hashlock.

Security Audit

Lair Finance (Vault)

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

Table of Contents

Executive Summary	4
Project Context	4
Audit scope	7
Security Rating	10
Intended Smart Contract Functions	11
Code Quality	18
Audit Resources	18
Dependencies	18
Severity Definitions	19
Status Definitions	20
Audit Findings	21
Centralisation	55
Conclusion	56
Our Methodology	57
Disclaimers	59
About Hashlock	60

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

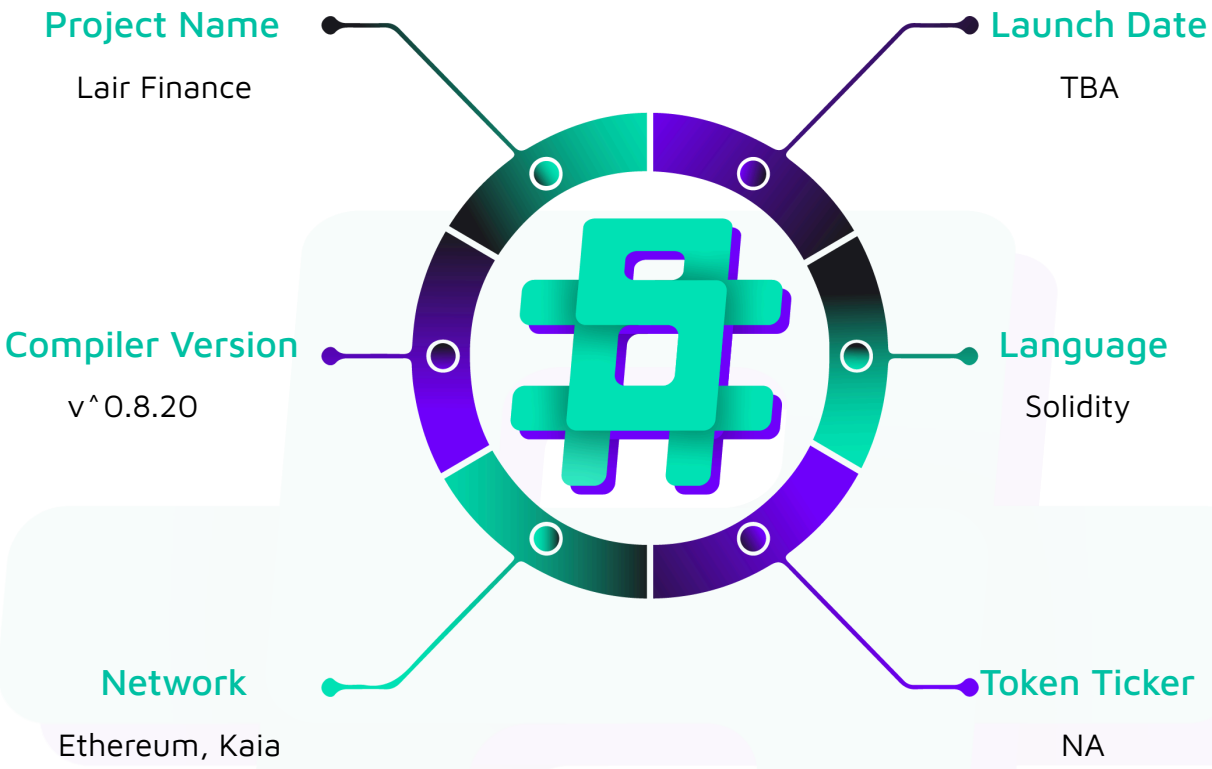
The Lair Finance team partnered with Hashlock to conduct a security audit of their ActiveVaultToken.sol, IActiveVaultToken.sol, IStakingTokenBill.sol, IStakingTokenBillFactory.sol, StakingTokenBill.sol, StakingTokenBillFactory.sol, IVaultTreasury.sol, VaultTreasury.sol, IVaultManager.sol, IVaultManagerHelper.sol, VaultManager.sol, VaultManagerHelper.sol, IVaultService.sol, IVaultServiceFactory.sol, VaultService.sol, VaultServiceFactory.sol, IVaultSwap.sol, VaultSwap.sol, State.sol, Validator.sol, Swap.sol, User.sol,VaultRound.sol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

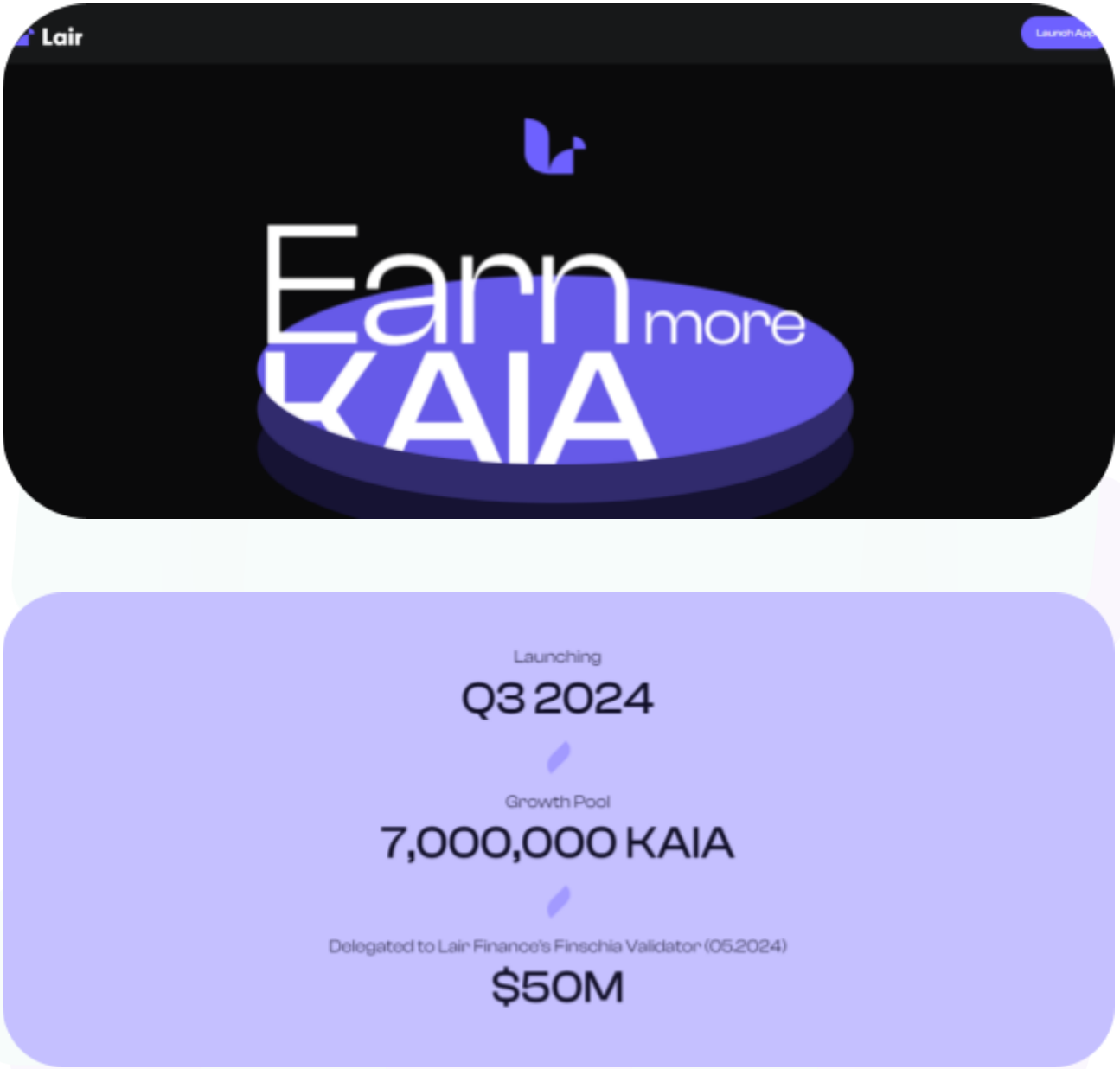
Lair Finance is a cross-chain liquid restaking service with over 600,000 users and 200,000,000+ KAIA delegated. The latest buzz is around liquidity restaking—a new framework designed to supercharge the Kaia ecosystem by building synergy between LINE mini dApps and Lair. This approach amplifies rewards for Lair stakers by incubating new Kaia ecosystem dApps, especially LINE games on the Kaia network. Users can 're'stake LSTs to unlock extra yields from Lair's AVSs.

Project Name: Lair Finance
Project Type: Vault
Compiler Version: ^0.8.20
Website: https://lair.fi/
Logo:

Visualised Context:



Project Visuals:



Audit scope

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

Description	**Lair Finance Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**January, 2025**
Contract 1	State.sol
Contract 1 MD5 Hash	0372eb9ca74d1f0c27ecea0d5904a1e3
Contract 2	Validator.sol
Contract 2 MD5 Hash	57a8cde434e82dbd864644d6a70ce8de
Contract 3	Swap.sol
Contract 3 MD5 Hash	c2f4ccdd5a83b3e54c048eacf8df1930
Contract 4	User.sol
Contract 4 MD5 Hash	943f983d1da88f693ca778a06ecfc925
Contract 5	VaultRound.sol
Contract 5 MD5 Hash	ea2f9be186370005f9af7e044bcab887
Contract 6	ActiveVaultToken.sol
Contract 6 MD5 Hash	7e69a7c41ddc4ae09b78c47e4ad2fcdb
Contract 7	IActiveVaultToken.sol
Contract 7 MD5 Hash	4f1060d24e91d228dc5d4958b837da95
Contract 8	IStakingTokenBill.sol
Contract 8 MD5 Hash	d2787136fe60b8eb46a1416726715e51

Contract 9	IStakingTokenBillFactory.sol
Contract 9 MD5 Hash	e2572a170d7fb2583a0e2c45cc8aea87
Contract 10	StakingTokenBill.sol
Contract 10 MD5 Hash	80fd116666632c1901f3aabd648d2881
Contract 11	StakingTokenBillFactory.sol
Contract 11 MD5 Hash	5c84f95c21b4e76f376aef33cc76aa8f
Contract 12	IVaultTreasury.sol
Contract 12 MD5 Hash	952273b76a56637ba4d5ec84667ae039
Contract 13	VaultTreasury.sol
Contract 13 MD5 Hash	ca6cc3d92e2efc77b570e97bdb257ba3
Contract 14	IVaultManager.sol
Contract 14 MD5 Hash	82101bdac508c43c2ebfc63d2086e7cf
Contract 15	IVaultManagerHelper.sol
Contract 15 MD5 Hash	d00775b4e015caeae4dfc20db71c61df
Contract 16	VaultManager.sol
Contract 16 MD5 Hash	ae97a2235b4b43efc3820c494a746cce
Contract 17	VaultManagerHelper.sol
Contract 17 MD5 Hash	8fbcde57449782d9976ff1c807bb5fd7
Contract 18	IVaultService.sol
Contract 18 MD5 Hash	e41dbe48d95b41b4a5efa42a45fba0a9
Contract 19	IVaultServiceFactory.sol
Contract 19 MD5 Hash	36659005a1268aaa73be98223027c97d
Contract 20	VaultService.sol
Contract 20 MD5 Hash	df734c3f42588bd35a7246913249639f
Contract 21	VaultServiceFactory.sol
Contract 21 MD5 Hash	e5dfafe074dcb4916d800d65563dfd9e

Contract 22	IVaultSwap.sol
Contract 22 MD5 Hash	fc439638c3ab3d22b81ab01afc28bcf9
Contract 23	VaultSwap.sol
Contract 23 MD5 Hash	531956e3f2b5ef92d6490f74a4b9517a

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 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.

Hashlock found:

4 High severity vulnerabilities

3 Medium severity vulnerabilities

5 Low severity vulnerabilities

2 Gas Optimisations

4 QAs

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

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

Validator.sol
- Allows developers to:
  - Use modifiers to validate addresses are not zero
  - Use modifiers to check if values are greater than zero
  - Use modifiers to check if values are greater than or equal to zero
  - Use a modifier to validate vault round states
  - Use a modifier to validate round indices
- Provides internal functions to:
  - Check if an address is not zero
  - Check if a value is greater than zero
  - Check if a value is greater than or equal to zero
- Defines constants:
  - ZERO_ADDRESS as the zero address (0x0)
	Contract achieves this functionality.

ActiveVaultToken.sol
- Allows users to:
  - Approve token spending
  - Transfer tokens
  - Transfer tokens from one address to another
- Allows admins to:
  - Add and remove roles
  - Mint new tokens
	Contract achieves this functionality.

- Burn existing tokens - Pause and unpause the contract - Transfer ownership of the contract - Set the active vault service address - Implements access control: - VAULT_SERVICE role for minting and burning - VAULT_PAUSE role for pausing/unpausing - BLOCK role to prevent certain addresses from transferring tokens - DEFAULT_ADMIN_ROLE for administrative functions - Provides modifiers: - checkBlockRole to prevent blocked addresses from transferring tokens - Overrides standard functions: - Disables renouncing ownership - Disables revoking roles directly - Modifies transferOwnership to require DEFAULT_ADMIN_ROLE - Implements pausable functionality: - All token transfers and approvals can be paused/unpaused	
StakingTokenBill.sol - Allows users to: - Approve token spending - Transfer tokens - Transfer tokens from one address to another - Allows admins to: - Add and remove roles - Mint new tokens	**Contract achieves this functionality.**

- Burn existing tokens - Pause and unpause the contract - Transfer ownership of the contract - Set the active vault manager address - Implements access control: - VAULT_MANAGER role for minting and burning - VAULT_PAUSE role for pausing/unpausing - BLOCK role to prevent certain addresses from transferring tokens - DEFAULT_ADMIN_ROLE for administrative functions - Provides modifiers: - checkBlockRole to prevent blocked addresses from transferring tokens - Overrides standard functions: - Disables renouncing ownership - Disables revoking roles directly - Modifies transferOwnership to require DEFAULT_ADMIN_ROLE - Implements pausable functionality: - All token transfers and approvals can be paused/unpaused - Additional features: - Stores and allows retrieval of a round index - Allows retrieval of the active vault manager address	
StakingTokenBillFactory.sol - Allows users to: - Create new StakingTokenBill contracts - Provides functions to:	**Contract achieves this functionality.**

- Deploy a new StakingTokenBill contract with specified parameters	

VaultTreasury.sol
- Allows admins to:
  - Set the vault manager address
  - Set the restaking token address
- Add and remove roles
  - Pause and unpause the contract
- Allows the vault manager to:
  - Withdraw restaking tokens to a specified receiver
- Implements access control:
  - DEFAULT_ADMIN_ROLE for administrative functions
  - VAULT_MANAGER_PAUSE role for pausing/unpausing
  - onlyVaultManager modifier for withdrawal function
- Provides functions to:
  - Get the vault manager address
  - Get the restaking token address
- Implements safety measures:
  - Uses ReentrancyGuard for the withdraw function
  - Uses SafeERC20 for token transfers
  - Implements pausable functionality
- Overrides standard functions:
  - Disables renouncing ownership
  - Disables transferring ownership
  - Disables revoking roles directly
  - Disables renouncing roles
- Additional features:
	Contract achieves this functionality.

- Can receive ETH (has a receive function) - Emits a Withdraw event when tokens are withdrawn	

VaultManager.sol
- Allows users to:
  - Stake tokens into active vaults
  - Unstake tokens from active vaults
  - Claim rewards from active vaults
  - Claim all rewards across multiple rounds and vaults
- Allows admins to:
  - Set and update round information
  - Set and update vault information for each round
  - Start reward distribution for vaults in a round
  - Set staking token bill receive addresses
  - Pause and unpause the contract
  - Add and remove roles
- Implements access control:
  - VAULT_PAUSE role for pausing/unpausing
  - VAULT_MANAGER role for administrative functions
  - DEFAULT_ADMIN_ROLE for high-level administrative functions
- Provides functions to:
  - Get information about rounds, vaults, and user staking amounts
  - Calculate and update rewards
  - Manage round states and vault states
  - Implements safety measures:
  - Uses ReentrancyGuard for critical
	Contract achieves this functionality.

functions - Implements pausable functionality - Uses SafeMath for calculations - Uses SafeERC20 for token transfers - Additional features: - Supports multiple rounds of staking - Supports multiple vaults per round - Calculates rewards based on staking duration and amount - Integrates with external contracts (VaultService, VaultTreasury, StakingTokenBill)	
VaultManagerHelper.sol - Allows users to: - Stake tokens - Unstake tokens - Claim rewards - Allows admins (VAULT_MANAGER role) to: - Set the vault manager address - Set the name and symbol of the vault - Set the total reward amount - Start the reward distribution - Set the staking token bill receive address - Implements access control: - VAULT_MANAGER role for administrative functions - VAULT_PAUSE role (unused in this contract) - DEFAULT_ADMIN_ROLE for high-level administrative functions - Provides functions to: - Get various information about the vault	**Contract achieves this functionality.**

(total amounts, user staking amounts, rewards, etc.) - Calculate and update rewards - Manage staking and unstaking operations	
VaultServiceFactory.sol - Allows users to: - Create new VaultService contracts - Provides functions to: - Deploy a new VaultService contract with specified parameters	**Contract achieves this functionality.**
VaultSwap.sol - Allows users to: - Swap tokens based on predefined pairs - Perform multiple swaps in a single transaction - Allows admins (VAULT_SWAP_MANAGER role) to: - Add new swap pairs - Update existing swap pairs - Withdraw remaining tokens after expiry - Implements access control: - VAULT_SWAP_MANAGER role for administrative functions - VAULT_SWAP_PAUSE role for pausing/unpausing the contract - DEFAULT_ADMIN_ROLE for high-level administrative functions - Provides functions to: - Get the list of all swap pairs - Get a paginated list of swap pairs - Perform swaps - Withdraw tokens	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Lair Finance 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 Lair Finance 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] ActiveVaultToken,StakingTokenBill&VaultTreasury - Inconsistent and Insecure Role Management

Description

The `ActiveVaultToken`, `StakingTokenBill`, and `VaultTreasury` contracts exhibit several critical issues in their role management systems. These issues could lead to security vulnerabilities, privilege escalation, or denial of service. The main problems are outlined below:

1. Inconsistent Role Removal Protection: The `removeRole` function in both the `ActiveVaultToken` and `StakingTokenBill` contracts allows an admin to remove critical roles from other accounts, including other admins. This could lead to centralization or loss of control.

Example from `StakingTokenBill`:

```
function removeRole(bytes32 role, address account) public onlyRole(DEFAULT_ADMIN_ROLE) {
    require(
        !(hasRole(DEFAULT_ADMIN_ROLE, account) && account == msg.sender),
        "Admin cannot revoke own admin role"
    );
    _revokeRole(role, account);
}

function revokeRole(bytes32 role, address) public override onlyRole(getRoleAdmin(role)) {
    revert("ActiveVaultToken:: revokeRole is disabled");
```

```
}
```

2. Conflicting Role Management: The `VaultTreasury` contract disables the `revokeRole` function:

```
function revokeRole(bytes32 role, address) public override onlyRole(getRoleAdmin(role)) {
    revert("ActiveVaultToken:: revokeRole is disabled");
}
```

This inconsistency with the `removeRole` function confuses how roles are managed within the contract.

3. Centralization Risk: An admin can remove critical roles like `VAULT_MANAGER` or `VAULT_SERVICE`, which could break core functionality. If an admin's account is compromised, this could lead to a complete takeover of the contract.
4. Missing Role Validation: There are no checks to prevent the removal of critical system roles or to ensure a minimum number of role holders.

Recommendation

1. Implement Critical Role Protection: Modify the `removeRole` function to include checks that prevent the removal of critical roles:

```
function removeRole(bytes32 role, address account) public onlyRole(DEFAULT_ADMIN_ROLE) {
    require(role != VAULT_SERVICE, "Cannot remove VAULT_SERVICE role");
    require(getRoleMemberCount(DEFAULT_ADMIN_ROLE) > 1, "Cannot remove last admin");
    require(
        !(hasRole(DEFAULT_ADMIN_ROLE, account) && account == msg.sender),
        "Admin cannot revoke own admin role"
    );
    _revokeRole(role, account);
    emit RoleRemoved(role, account);
}
```

2. Consider Additional Security Measures:
    a. Implement a two-step role removal process.
    b. Introduce a timelock for critical role changes.
    c. Set limits on role removals.
    d. Add an emergency role recovery mechanism.
3. Resolve Inconsistency: Ensure consistency between the `removeRole` and `revokeRole` functions across both contracts to avoid confusion and potential misuse.

Status

Resolved

[H-02] VaultManagerHelper#setVaultManagerAddress - Critical Access Control Vulnerability

Description

The `VaultManagerHelper` contract contains a critical access control vulnerability in the `setVaultManagerAddress` function. This function is currently declared as external, allowing any address to call it and change the `vaultManagerAddress`. This poses a severe security risk as it could allow an attacker to point the helper contract to a malicious `VaultManager`, potentially disrupting the entire system's functionality and compromising user funds.

The vulnerable code is as follows:

```
function setVaultManagerAddress(address _vaultManagerAddress) external {

    vaultManagerAddress = _vaultManagerAddress;

}
```

Impact:

An attacker could exploit this vulnerability to point the helper contract to a malicious `VaultManager`, leading to incorrect data retrieval and potential fund loss. Additionally, they could disrupt the system's functionality by setting an invalid address. The attacker

might also manipulate the contract's behavior to their advantage. This vulnerability effectively undermines the trustworthiness and security of the entire `VaultManagerHelper` contract.

Recommendation

Implement proper access control for the `setVaultManagerAddress` function. This can be achieved by:

Using OpenZeppelin's Ownable contract and the `onlyOwner` modifier:

```
import "@openzeppelin/contracts/access/Ownable.sol";


contract VaultManagerHelper is Ownable {
    // ...

    function setVaultManagerAddress(address _vaultManagerAddress) external onlyOwner {
        vaultManagerAddress = _vaultManagerAddress;
    }

    // ...
}
```

Alternatively, implement a custom access control mechanism:

```
address private owner;

constructor(address _vaultManagerAddress) {
    vaultManagerAddress = _vaultManagerAddress;
    owner = msg.sender;
}

modifier onlyOwner() {
    require(msg.sender == owner, "Caller is not the owner");
```

```
    _;
}
function setVaultManagerAddress(address _vaultManagerAddress) external onlyOwner {
    vaultManagerAddress = _vaultManagerAddress;
}
```

Consider implementing a time-lock or multi-signature requirement for sensitive operations like changing the `VaultManager` address.

Emit an event when the `VaultManager` address is changed for better transparency:

```
event VaultManagerAddressChanged(address indexed oldAddress, address indexed newAddress);
function setVaultManagerAddress(address _vaultManagerAddress) external onlyOwner {
    address oldAddress = vaultManagerAddress;
    vaultManagerAddress = _vaultManagerAddress;
    emit VaultManagerAddressChanged(oldAddress, _vaultManagerAddress);
}
```

Status

Resolved

[H-03] VaultManagerHelper#getVaultInfos - Critical Indexing Bug

Description

The `getVaultInfos` function in the `VaultManagerHelper` contract contains a critical indexing bug that leads to incomplete and incorrect data being returned. The function aims to aggregate vault information across all rounds, but due to an indexing error in the nested loop, it only retains information from the last processed round.

The problematic code is as follows:

```
function getVaultInfos() public view returns (VaultRound.VaultInfo[] memory) {
    // ... (vaultCount calculation)
```

```
    VaultRound.VaultInfo[] memory vaultInfoList = new VaultRound.VaultInfo[](vaultCount);


       for (uint256 roundIndex = 0; roundIndex < getVaultManager().getRoundCount();
roundIndex++) {
        VaultRound.VaultRoundInfoView memory roundInfo = roundInfos[roundIndex];
            for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length;
vaultIndex++) {
            vaultInfoList[vaultIndex] = roundInfo.vaultInfoList[vaultIndex]; // Bug here
        }
    }


    return vaultInfoList;
}
```

Impact:

The function suffers from several critical issues. Data loss is a significant concern, as information from all rounds except the last one is overwritten and lost. This leads to incomplete data, with the function returning only vault information from the last processed round. There's also a potential for array access violation if the last round has fewer vaults than earlier rounds, leaving parts of the array uninitialized. These problems culminate in misleading results, where users or dependent contracts relying on this function will receive incorrect and incomplete data. This inaccuracy could potentially lead to incorrect decisions or calculations, compromising the integrity and reliability of the entire system.

Recommendation

Implement a cumulative index to correctly populate the vaultInfoList array across all rounds:

```
function getVaultInfos() public view returns (VaultRound.VaultInfo[] memory) {
```

```
                        VaultRound.VaultRoundInfoView[]      memory      roundInfos      =
getVaultManager().getVaultRoundList();

    uint256 vaultCount = 0;


    for (uint256 roundIndex = 0; roundIndex < roundInfos.length; roundIndex++) {

        vaultCount += roundInfos[roundIndex].vaultInfoList.length;

    }


    VaultRound.VaultInfo[] memory vaultInfoList = new VaultRound.VaultInfo[](vaultCount);


    uint256 cumulativeIndex = 0;

    for (uint256 roundIndex = 0; roundIndex < roundInfos.length; roundIndex++) {

        VaultRound.VaultRoundInfoView memory roundInfo = roundInfos[roundIndex];

            for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length;
vaultIndex++) {

            vaultInfoList[cumulativeIndex] = roundInfo.vaultInfoList[vaultIndex];

            cumulativeIndex++;

        }

    }


    return vaultInfoList;

}
```

Status

Resolved

[H-04] VaultSwap#_withdraw - Incorrect State Management

Description

The `_withdraw` function in the `VaultSwap` contract contains a critical bug where it fails to update the `remainSwapTokenAmount` after transferring tokens. This oversight can lead to a significant discrepancy between the actual token balance and the recorded state in the contract.

The problematic code is as follows:

```
function _withdraw(uint256 pairIndex, address receiveAddress) private {

      require(pairIndex < vaultSwapPairList.length, "VaultSwap::withdraw: invalid pair index");


    VaultRound.VaultSwapPair memory vaultSwapPair = vaultSwapPairList[pairIndex];


        require(vaultSwapPair.expiredTime < block.timestamp, "VaultSwap::withdraw: not expired");

     require(vaultSwapPair.remainSwapTokenAmount > 0, "VaultSwap::withdraw: insufficient total withdraw amount");


                    IERC20(vaultSwapPair.swapTokenAddress).safeTransfer(receiveAddress, vaultSwapPair.remainSwapTokenAmount);


           emit Withdraw(pairIndex, receiveAddress, vaultSwapPair.swapTokenAddress, vaultSwapPair.remainSwapTokenAmount);

}
```

Impact:

The contract suffers from a critical flaw that results in an incorrect state, where the `remainSwapTokenAmount` in the contract state does not accurately reflect the actual remaining balance after a withdrawal. This discrepancy leads to a serious vulnerability

allowing for multiple withdrawals of the same tokens, potentially enabling malicious actors to drain the contract of more tokens than intended. Consequently, the contract's state becomes inconsistent with its actual token balance, which can lead to cascading issues in other functions that rely on this state for their operations. This inconsistency undermines the contract's integrity and reliability, potentially causing significant financial losses and operational disruptions.

Recommendation

Update the `remainSwapTokenAmount` after the token transfer:

```
function _withdraw(uint256 pairIndex, address receiveAddress) private {

      require(pairIndex < vaultSwapPairList.length, "VaultSwap::withdraw: invalid pair index");

    VaultRound.VaultSwapPair storage vaultSwapPair = vaultSwapPairList[pairIndex];

       require(vaultSwapPair.expiredTime < block.timestamp, "VaultSwap::withdraw: not expired");

     require(vaultSwapPair.remainSwapTokenAmount > 0, "VaultSwap::withdraw: insufficient total withdraw amount");

    uint256 amountToWithdraw = vaultSwapPair.remainSwapTokenAmount;

                    IERC20(vaultSwapPair.swapTokenAddress).safeTransfer(receiveAddress, amountToWithdraw);

    vaultSwapPair.remainSwapTokenAmount = 0;

           emit Withdraw(pairIndex, receiveAddress, vaultSwapPair.swapTokenAddress, amountToWithdraw);

}
```

Status

Resolved

Medium

[M-01] VaultManager#claimAll - Potential Denial of Service (DoS) Vulnerability

Description

The `claimAll` function in the `VaultManager` contract contains a potential Denial of Service (DoS) vulnerability. This function iterates over all rounds and all vaults within each round to process claims for the caller. As the number of rounds and vaults increases, the gas cost of this function will grow linearly, potentially exceeding the block gas limit and causing the transaction to fail.

The problematic code is as follows:

```
function claimAll() public nonReentrant {

    for (uint256 roundIndex = 0; roundIndex < getRoundCount(); roundIndex++) {

        VaultRound.VaultRoundInfo storage roundInfo = _getRound(roundIndex);

            for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length;
vaultIndex++) {

            VaultRound.VaultRoundInfo storage roundInfo = _getRound(roundIndex);


            if (roundInfo.state == State.VaultRoundState.None) {

                continue;

            }


                                                address    activeVaultAddress    =
roundInfo.vaultInfoList[vaultIndex].activeVaultServiceAddress;

            _claim(roundIndex, activeVaultAddress, _msgSender());

        }

    }

}
```

This implementation could lead to several significant issues that compromise the contract's functionality and security. Transactions may fail due to out-of-gas errors if there are many rounds or vaults, rendering the contract unusable in certain scenarios. Users might find themselves unable to claim their rightful rewards if the gas cost exceeds the block gas limit, effectively locking their assets. Furthermore, this implementation increases the contract's vulnerability to gas price manipulation attacks, potentially allowing malicious actors to exploit the system for their benefit. These problems collectively undermine the contract's reliability and user experience, potentially leading to financial losses and eroded trust in the platform.

Recommendation

To mitigate this issue, consider implementing one or more of the following solutions:

Implement a batched claim function that allows claiming rewards for a specific range of rounds or vaults:

```
function claimBatch(uint256 startRound, uint256 endRound) public nonReentrant {
    require(startRound <= endRound && endRound < getRoundCount(), "Invalid round range");
    for (uint256 roundIndex = startRound; roundIndex <= endRound; roundIndex++) {
        // Claim logic here
    }
}
```

Implement a per-round or per-vault claim function:

```
function claimForRound(uint256 roundIndex) public nonReentrant {
    require(roundIndex < getRoundCount(), "Invalid round index");
    // Claim logic for specific round
}
```

Use a pull-based reward system instead of the current push-based system, allowing users to claim rewards for each vault individually.

Implement a gas-efficient iteration mechanism that tracks the last processed round and vault, allowing users to claim rewards incrementally across multiple transactions.

Status

Resolved

[M-02] VaultManagerHelper#getVaultInfos - Potential Array Out-of-Bounds Access

Description

The `getVaultInfos` function in the `VaultManagerHelper` contract contains a potential array of out-of-bounds access vulnerabilities. In the nested loop that populates the vaultInfoList array, the function uses the vaultIndex from the inner loop to assign values to vaultInfoList without ensuring that this index is within the bounds of the array.

The problematic code is as follows:

```
for (uint256 roundIndex = 0; roundIndex < getVaultManager().getRoundCount(); roundIndex++) {
    VaultRound.VaultRoundInfoView memory roundInfo = roundInfos[roundIndex];
        for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length; vaultIndex++) {
        vaultInfoList[vaultIndex] = roundInfo.vaultInfoList[vaultIndex];
    }
}
```

This implementation assumes that the `vaultIndex` will always be less than the total `vaultCount` calculated earlier. However, if the total number of vaults across all rounds exceeds `vaultCount`, this could lead to an out-of-bounds array access, potentially causing the transaction to revert or, in worse cases, corrupting memory.

Recommendation

To fix this issue, use a separate counter for the `vaultInfoList` array. Here's an example of how to modify the code:

```
uint256 totalVaultIndex = 0;

for (uint256 roundIndex = 0; roundIndex < getVaultManager().getRoundCount(); roundIndex++) {

    VaultRound.VaultRoundInfoView memory roundInfo = roundInfos[roundIndex];

        for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length; vaultIndex++) {

        if (totalVaultIndex < vaultCount) {

            vaultInfoList[totalVaultIndex] = roundInfo.vaultInfoList[vaultIndex];

            totalVaultIndex++;

        } else {

            // Optional: Log an error or emit an event if we exceed vaultCount

            break;

        }

    }

    if (totalVaultIndex >= vaultCount) {

        break;

    }

}
```

Status

Resolved

[M-03] VaultSwap#swap - Front-Running Vulnerability

Description

The `swap` function in the `VaultSwap` contract is vulnerable to front-running attacks. This vulnerability allows malicious actors to observe pending swap transactions in the `mempool` and place their own transactions with higher gas prices, potentially manipulating the swap outcome to their advantage.

The vulnerable code is in the swap function:

```
function swap(uint256 pairIndex, uint256 amount) public whenNotPaused nonReentrant {

    _swap(_msgSender(), pairIndex, amount);

    emit SwapEvent(_msgSender(), pairIndex, amount);

}
```

Recommendation

Implement a commit-reveal scheme or use a decentralized oracle for pricing to mitigate front-running:

Several strategies can be implemented to mitigate front-running vulnerabilities. A commit-reveal scheme can be employed, where users first submit a hash of their intended swap parameters in the commit phase, and after a set period, they reveal their parameters and execute the swap. This approach makes it more challenging for attackers to front-run as they don't know the exact swap details in advance. Another effective strategy is to use a decentralized oracle, such as Chainlink, to obtain fair market prices. By executing swaps based on the Oracle price rather than the current contract state, the system becomes more resistant to manipulation. Additionally, implementing slippage protection allows users to specify the minimum amount of tokens they expect to receive, with the transaction reverting if the actual output falls below this minimum. These combined measures significantly enhance the contract's resistance to front-running attacks and improve overall fairness and security for users.

```
function swap(uint256 pairIndex, uint256 amount, uint256 minAmountOut) public whenNotPaused nonReentrant {
    (uint256 depositAmount, uint256 swapAmount) = _swap(_msgSender(), pairIndex, amount);
    require(swapAmount >= minAmountOut, "VaultSwap: Slippage exceeded");
    emit SwapEvent(_msgSender(), pairIndex, amount);
}
```

Status

Resolved

Low

[L-01] VaultManager#setRound - Insufficient Input Validation

Description

The `setRound()` function in the `VaultManager` contract lacks crucial input validations, potentially leading to the creation of invalid or problematic vault rounds. This oversight could result in unexpected behavior, including the creation of rounds that start in the past, have zero rewards, or last for unreasonably long periods.

The function is missing the following crucial validations:

`startTime` should be greater than the current block timestamp to prevent creating rounds that start in the past.

`totalStakingTokenAmount` should be greater than zero to ensure there are rewards for the round.

There's no maximum duration check, potentially allowing for extremely long-running rounds.

Recommendation

Implement the following additional checks in the `setRound()` function:

```
function setRound(uint256 roundIndex, uint256 startTime, uint256 endTime, State.VaultRoundState state, uint256 totalStakingTokenAmount)

public

nonReentrant

onlyRole(VAULT_MANAGER)

validMoreThanEqualZero(roundIndex)

validMoreThanEqualZero(startTime)

validMoreThanEqualZero(endTime)

checkState(state)
```

```
{
    uint256 nextRound = getRoundCount().add(1);
    require(roundIndex <= nextRound, "must be less than or equal nextRound");
    require(startTime <= endTime, "endTime is always bigger or equal startTime");

    // New validations
    require(startTime > block.timestamp, "Start time must be in the future");
    require(totalStakingTokenAmount > 0, "Total staking token amount must be greater than zero");
    require(endTime.sub(startTime) <= MAX_ROUND_DURATION, "Round duration exceeds maximum allowed");

    // ... rest of the function
}
```

Also, consider adding a constant for `MAX_ROUND_DURATION` at the contract level:

```
uint256 private constant MAX_ROUND_DURATION = 365 days; // Or any other appropriate maximum duration
```

Status

Resolved

[L-02] VaultManager#claimAll - Redundant and Potentially Misleading Variable Declaration

Description

The `claimAll()` function in the `VaultManager` contract contains a redundant and potentially misleading variable declaration. The roundInfo variable is declared twice within nested loops, which is unnecessary and could lead to confusion or subtle bugs.

The problematic code is as follows:

```
function claimAll() public nonReentrant {

    for (uint256 roundIndex = 0; roundIndex < getRoundCount(); roundIndex++) {

        VaultRound.VaultRoundInfo storage roundInfo = _getRound(roundIndex);

            for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length;
vaultIndex++) {

               VaultRound.VaultRoundInfo storage roundInfo = _getRound(roundIndex);   //
Redundant declaration


            if (roundInfo.state == State.VaultRoundState.None) {

                continue;

            }


                                                address    activeVaultAddress    =
roundInfo.vaultInfoList[vaultIndex].activeVaultServiceAddress;

            _claim(roundIndex, activeVaultAddress, _msgSender());

        }

    }

}
```

The issue is that roundInfo is declared twice:

- Once in the outer loop
- Again in the inner loop

This redundant declaration is unnecessary and potentially confusing. It doesn't cause immediate issues because both declarations refer to the same storage location, but it's a poor coding practice that could lead to maintenance problems or subtle bugs if the code is modified in the future.

Recommendation

Remove the redundant declaration in the inner loop. The function should be refactored as follows:

```
function claimAll() public nonReentrant {

    for (uint256 roundIndex = 0; roundIndex < getRoundCount(); roundIndex++) {

        VaultRound.VaultRoundInfo storage roundInfo = _getRound(roundIndex);

        if (roundInfo.state == State.VaultRoundState.None) {

            continue;

        }

            for (uint256 vaultIndex = 0; vaultIndex < roundInfo.vaultInfoList.length;
vaultIndex++) {

                                                address    activeVaultAddress    =
roundInfo.vaultInfoList[vaultIndex].activeVaultServiceAddress;

            _claim(roundIndex, activeVaultAddress, _msgSender());

        }

    }

}
```

Status

Resolved

[L-03] VaultSwap#addPair - Insufficient Input Validation

Description

The `addPair` function in the `VaultSwap` contract lacks a crucial input validation check. Specifically, it does not verify that the deposit token address (`_depositTokenAddress`) and the swap token address (`_swapTokenAddress`) are different. This oversight could lead to the creation of invalid or nonsensical swap pairs, potentially disrupting the intended functionality of the contract.

The vulnerable code is as follows:

```
function addPair(

    address _depositTokenAddress,
```

```
    address _swapTokenAddress,
    uint256 _swapRate,
    uint256 _totalSwapTokenAmount,
    uint256 _expiredTime
) external
    onlyRole(VAULT_SWAP_MANAGER)
    nonReentrant
    validAddress(_depositTokenAddress)
    validAddress(_swapTokenAddress)
    validMoreThanZero(_swapRate)
    validMoreThanZero(_totalSwapTokenAmount)
{
    vaultSwapPairList.push(
        VaultRound.VaultSwapPair({
            index: vaultSwapPairList.length,
            depositTokenAddress: _depositTokenAddress,
            swapTokenAddress: _swapTokenAddress,
            // ... other fields ...
        })
    );
}
```

Recommendation

Add a check to ensure that the deposit token address and swap token address are different:

```
function addPair(
    address _depositTokenAddress,
```

```
    address _swapTokenAddress,
    uint256 _swapRate,
    uint256 _totalSwapTokenAmount,
    uint256 _expiredTime
) external
    onlyRole(VAULT_SWAP_MANAGER)
    nonReentrant
    validAddress(_depositTokenAddress)
    validAddress(_swapTokenAddress)
    validMoreThanZero(_swapRate)
    validMoreThanZero(_totalSwapTokenAmount)
{
       require(_depositTokenAddress != _swapTokenAddress, "VaultSwap: Deposit and swap
tokens must be different");

    vaultSwapPairList.push(
        VaultRound.VaultSwapPair({
            index: vaultSwapPairList.length,
            depositTokenAddress: _depositTokenAddress,
            swapTokenAddress: _swapTokenAddress,
            // ... other fields ...
        })
    );
}
```

Status

Resolved

[L-04] VaultSwap#addPair - Missing Expiration Time Validation

Description

The `addPair` function in the `VaultSwap` contract lacks crucial input validation for the `_expiredTime` parameter. This oversight allows the creation of swap pairs with expiration times set in the past or the immediate future, potentially leading to pairs that are immediately expired or have an unreasonably short active period.

The vulnerable code is as follows:

```
function addPair(
    address _depositTokenAddress,
    address _swapTokenAddress,
    uint256 _swapRate,
    uint256 _totalSwapTokenAmount,
    uint256 _expiredTime
) external
    onlyRole(VAULT_SWAP_MANAGER)
    nonReentrant
    validAddress(_depositTokenAddress)
    validAddress(_swapTokenAddress)
    validMoreThanZero(_swapRate)
    validMoreThanZero(_totalSwapTokenAmount)
{
    vaultSwapPairList.push(
        VaultRound.VaultSwapPair({
            // ... other fields ...
            expiredTime: _expiredTime,
            // ... remaining fields ...
```

```
        })
    );
}
```

Recommendation

Implement proper validation for the `_expiredTime` parameter:

```
function addPair(
    address _depositTokenAddress,
    address _swapTokenAddress,
    uint256 _swapRate,
    uint256 _totalSwapTokenAmount,
    uint256 _expiredTime
) external
    onlyRole(VAULT_SWAP_MANAGER)
    nonReentrant
    validAddress(_depositTokenAddress)
    validAddress(_swapTokenAddress)
    validMoreThanZero(_swapRate)
    validMoreThanZero(_totalSwapTokenAmount)
{
     require(_expiredTime > block.timestamp, "VaultSwap: Expiration time must be in the future");
     require(_expiredTime - block.timestamp >= 1 hours, "VaultSwap: Expiration time must be at least 1 hour in the future");

    vaultSwapPairList.push(
        VaultRound.VaultSwapPair({
```

```
            // ... other fields ...
            expiredTime: _expiredTime,
            // ... remaining fields ...
        })
    );
}
```

Status

Resolved

[L-05] VaultSwap#renounceOwnership - Incorrect Error Message

Description

The `renounceOwnership` function in the `VaultSwap` contract contains an incorrect error message. The error message refers to "`VaultManager`" instead of "`VaultSwap`", which is the actual name of the contract.

The problematic code is as follows:

```
function renounceOwnership() public override onlyOwner {
    revert("VaultManager:: ownership cannot be renounced");
}
```

Recommendation

Correct the error message to accurately reflect the contract name:

```
function renounceOwnership() public override onlyOwner {
    revert("VaultSwap:: ownership cannot be renounced");
}
```

Status

Resolved

Gas

[G-01] Validator#ZERO_ADDRESS - Unnecessary Public Constant Declaration

Description

In the Validator contract, the `ZERO_ADDRESS` constant is declared as public:

```
address public constant ZERO_ADDRESS = address(0);
```

This public declaration automatically generates a getter function for the constant. If this constant is only used internally within the contract or its inheriting contracts, public visibility is unnecessary. This can lead to slightly increased gas costs during contract deployment and potentially expose internal implementation details.

Recommendation

Change the visibility of the `ZERO_ADDRESS` constant to internal or private, depending on its intended use

Status

Resolved

[G-02] VaultManagerHelper#getVaultInfoBySymbol - Inefficient Memory Usage

Description

The `getVaultInfoBySymbol` function in the `VaultManagerHelper` contract exhibits inefficient memory usage. It creates a temporary array `tempInfoList` with the maximum possible size (equal to the total number of rounds), regardless of how many rounds actually match the given symbol. This approach can be wasteful, especially when there are many rounds but only a few matches.

The problematic code is as follows:

```
function getVaultInfoBySymbol(string memory symbol) public view returns (VaultRound.VaultRoundInfoView[] memory) {
```

```
                VaultRound.VaultRoundInfoView[] memory tempInfoList = new
VaultRound.VaultRoundInfoView[](getVaultManager().getRoundCount());
    uint256 count = 0;

    // ... (loop to populate tempInfoList)

                VaultRound.VaultRoundInfoView[] memory vaultInfoList = new
VaultRound.VaultRoundInfoView[](count);
    for (uint256 i = 0; i < count; i++) {
        vaultInfoList[i] = tempInfoList[i];
    }

    return vaultInfoList;
}
```

This implementation allocates memory for `tempInfoList` based on the total number of rounds, which may be significantly larger than the number of matching rounds. It then creates a second array vaultInfoList with the correct size and copies the data over.

Recommendation

Consider using a two-pass approach or a dynamic array to determine the exact size needed. Here's an example of a two-pass approach:

```
function getVaultInfoBySymbol(string memory symbol) public view returns
(VaultRound.VaultRoundInfoView[] memory) {
    uint256 matchCount = 0;

    // First pass: Count matches
       for (uint256 roundIndex = 0; roundIndex < getVaultManager().getRoundCount();
roundIndex++) {
```

```
                                VaultRound.VaultRoundInfoView    memory    roundInfo    =
getVaultManager().getVaultRoundByRoundIndex(roundIndex);

            for  (uint256  vaultIndex  =  0;  vaultIndex  <  roundInfo.vaultInfoList.length;
vaultIndex++) {

              if (keccak256(abi.encodePacked(roundInfo.vaultInfoList[vaultIndex].symbol))
== keccak256(abi.encodePacked(symbol))) {

                matchCount++;

                break;  // Only count each round once

            }

        }

    }


    // Allocate array with exact size

                 VaultRound.VaultRoundInfoView[]    memory    vaultInfoList    =    new
VaultRound.VaultRoundInfoView[](matchCount);

    uint256 currentIndex = 0;


    // Second pass: Populate array

       for  (uint256  roundIndex  =  0;  roundIndex  <  getVaultManager().getRoundCount();
roundIndex++) {

                                VaultRound.VaultRoundInfoView    memory    roundInfo    =
getVaultManager().getVaultRoundByRoundIndex(roundIndex);

        bool isMatched = false;

                        VaultRound.VaultInfo[]   memory   matchedVaultInfoList   =   new
VaultRound.VaultInfo[](roundInfo.vaultInfoList.length);

        uint256 vaultCount = 0;


            for  (uint256  vaultIndex  =  0;  vaultIndex  <  roundInfo.vaultInfoList.length;
vaultIndex++) {
```

```
            VaultRound.VaultInfo memory vaultInfo = roundInfo.vaultInfoList[vaultIndex];

                                if   (keccak256(abi.encodePacked(vaultInfo.symbol))   ==
keccak256(abi.encodePacked(symbol))) {

                matchedVaultInfoList[vaultCount] = vaultInfo;

                vaultCount++;

                isMatched = true;

            }

        }


        if (isMatched) {

            vaultInfoList[currentIndex] = VaultRound.VaultRoundInfoView({

                // ... (populate fields as before)

            });

            currentIndex++;

        }

    }


    return vaultInfoList;

}
```

Status

Resolved

QA

[Q-01] Validator - Redundant String Conversion in Require Statements

Description

In the Validator contract, several require statements use the `string()` function to convert error messages:

```
require(value != address(0), string("Validator:: zero address"));

require(value > 0, string("Validator:: zero or less than zero"));

require(value >= 0, string("Validator:: less than zero"));
```

This `string()` conversion is unnecessary in Solidity versions 0.8.0 and later, as these versions allow direct use of string literals in require statements. The redundant conversion may lead to slightly increased gas costs and reduced code readability.

Recommendation

Remove the `string()` conversion from all require statements in the contract. Update the statements to directly use string literals:

```
require(value != address(0), "Validator:: zero address");

require(value > 0, "Validator:: zero or less than zero");

require(value >= 0, "Validator:: less than zero");
```

Status

Resolved

[Q-02] Validator - Unnecessary Non-Negative Check for Unsigned Integer

Description

In the Validator contract, there is a function `checkMoreThanEqualZero` and a corresponding modifier `validMoreThanEqualZero` that check if a `uint256` value is greater than or equal to zero:

```
function checkMoreThanEqualZero(uint256 value) internal pure {

    require(value >= 0, "Validator:: less than zero");

}


modifier validMoreThanEqualZero(uint256 value) {

    checkMoreThanEqualZero(value);

    _;

}
```

This check is redundant because `uint256` is an unsigned integer type in Solidity, which means it can never be negative. The value of a `uint256` is always greater than or equal to zero by definition. This unnecessary check adds gas cost and complexity to the contract without providing any actual validation.

Recommendation

Remove the `checkMoreThanEqualZero` function and the `validMoreThanEqualZero` modifier entirely from the contract. If there's a need to ensure a value is strictly greater than zero, use the existing `validMoreThanZero` modifier instead.

If these were intended to check for specific business logic rather than type constraints, consider renaming them to better reflect their purpose, or replace them with more meaningful validation that aligns with the contract's requirements.
By removing this redundant check, you'll simplify the contract, reduce gas costs, and eliminate a potential source of confusion for developers working with the contract in the future.

Status

Resolved

[Q-03] VaultManagerHelper#getVaultInfoViewListByRoundIndexAndVaultIndex - Unused Variable

Description

In the `getVaultInfoViewListByRoundIndexAndVaultIndex` function of the `VaultManagerHelper` contract, there is an unused variable `nowTime`. This variable is declared and assigned the current block timestamp but is never used in the function's logic.

The problematic code is as follows:

```
function getVaultInfoViewListByRoundIndexAndVaultIndex(uint256 roundIndex, uint256 vaultIndex) public view returns (VaultRound.VaultInfoView memory) {
    // ... (other code)

    IVaultService vaultService = IVaultService(vaultInfo.activeVaultServiceAddress);

    uint256 nowTime = block.timestamp;  // This variable is declared but never used

    uint256 rewardPerToken = vaultService.earnedPerToken();

    // ... (rest of the function)
}
```

This `string()` conversion is unnecessary in Solidity versions 0.8.0 and later, as these versions allow direct use of string literals in required statements. The redundant conversion may lead to slightly increased gas costs and reduced code readability.

Recommendation

Remove the unused `nowTime` variable to save gas and improve code clarity. The modified function should look like this:

```
function getVaultInfoViewListByRoundIndexAndVaultIndex(uint256 roundIndex, uint256 vaultIndex) public view returns (VaultRound.VaultInfoView memory) {

                        require(roundIndex < getVaultManager().getRoundCount(), "VaultManagerHelper::getVaultinfoViewListByRoundIndexAndVaultIndex: invalid round index");


                        VaultRound.VaultRoundInfoView memory roundInfo = getVaultManager().getVaultRoundByRoundIndex(roundIndex);

    VaultRound.VaultInfo[] memory vaultInfoList = roundInfo.vaultInfoList;


                                require(vaultIndex < vaultInfoList.length, "VaultManagerHelper::getVaultinfoViewListByRoundIndexAndVaultIndex: invalid vault index");


    VaultRound.VaultInfo memory vaultInfo = vaultInfoList[vaultIndex];


    IVaultService vaultService = IVaultService(vaultInfo.activeVaultServiceAddress);


    uint256 rewardPerToken = vaultService.earnedPerToken();


    return VaultRound.VaultInfoView({
        state: vaultInfo.state,
        activeVaultServiceAddress: vaultInfo.activeVaultServiceAddress,
        stakingTokenBillReceiveAddress: vaultInfo.stakingTokenBillReceiveAddress,
        symbol: vaultInfo.symbol,
```

```
        name: vaultInfo.name,
        tokenName: vaultInfo.tokenName,
        tokenSymbol: vaultInfo.tokenSymbol,
        totalReward: vaultInfo.totalReward,
        totalAmount: vaultInfo.totalAmount,
        totalStakedAmount: vaultInfo.totalStakedAmount,
        totalUnStakedAmount: vaultInfo.totalUnStakedAmount,
        rewardDebt: vaultInfo.rewardDebt,
        rewardPerTokenPaid: vaultInfo.rewardPerTokenPaid,
        rewardPerToken: rewardPerToken
    });
}
```

Status

Resolved

[Q-04] VaultService - Unused State Variable

Description

The `VaultService` contract contains an unused state variable `totalPaidReward`. This variable is declared but never used throughout the contract:

```
uint256 private totalPaidReward;
```

Unused state variables in Solidity contracts waste gas during deployment and can lead to confusion for developers maintaining or auditing the code.

Recommendation

Remove the unused `totalPaidReward` variable from the contract. If this variable was intended for future use, consider adding it only when it's needed, or add a comment explaining its intended future purpose.

The modified code should look like this:

```
contract VaultService is Validator, Ownable, AccessControl, ReentrancyGuard {

    // ... other variables ...

    uint256 private totalReward;

    // uint256 private totalPaidReward; // Remove this line

    uint256 private totalClaimedReward;

    // ... rest of the contract ...

}
```

Status

Resolved

Centralisation

The Lair Finance project values security and utility over decentralisation.

Lair Finance emphasizes a security-focused approach while ensuring operational functionality. By centralizing critical functions, the project enhances reliability and rapid decision-making while maintaining accountability, ensuring the system remains robust and efficient.

Conclusion

After Hashlock's analysis, the Lair Finance 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