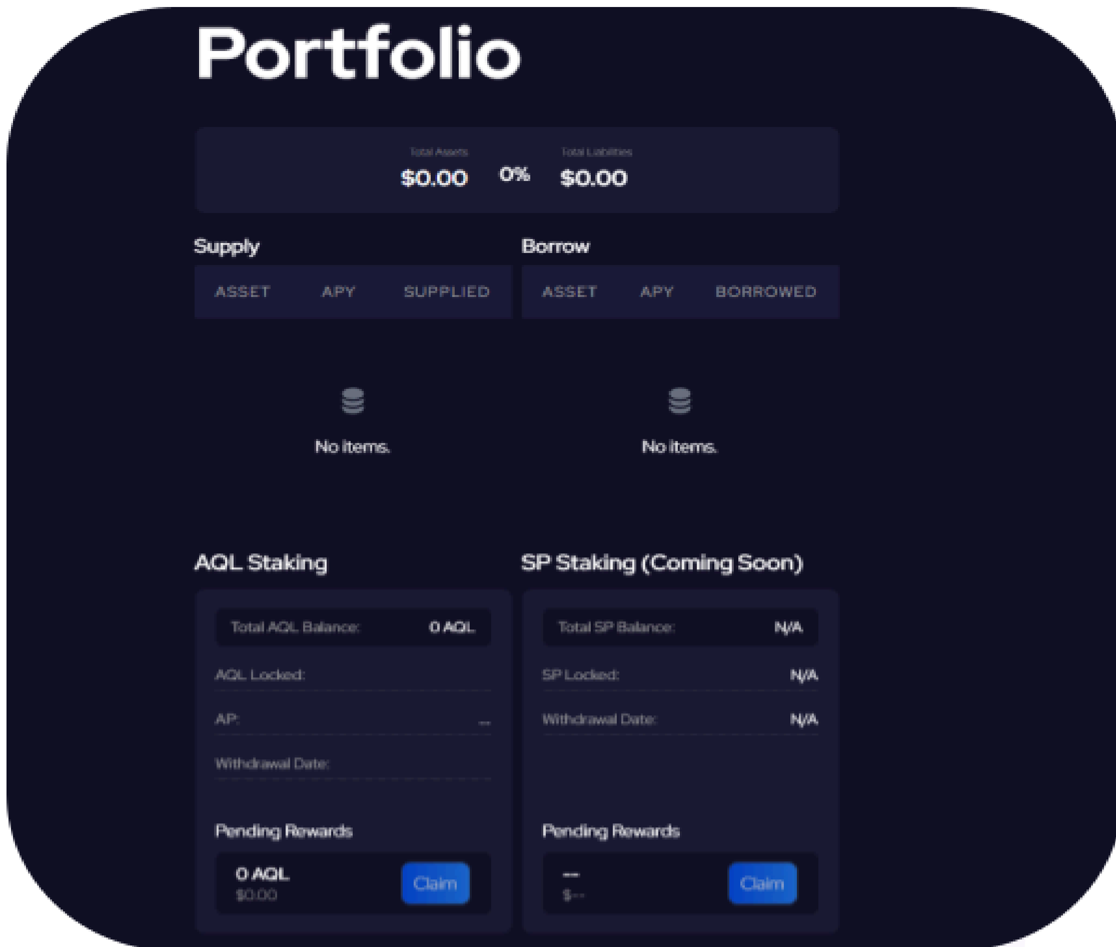
Aqualis Protocol is a multi-functional DeFi protocol with both trading and lending capabilities. What sets Aqualis Protocol apart from existing DEXs is the ability to unlock liquidity locked in a DEX liquidity pool for use in lending, without impacting the trading efficiency. This process is powered by the Aqualis Protocol asset multi-utilization (AMU) algorithm, which allocates under-utilized stablecoin DEX liquidity in the Aqualis Protocol SP as capital for over-collateralized lending liquidity. Effectively, this means users have access to more capital for loans at lower interest rates, while traders can enjoy lower fees while depositors can earn higher yield farming rewards.
With the integration of AI, the AMU will also be able to support any crypto and token in addition to just stablecoins, allowing Aqualis to build the next generation of DEXs, where all liquidity pool positions are earning even if no trades are happening!

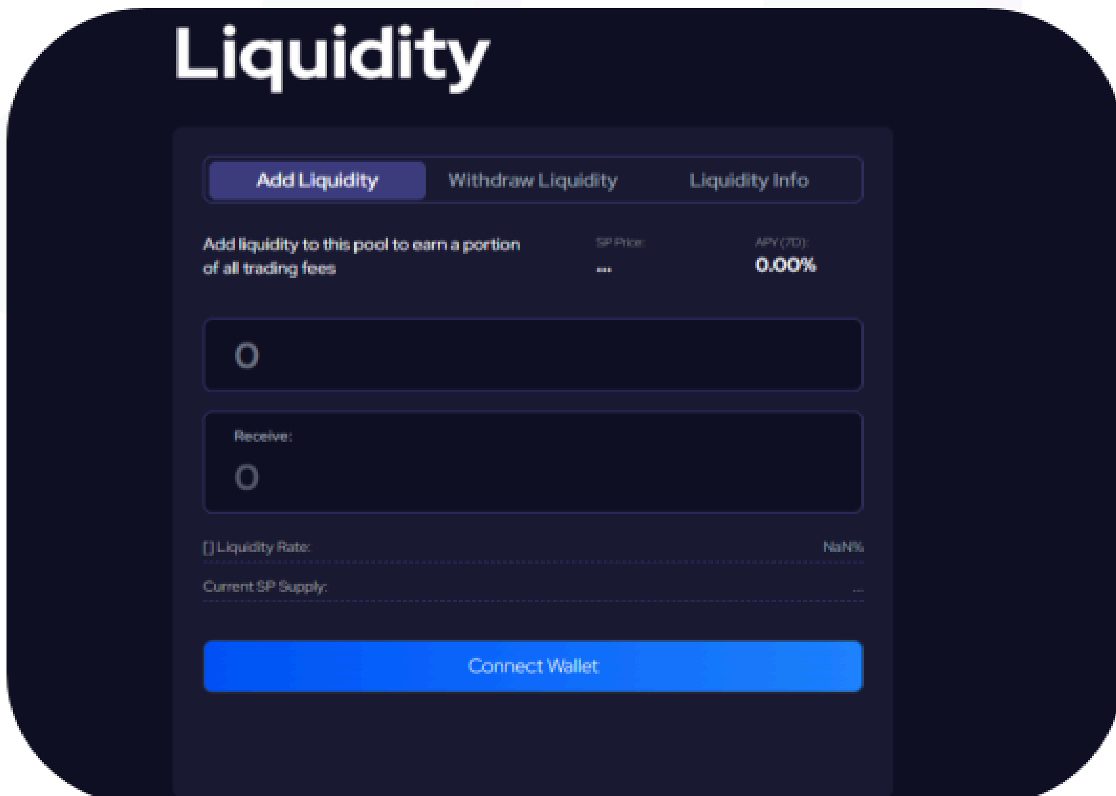
Project Name: Aqualis
Compiler Version: ^0.8.22
Website: www.aqualis.ai
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Aqualis 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. Version 1 of the Aqualis smart contract (AqualisSPv1.sol) is a remediation review of Version 0, or the initial version (AqualisSP.sol), which was initially audited by Hashlock. The updated report provides a comprehensive assessment of the fixes implemented in response to the findings from the AqualisSP.sol (V0) audit.

Description	**Aqualis Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**December, 2024**
Contract 1	AqualisSP.sol (Version 0)
Contract 1 MD5 Hash	8844c538d33aa3f9bdeadf345a765514
Contract 2	AqualisSPv1.sol (Version 1)
Contract 2 MD5 Hash	d8020df0516eaeb2d50f7918d43530d4
GitHub Commit Hash	6b37df5c31a5f56bdf6177e4a5d3dc3844fae02a
Github Commit Hash (v1)	55db5f442102b68f2e641187195525dc9931e1c5

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Hashlocked"**. 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.

Not Secure | Vulnerable | Secure | Hashlocked

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 general security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Aqualis Bug Bounty Program: https://hashlock.com/bug-bounty/aqualis

Hashlock found:

5 High severity vulnerabilities

2 Medium-severity vulnerability

1 Low-severity vulnerability

1 Gas Optimisation

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

AqualisSP.sol (Version 0)
- Allows users to:
  - Swap stablecoins
  - Deposit stablecoins and mint SP tokens
  - Withdraw stablecoins by burning SP tokens
- Allows admin to:
  - Set lending contract address
  - Set AMU thresholds
  - Set AMU rates
- Allows DAOSigners to:
  - Set swap fee components
  - Set liquidity fee components
	Contract achieves this functionality.

AqualisSPv1.sol (Version 1)
- Allows users to:
  - Swap stablecoins
  - Deposit stablecoins and mint SP tokens
  - Withdraw stablecoins by burning SP tokens
- Allows admin to:
  - Set handicap values for tokens
  - Distribute protocol revenue
	Contract achieves this functionality.

Code Quality

This audit scope involves the smart contracts of the Aqualis 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 required.

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 Aqualis 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] AqualisSP#getSharePrice - Incorrect fee calculation will lead to protocol insolvency during low liquidity

Description

The `getSharePrice` function calculates the price of the SP token. Due to fee calculations, during low liquidity periods, this function can return a negative value which will lead to protocol insolvency, bricking the contract.

Vulnerability Details

The `getSharePrice` function calculates the price of the SP token.

```
function getSharePrice() public view returns (int128 sharePrice) {
    uint256 spTokenSupply = SPtoken.totalSupply();
    if (spTokenSupply == 0) return ABDKMath64x64.fromUInt(1);
    sharePrice = totalStablecoinBalance.add(totalSuppliedToLending).div(spTokenSupply.divu(1 ether));
}
```

The share price is dependent on the `totalStablecoinBalance` in the protocol divided by the total supply of SP tokens.

The `totalStablecoinBalance` is calculated with the `_syncStablecoinBalances` function.

```
function _syncStablecoinBalances() internal returns (int128 totalBalance) {
    totalBalance = _getStablecoinTotalBalance().sub(protocolRevenue);
    totalStablecoinBalance = totalBalance;
}
```

This function will return a negative `totalStablecoinBalance` value when the `protocolRevenue` is greater than the total stable coin liquidity in the protocol.

Take a look at how the `protocolRevenue` is calculated:

```
    function _getLiquidityFee(
        address _stablecoin,
        uint256 _amount,
        bool _isExit
    )
        internal
        view
        returns (int128 fee, int128 newLiquidityFixed)
    {
        uint256 decimals = _decimals[_stablecoin];
        uint256 liquidity = IERC20Metadata(_stablecoin).balanceOf(address(this));
        /// @notice Using 18 as decimals on exit, because SPToken as default 18 decimals
        int128 amountFixed = _amount.divu(10 ** (_isExit ? 18 : decimals));
        if (liquidity == 0) return (amountFixed.mul(liquidityBaseFee), amountFixed);

                 (int128  initialLiquidityFixed,  int128  newTotalLiquidityFixed,  int128
_newLiquidityFixed, int128 totalLiquidityFixed) =
            _calculateLiquidityChanges(liquidity, amountFixed, decimals, _isExit);

        int128 targetRatio = _getTargetRatio(_stablecoin);
        fee = _calculateFee(
            targetRatio,
            initialLiquidityFixed,
            totalLiquidityFixed,
            newLiquidityFixed,
            newTotalLiquidityFixed,
            amountFixed,
            liquidityBaseTax,
            liquidityBaseFee
        );
```

```
        newLiquidityFixed = _newLiquidityFixed;
    }
```

We can observe that in the highlighted part, the existing protocol fee is used as available liquidity when calculating the fee. This leads to incorrect fee calculations where it surpasses the amount of total liquidity the protocol has during large transactions that lead to low liquidity in the protocol.

Impact

The contract will be bricked due to the `getSharePrice` function returning a negative value.

Recommendation

Revise the way protocol fees are calculated, do not use the existing fees as available liquidity when calculating the new fees. Take the liquidity ratios in the contract into account when calculating fees.

Status

Resolved

[H-02] AqualisSP#createSP - Incorrect calculation of the amount of SP to be minted leads to minting more tokens as price increases

Description

The `createSP` allows users to mint SP tokens via depositing stable coins into the contract. However, the calculation on how much SP should be minted is incorrect. This leads to users minting more SP tokens as SP price increases.

Vulnerability Details

The `createSP` allows users to mint SP tokens.

```
    function createSP(address _input, uint256 _amount) external watchTVL {
        _validateToken(_input);
        int128 amountFixed = _amount.divu(10 ** _decimals[_input]);
```

```
        (int128 fee, int128 newLiquidity) = getLiquidityFee(_input, _amount, false);
        /// @dev recursive depeg check
        if (isDepeg[_input]) {
            if (!_isDepegOver(_input)) revert Errors.DepegProtection();
        }
        uint256 mintAmount = amountFixed.sub(fee).mul(getSharePrice()).mulu(1 ether);
        _validateAmount(mintAmount);
        IERC20(_input).safeTransferFrom(msg.sender, address(this), _amount);
        SPtoken.mint(msg.sender, mintAmount);
        _accumulateFees(fee);
        /// @dev sync stablecoin balances and run depeg check
        _depegCheck(_input, newLiquidity.div(_syncStablecoinBalances()));
        if (isDepeg[_input]) revert Errors.DepegProtection();
        emit SPCreated(msg.sender, _input, _amount, mintAmount);
    }
```

As observed in the highlighted part, the amount to be minted is calculated by multiplying the liquidity deposit by the share price of the SP token. This leads to more tokens being minted when the SP price is high, diluting the SP pool and causing its price to fall down.

Impact

Users will mint more SP tokens than it is intended when share price is greater than 1 and less tokens that intended when share price is less than 1.

Recommendation

Use division instead of multiply when calculating the amount to be minted.

Status

Resolved

[H-03] AqualisSP#_withdrawOnDepegEvent - The amount of stable coins to be sent to the user does not account for the share price

Description

The `_withdrawOnDepegEvent` function allows users to exchange their SP tokens for stable coins during the event of a stable coin depeg. However, this function does not take the SP tokens price when sending stable coins back to the user.

Vulnerability Details

The `_withdrawOnDepegEvent` function allows users to exchange their SP tokens for stable coins

```
    function _withdrawOnDepegEvent(uint256 SPamount, address to) internal {
        _validateAmount(SPamount);
        int128 total = totalStablecoinBalance.add(totalSuppliedToLending);
        int128 amountFixed = SPamount.divu(1e18);
        address stablecoin;
        uint256 decimals;
        int128 liquidity;
        int128 ratio;
        uint256 amount;
        uint256 transferAmount;
        for (uint256 i = 0; i < N_COINS; i++) {
            stablecoin = _stablecoins.at(i);
            decimals = _decimals[stablecoin];
            liquidity = getLiquidity(stablecoin, decimals);
                        ratio = liquidity.add(suppliedAmounts[stablecoin].divu(10 **
decimals)).div(total);
            amount = ratio.mul(amountFixed).mulu(10 ** decimals);
            (int128 fee,) = getLiquidityFee(stablecoin, amount, true);
            if (amount > liquidity.mulu(10 ** decimals)) {
                uint256 diff = amount - liquidity.mulu(10 ** decimals) + 1;
                _withdrawFromLending(stablecoin, diff);
                suppliedAmounts[stablecoin] -= diff;
```

```
            }
            transferAmount = ratio.mul(amountFixed).sub(fee).mulu(10 ** decimals);
            IERC20(stablecoin).safeTransfer(to, transferAmount);
            _accumulateFees(fee);
            emit SPRedeemed(to, stablecoin, transferAmount, SPamount);
        }
    }
```

As observed in this function, the current share price of the SP token is not taken into account when transferring stable coins to the user. This will lead to users receiving incorrect stable coin amounts.

Impact

The amount of stable coin to be transferred to the users will be incorrect.

Recommendation

Take the share price of the SP token into account when transferring stable coins to the users.

Status

Resolved

[H-04] AqualisSP#_withdrawOnDepegEvent - Function does not burn SP tokens when users are redeeming their SP for stable coins

Description

The `_withdrawOnDepegEvent` function allows users to exchange their SP tokens for stable coins during the event of a stable coin depeg. However, this function does not burn the SP tokens that users use to redeem stable coins.

Vulnerability Details

The `_withdrawOnDepegEvent` function allows users to exchange their SP tokens for stable coins

```
function _withdrawOnDepegEvent(uint256 SPamount, address to) internal {
```

```
        _validateAmount(SPamount);

        int128 total = totalStablecoinBalance.add(totalSuppliedToLending);

        int128 amountFixed = SPamount.divu(1e18);

        address stablecoin;

        uint256 decimals;

        int128 liquidity;

        int128 ratio;

        uint256 amount;

        uint256 transferAmount;

        for (uint256 i = 0; i < N_COINS; i++) {

            stablecoin = _stablecoins.at(i);

            decimals = _decimals[stablecoin];

            liquidity = getLiquidity(stablecoin, decimals);

                          ratio   =  liquidity.add(suppliedAmounts[stablecoin].divu(10  **
decimals)).div(total);

            amount = ratio.mul(amountFixed).mulu(10 ** decimals);

            (int128 fee,) = getLiquidityFee(stablecoin, amount, true);

            if (amount > liquidity.mulu(10 ** decimals)) {

                uint256 diff = amount - liquidity.mulu(10 ** decimals) + 1;

                _withdrawFromLending(stablecoin, diff);

                suppliedAmounts[stablecoin] -= diff;

            }

            transferAmount = ratio.mul(amountFixed).sub(fee).mulu(10 ** decimals);

            IERC20(stablecoin).safeTransfer(to, transferAmount);

            _accumulateFees(fee);

            emit SPRedeemed(to, stablecoin, transferAmount, SPamount);

        }

    }
```

As observed in this function, when users redeem their SP tokens for stable coins, their SP tokens are not burnt.

Impact

Users' SP tokens are not burnt when redeeming them for stable coins. Users can repeatedly call this function to empty out the protocol's stable coin liquidity and steal tokens.

Recommendation

Burn users' SP tokens in the function.

Status

Resolved

[H-05] AqualisSPv1#_watchTVLLogic - Function incorrectly uses handicap adjusted liquidity as the total liquidity in the system resulting in incorrect AMU actions

Description

The `_watchTVLLogic` function executes the AMU action that should be performed depending on the total liquidity and the total amount of tokens supplied to lending. However, this function incorrectly uses handicap adjusted liquidity instead of the raw liquidity amount in the system. This leads to the function receiving inflated values that impact AMU actions.

Vulnerability Details

The `_watchTVLLogic` function executes the AMU action that should be performed

```
function _watchTVLLogic() internal {
    (,, int128 totalLiquidity,) = _getHandicapAdjustedLiquidity();
    (IAqualisConfig.AMUAction action, int128 amount) = configs.checkAMUAction(
        totalLiquidity,
        totalSuppliedToLending
    );

    if (action == IAqualisConfig.AMUAction.DEPOSIT) {
```

```
            _depositToLending(amount);
        } else if (action == IAqualisConfig.AMUAction.WITHDRAW) {
            _AMUWithdrawFromLending(amount);
        } else if (action == IAqualisConfig.AMUAction.WITHDRAW_ALL) {
            _withdrawAllFromLending();
        }

        if (action != IAqualisConfig.AMUAction.NONE) {
            configs.setLastRebalanceBlock(block.number);
        }
    }
```

As observed in the highlighted part, the function uses the handicap adjusted values when deciding on the AMU action that should be taken.

Taking a look at the `_getHandicapAdjustedLiquidity` function

```
    function _getHandicapAdjustedLiquidity() internal view returns (
        int128[] memory,
        int128[] memory,
        int128,
        int128
    ) {
        (uint256[] memory rawBalances, int128[] memory rawRatios, , int128 sharePrice ) =
_getRawStablecoinBalances();

        int128[] memory adjustedLiquidities = new int128[](N_COINS);
        int128 totalLiquidity = ABDKMath64x64.fromUInt(0);
        address stablecoin;

        for (uint256 i = 0; i < N_COINS; i++) {
            if (rawBalances[i] > 0) {
                stablecoin = _stablecoins.at(i);
                      adjustedLiquidities[i] = ABDKMath64x64.divu(rawBalances[i], 10 **
_decimals[stablecoin])
                    .mul(ABDKMath64x64.fromUInt(handicaps[stablecoin]));
```

```
                totalLiquidity = totalLiquidity.add(adjustedLiquidities[i]);
            } else {
                adjustedLiquidities[i] = ABDKMath64x64.fromUInt(0);
            }
        }

        return (adjustedLiquidities, rawRatios, totalLiquidity, sharePrice);
    }
```

It is observed in the highlighted part that this function returns the handicap adjusted values of the token liquidities in the protocol.

Taking a look at how this value is used in the `checkAMUAction` function

```
    function checkAMUAction(
        int128 totalStablecoinBalance,
        int128 totalSuppliedToLending
    ) external view returns (AMUAction action, int128 amount) {
        if (address(lendingPool) == address(0) || block.number - lastRebalanceBlock < 5)
{
            return (AMUAction.NONE, 0);
        }

        int128 TVL = totalStablecoinBalance.add(totalSuppliedToLending);

        // First check if TVL is below minimum threshold
        if (TVL.add(AMUThresholdAmount) < minTVLToEnableAMU) {
            if (totalSuppliedToLending > 0) {
                return (AMUAction.WITHDRAW_ALL, 0);
            }
            return (AMUAction.NONE, 0);
        }

        // Check if TVL is above minimum + threshold
        if (TVL > minTVLToEnableAMU.add(AMUThresholdAmount)) {
```

```
            int128 currRatio = totalSuppliedToLending.div(totalStablecoinBalance);
            int128 diff = ABDKMath64x64.abs(lendingToTradeRatio.sub(currRatio));
            int128 diffAmount = diff.mul(totalStablecoinBalance);

            if (diffAmount > AMUThresholdAmount) {
                if (currRatio < lendingToTradeRatio) {
                    return (AMUAction.DEPOSIT, AMUThresholdAmount);
                } else {
                    return (AMUAction.WITHDRAW, AMUThresholdAmount);
                }
            }
        }

        return (AMUAction.NONE, 0);
    }
```

It is observed that using the handicap adjusted liquidities will lead to incorrect AMU actions to be decided and executed.

Impact

The contract will use incorrect AMU actions.

Recommendation

Use raw liquidities of the tokens in the contract instead of handicap adjusted liquidities.

Status

Resolved

Medium

[M-01] AqualisSPv1#redeemSPOnDepeg - Users are overcharged fees while redeeming SP tokens for stable coins

Description

The `redeemSPOnDepeg` function allows users to redeem their SP tokens for stable coins in the event of a depeg. These actions are charged fees depending on the amount that is being redeemed, however, this function overcharges fees to users attempting to redeem.

Vulnerability Details

The `redeemSPOnDepeg` function allows users to redeem their SP tokens for stable coins in the event of a depeg.

```
function redeemSPOnDepeg(uint256 spAmount) external nonReentrant {
        if (!_isAnyDepeg()) revert DepegProtection();
        _validateAmount(spAmount);
        int128 spAmountFixed = spAmount.divu(1 ether);
              (int128[] memory adjustedLiquidities, , int128 totalLiquidity, ) = _getHandicapAdjustedLiquidity();
        int128 totalSPPlusLending = totalLiquidity.add(totalSuppliedToLending);
        /// @dev Burn SPtoken before iterating transfer
        SPtoken.burn(msg.sender, spAmount);
        address stablecoin;
        uint256 decimals;
        int128 ratio;
        int128 output;
        for (uint256 i = 0; i < N_COINS; i++) {
            stablecoin = _stablecoins.at(i);
            decimals = _decimals[stablecoin];
             ratio = adjustedLiquidities[i].add(suppliedAmounts[stablecoin].divu(10 ** decimals)).div(totalSPPlusLending);
```

```
                    (int128 fee, , , int128 sharePrice) = _getLiquidityFee(stablecoin,
spAmount, true);
            output = ratio.mul(spAmountFixed).mul(sharePrice).sub(fee);
            if (output > adjustedLiquidities[i]) {
                                                    _withdrawFromLending(stablecoin,
ABDKMath64x64.mulu(output.add(fee).sub(adjustedLiquidities[i]), 10 ** decimals));
            }
                    IERC20Metadata(stablecoin).transfer(msg.sender, output.mulu(10 **
decimals));
            _accumulateFees(fee);
                emit SPRedeemed(msg.sender, stablecoin, output.mulu(10 ** decimals),
spAmount);
        }
    }
```

As observed in the highlighted part, the fee is calculated using the full `spAmount` user has inputted for redeeming for each token that the protocol uses. Since there are 3 stable coins used by the protocol at the moment, this will lead to users paying 3 times the fees when redeeming with this function.

Impact

Users will be overcharged fees for their redeems

Recommendation

Correctly calculate the amount of `spAmount` being redeemed for the token in the loop and only charge fees based on this amount.

Status

Resolved

[M-02] AqualisSPv1#redeemSPOnDepeg - Users will receive more stable coins than they should due to the order of execution

Description

The `redeemSPOnDepeg` function allows users to redeem their SP tokens for stable coins in the event of a depeg. This function first burns the SP token that the user supplies back

to the protocol and then transfers the stable coins to the user. However, due to the ordering being incorrect, users will receive more stable coins than they should.

Vulnerability Details

The `redeemSPOnDepeg` function allows users to redeem their SP tokens for stable coins in the event of a depeg.

```
function redeemSPOnDepeg(uint256 spAmount) external nonReentrant {
        if (!_isAnyDepeg()) revert DepegProtection();
        _validateAmount(spAmount);
        int128 spAmountFixed = spAmount.divu(1 ether);
        (int128[] memory adjustedLiquidities, , int128 totalLiquidity, ) = _getHandicapAdjustedLiquidity();
        int128 totalSPPlusLending = totalLiquidity.add(totalSuppliedToLending);
        /// @dev Burn SPtoken before iterating transfer
        SPtoken.burn(msg.sender, spAmount);
        address stablecoin;
        uint256 decimals;
        int128 ratio;
        int128 output;
        for (uint256 i = 0; i < N_COINS; i++) {
            stablecoin = _stablecoins.at(i);
            decimals = _decimals[stablecoin];
            ratio = adjustedLiquidities[i].add(suppliedAmounts[stablecoin].divu(10 ** decimals)).div(totalSPPlusLending);
            (int128 fee, , , int128 sharePrice) = _getLiquidityFee(stablecoin, spAmount, true);
            output = ratio.mul(spAmountFixed).mul(sharePrice).sub(fee);
            if (output > adjustedLiquidities[i]) {
                _withdrawFromLending(stablecoin, ABDKMath64x64.mulu(output.add(fee).sub(adjustedLiquidities[i]), 10 ** decimals));
            }
            IERC20Metadata(stablecoin).transfer(msg.sender, output.mulu(10 ** decimals));
            _accumulateFees(fee);
```

```
                emit SPRedeemed(msg.sender, stablecoin, output.mulu(10 ** decimals), spAmount);
        }
    }
```

As observed in the part highlighted in yellow, function first burns the SP tokens from the user which reduces the SP token total supply. Since the `sharePrice` is dependent on the total supply of SP tokens and the amount of stable coin liquidity in the protocol, burning SP tokens before stable coin transfer will lead to an inflation of the `sharePrice` which leads to users receiving more stable coins than they should.

Impact

Users will be receiving more stable coins than they should

Recommendation

Burn the SP tokens after the transfer of stable coins so that this emergency function favours the protocol.

Status

Resolved

Low

[L-01] AqualisSPv1#redeemSPOnDepeg - Use a `minAmountOut` input for consistency across functions

Description

The `redeemSPOnDepeg` function allows users to redeem stable coins for their SP tokens at the event of a depeg. This function does not utilize the `_minAmountOut` input that is found in the `redeemSP` function.

```
    function redeemSP(address _output, uint256 _amount, uint256 _minAmountOut) external watchTVL {

    function redeemSPOnDepeg(uint256 spAmount) external nonReentrant {
```

Recommendation

Implement a minimum amount out check for this function.

Status

Resolved

Gas

[G-01] AqualisSPv1::_getAMUWithdrawFromLendingRatio - Function makes unnecessary jumps and calculations

Description

The `_getAMUWithdrawFromLendingRatio` function makes a call to the `_getHandicapAdjustedLiquidity` function to receive the `spRatios`. However, the `_getHandicapAdjustedLiquidity` function takes this value directly from the `_getRawStablecoinBalances` function. This leads to making an extra jump and unnecessary calculations that are in the `_getHandicapAdjustedLiquidity` function

Recommendation

Use the `_getRawStablecoinBalances` function instead of the `_getHandicapAdjustedLiquidity` function to receive ratios.

Status

Resolved

Centralisation

The Aqualis 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 Aqualis Protocol project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. 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. 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

hashlock.