hashlock.

Security Audit

Functionland (DePIN)

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

Table of Contents

CAUTION

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

Executive Summary

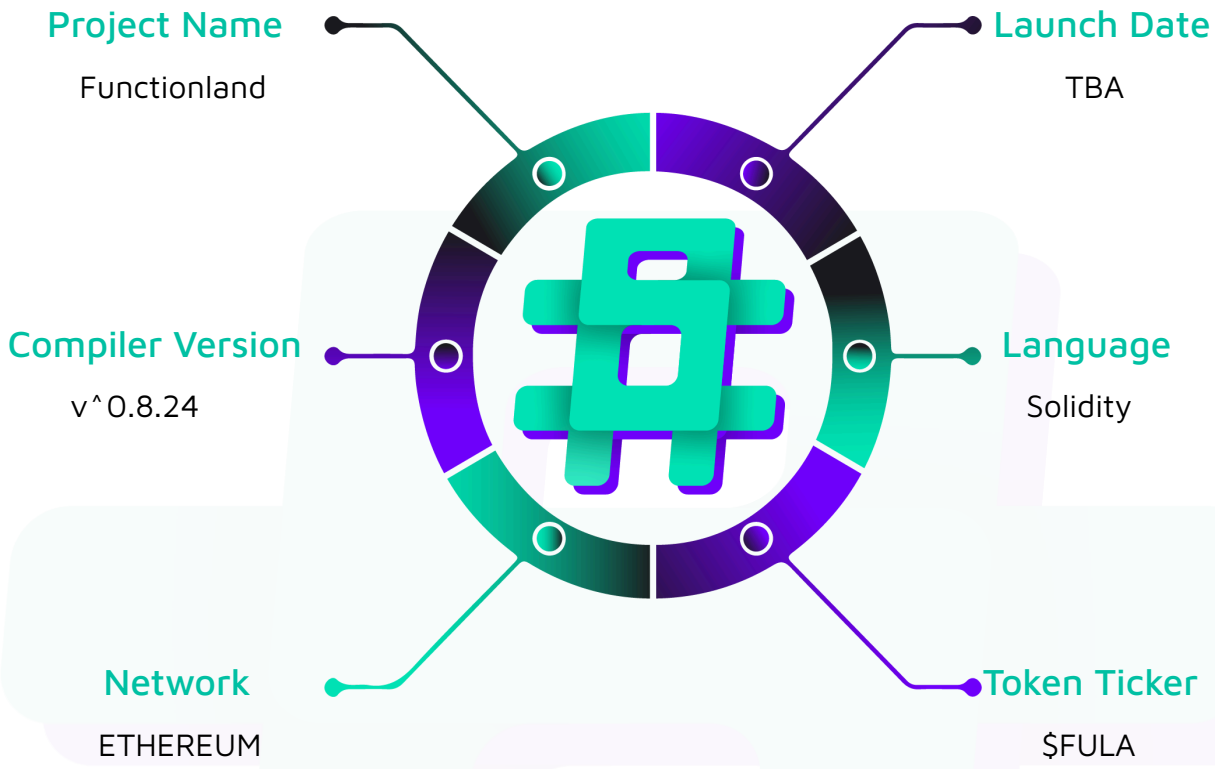
The Functionland team partnered with Hashlock to conduct a security audit of their GovernanceModule.sol, Treasury.sol, TokenDistributionEngine.sol, StorageToken.sol and other smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

Functionland is pioneering a decentralized approach to data storage with its FxBlox hardware and the Fula Network. By utilizing personal devices instead of central servers, users can own, store, and monetize their data securely. The Fula ecosystem is designed for scalability, allowing users to easily expand storage by connecting additional hard drives to their devices. It's also environmentally friendly, operating efficiently on low-power devices like Raspberry Pi. Functionland's modular design enables other projects to seamlessly integrate with FxBlox, fostering a diverse and user-owned data ecosystem.

Project Name: Functionland
Project Type: DePIN, Storage dApp, Tokenization
Compiler Version: ^0.8.24
Website: https://fx.land/
Logo:

Visualised Context:



Project Visuals:

Audit scope

We at Hashlock audited the solidity code within the Functionland 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	Functionland Protocol Smart Contracts
Platform	**Ethereum / Solidity**
Audit Date	**January, 2025**
Contract 1	Lock.sol
Contract 1 MD5 Hash	a822b17c2aa12ed2c511e4ca1ed3f385
Contract 2	Proxies.sol
Contract 2 MD5 Hash	9407c55421718cf27e7900256045c39d
Contract 3	Treasury.sol
Contract 3 MD5 Hash	7084ae957a90305691673db34667b39b
Contract 4	GovernanceModule.sol
Contract 4 MD5 Hash	70c9de5b9a7bebda4d13c238b5538e63
Contract 5	ProposalTypes.sol
Contract 5 MD5 Hash	f5d9ed6fdd804a89b6056f760ada5a13
Contract 6	StorageToken.sol
Contract 6 MD5 Hash	79ebdfd54cddcde8cfa020a3373a4dfd
Contract 7	TokenDistributionEngine.sol
Contract 7 MD5 Hash	fa40aa689dac651b49796714b6de8bbb
GitHub Commit Hash	a036edc40860156ea8252aed3785ede6eac19237

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.

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:

2 High severity vulnerabilities

3 Medium severity vulnerabilities

3 Low severity vulnerabilities

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
StorageToken.sol - ERC20 token with governance capabilities - Handles blacklisting/whitelisting of addresses - Collects transaction fees for treasury - Manages bridge operations for cross-chain transfers - Allows token burns - Implements upgradeable pattern - Allows admin role management	**Contract achieves this functionality.**
Treasury.sol - Fee management contract - Collects and holds fees from token transfers - Allows fee withdrawal by admin - Access control for operations	**Contract achieves this functionality.**
GovernanceModule.sol - Multi-signature governance functionality - Handles various proposal types - Role-based access control - Emergency pause mechanism - Upgradeable contract pattern - Manages ownership transfers - Handles quorum settings	**Contract achieves this functionality.**

TokenDistributionEngine.sol - Manages token distribution with vesting - Implements cliff and vesting periods - Handles cap-based allocation system - Token Generation Event (TGE) management - Allows token claims based on vesting schedule - Manages distribution wallets - Upgradeable functionality	**Contract achieves this functionality.**
Lock.sol - Simple time-lock contract - Managed by owner - Time-based unlocking mechanism	**Contract achieves this functionality.**
ProposalTypes.sol - Defines various proposal types (Roles, Upgrades, Recovery, Whitelist/Blacklist, Treasury) - Sets time constraints for governance (24h execution delay, 3 day timeout, etc) - Manages core roles (Admin, Bridge Operator, Contract Operator) - Handles proposal configurations and tracking - Controls role permissions and limitations - Provides time-lock mechanisms - Maintains whitelist/blacklist operations	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Functionland 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 Functionland 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] GovernanceModule#_checkExpiredProposal - Expired Proposals Cause Permanent State Bloat and Critical Governance Failure

Description

A critical vulnerability exists in the `GovernanceModule` contract where expired proposals get permanently stuck in storage due to incorrect state cleanup handling. The `_checkExpiredProposal` function reverts after attempting state changes, leading to permanent storage bloat and blocking vital governance functionality.

Vulnerability Details

The vulnerability centers around a critical flaw in the proposal lifecycle management within the `GovernanceModule` contract. Let's analyze the complex interaction of components that create this vulnerability:

1. Proposal Lifecycle Status

```
struct ProposalConfig{

	uint64 expiryTime; // 48 hours from creation

	uint64 executionTime; // 24 hours from creation

	uint16 approvals;

	uint8 status; // 0: active, 1: executed

}
```

2. Core Storage Architecture

```
mapping(bytes32 => ProposalTypes.UnifiedProposal) public proposals;

mapping(address => ProposalTypes.PendingProposals) public pendingProposals;
```

```
mapping(address => bytes32) public upgradeProposals;

uint256 public proposalCount;

mapping(uint256 => bytes32) public proposalRegistry;
```

The vulnerability lies in the `_checkExpiredProposal` function:

```
function _checkExpiredProposal(bytes32 proposalId) internal {

    ProposalTypes.UnifiedProposal storage proposal = proposals[proposalId];

    if (block.timestamp >= proposal.config.expiryTime) {

        // Attempt cleanup based on proposal type

        if (proposal.proposalType == uint8(ProposalTypes.ProposalType.Upgrade)) {

            delete upgradeProposals[proposal.target];

        } else {

            _handleCustomProposalExpiry(proposalId);

        }

        // Critical state changes that get reverted

        delete pendingProposals[proposal.target];

        delete proposals[proposalId];

        proposalCount -= 1;

        _removeFromRegistry(proposalId);

        emit ProposalExpired(proposalId, proposal.proposalType, proposal.target);

        revert ProposalErr(2);  // Reverts all state changes

    }

}
```

The function is called in three critical governance paths:

1. During Proposal Approval:

```
function approveProposal(bytes32 proposalId) external {
```

```
    ProposalTypes.UnifiedProposal storage proposal = proposals[proposalId];

    // ... validations ...

    _checkExpiredProposal(proposalId);  // Reverts if expired

    proposal.hasApproved[msg.sender] = true;

    proposal.config.approvals++;

}
```

2. During Proposal Execution:

```
function _executeProposal(bytes32 proposalId) internal {

    ProposalTypes.UnifiedProposal storage proposal = proposals[proposalId];

    _checkExpiredProposal(proposalId);  // Reverts if expired

    // Execution logic per proposal type

    if (proposal.proposalType == uint8(ProposalTypes.ProposalType.AddRole)) {

        // ... role execution logic ...

    } else if (proposal.proposalType == uint8(ProposalTypes.ProposalType.Recovery)) {

        // ... recovery execution logic ...

    }

}
```

3. During Contract Upgrades

```
function _checkUpgrade(address newImplementation) internal virtual returns (bool) {

    bytes32 currentId = upgradeProposals[newImplementation];

    ProposalTypes.UnifiedProposal storage currentProposal = proposals[currentId];

    _checkExpiredProposal(currentId);  // Reverts if expired

}
```

The vulnerability creates complex state management issues:

Timing Window Problem: Proposals have 48 hours until expiry(`PROPOSAL_TIMEOUT`) and must be executed after 24 hours(`MIN_PROPOSAL_EXECUTION_DELAY`) which creates 24-hour windows for valid execution. After expiry, the proposal becomes permanently stuck, also the state inconsistency which is shown below:

```
// These states remain out of sync

proposalCount                 // Never decrements

proposals[proposalId]         // Proposal remains stored

pendingProposals[target]     // Keeps expired proposal type

proposalRegistry             // Keeps expired proposal ID
```

And possible compound effects:

```
// Each new proposal adds to storage

proposalRegistry[proposalCount] = proposalId;

proposalCount += 1;

pendingProposals[target].proposalType = uint8(proposalType);
```

This creates a cascading effect where expired proposals remain in storage, affecting multiple contract states and preventing crucial governance operations. The combination of these factors creates a systemic issue that grows worse over time, affecting the entire governance mechanism's functionality and efficiency.

Proof of Concept

An example of a test that simulates the above issue is shown below.

```
function testExpiredProposalDOS() public {

       // Create proposal

       vm.startPrank(admin);

       bytes32 proposalId = gov.createProposal(

           uint8(ProposalTypes.ProposalType.AddRole),

           0,
```

```
            user,
            ProposalTypes.ADMIN_ROLE,
            0,
            address(0)
        );
        // Forward time past expiry (48 hours)
        vm.warp(block.timestamp + 49 hours);
        // Try to approve - will revert
        vm.expectRevert(abi.encodeWithSelector(ProposalErr.selector, 2));
        gov.approveProposal(proposalId);
        // Verify proposal still exists
        assertTrue(gov.proposals(proposalId).target != address(0));
        assertEq(gov.proposalCount(), 1);
        // Try to execute - will revert
        vm.expectRevert(abi.encodeWithSelector(ProposalErr.selector, 2));
        gov.executeProposal(proposalId);
        // Verify no state changes
        assertTrue(gov.proposals(proposalId).target != address(0));
        assertEq(gov.proposalCount(), 1);
        vm.stopPrank();
```

Impact

Governance Disruption as there is no mechanism to clean expired proposals and proposals can't be approved/executed or removed after expiry. Along with each expired proposal permanently occupies storage increasing gas costs for contract operations as storage grows unbounded over time.

Recommendation

- Separate Expiry Check and Cleanup in different functions to make sure the flow is smooth.
- Instead of reverting the state on `approveProposal` and `executeProposal`, just return false and make the state changes accordingly.

Status

Resolved

[H-02] StorageToken#_setPlatformFee - Fee Calculation Logic Combined with MAX_BPS Setting Allows 100% Fee Capture

Description

The `StorageToken` contract has a critical vulnerability in its fee mechanism where the combination of fee calculation logic and maximum fee setting allows capturing 100% of transfer amounts as fees.

Vulnerability Details

The vulnerability stems from the interaction between fee setting and transfer mechanisms in the `StorageToken` contract. Let's analyze the components:

1. Fee Configuration:

```
uint256 private constant MAX_BPS = 500; // 5%

uint256 private platformFeeBps;

function _setPlatformFee(uint256 _platformFeeBps) internal whenNotPaused {

    if(_platformFeeBps > MAX_BPS) revert FeeExceedsMax(_platformFeeBps);

    platformFeeBps = _platformFeeBps;

}
```

As you can see, the `MAX_BPS` is set to 500(5%) and Validation only prevents that input parameter from being more than 500 BPS. But the fees can be set to exactly 500 BPS and it works as per the logic.

2. Fee Application During Transfers:

```
function _update(address from, address to, uint256 amount) internal virtual {

    if (platformFeeBps > 0) {

        uint256 fee = (amount * platformFeeBps) / MAX_BPS;

        super._update(from, address(treasury), fee);

        super._update(from, to, amount - fee);

    }

}
```

When `platformFeeBPS = MAX_BPS = 500`:

Fee is calculated as: `fee = amount * 500 / 500 = amount`

This results in

- Treasury receives: amount
- Recipient receives: amount - amount = 0

The lack of a reasonable upper bound on fees along with Wrong Fee calculation logic allows the USER to lose all the Amount in terms of fees alone thus disrupting the whole flow as it breaks token functionality and value loss.

Impact

The combination of flawed fee calculation and maximum fee validation creates a critical issue where transfers can be completely diverted to the treasury. This leads to total value capture from transfers, breaks core token functionality, and causes direct user losses.

Recommendation

Fix the Fee Calculation in `_update` function

```
uint256 fee = (amount * platformFeeBps) / 10000;

// Where MAX_BPS will be still 500(5%)
```

Status

Resolved

Medium

[M-01] GovernanceModule#_executeProposal - Recovery Proposals Lack Cleanup Afrer Execution Allowing Multiple Token Drains within Execution Window

Description

The `GovernanceModule` contract contains a critical oversight in its proposal cleanup mechanism where Recovery type proposals remain active in storage after execution. While other proposal types properly clean up their state after execution, Recovery proposals lack this cleanup step, allowing the same proposal to be executed multiple times within the 24-48 hour execution window.

Vulnerability Details

The vulnerability exists due to the different handling of Recovery proposals in `_executeProposal`. Recovery proposals can be executed multiple times in the window between `executionTime`(24h) and `expiryTime`(48h).

Let's look into the function:

1. Role Proposal Cleanup (Complete)

```
if (proposal.proposalType == uint8(ProposalTypes.ProposalType.AddRole)) {

    _executeCommonProposal(proposalId);

    // Proper cleanup

    delete pendingProposals[proposal.target];

    delete proposals[proposalId];

    proposalCount -= 1;

    _removeFromRegistry(proposalId);
```

```
}
```

2. Recovery Proposal (No Cleanup)

```
else if (proposal.proposalType == uint8(ProposalTypes.ProposalType.Recovery)) {

    IERC20 token = IERC20(proposal.tokenAddress);

    bool success = token.transfer(proposal.target, proposal.amount);

    // No cleanup - can be executed again

    // Only limited by:

    // 1. 24-48h execution window

    // 2. Contract token balance

}
```

Proof of Concept

Here is the Proof-of-Concept for the above-given issue:

```
function testRecoveryProposalReuse() public {

    address token = address(new MockERC20());

    MockERC20(token).mint(address(gov), 1000e18);

    // Create and get proposal timestamps

    vm.prank(admin);

    bytes32 proposalId = gov.createProposal(

        uint8(ProposalTypes.ProposalType.Recovery),

        0,

        user,

        0,

        100e18,

        token

    );
```

```
    // Move to execution window (after 24h, before 48h)
    vm.warp(block.timestamp + 25 hours);
    // First execution
    vm.prank(admin);
    gov.executeProposal(proposalId);
    assertEq(MockERC20(token).balanceOf(user), 100e18);


    // Second execution still works
    vm.prank(admin);
    gov.executeProposal(proposalId);
    assertEq(MockERC20(token).balanceOf(user), 200e18);
    // Proposal still exists until expiry
    assertTrue(gov.proposals(proposalId).target != address(0));
}
```

Impact

While requiring admin access, Recovery proposals can be executed multiple times between 24-48h window, and each execution transfers tokens to the target. Additionally, it creates storage bloat until the proposal expires(if High-01 is fixed) and breaks the expected single-execution proposal lifecycle.

Recommendation

Add proper cleanup after Recovery Proposal Execution:

```
else if (proposal.proposalType == uint8(ProposalTypes.ProposalType.Recovery)) {
    // ... existing recovery logic ...
    // Add cleanup
    delete pendingProposals[proposal.target];
    delete proposals[proposalId];
```

```
    proposalCount -= 1;

    _removeFromRegistry(proposalId);

    emit ProposalExecuted(proposalId, proposal.proposalType, proposal.target);

}
```

Status

Resolved

[M-02] GovernanceModule#_removeFromRegistry - Unbounded Loop in _removeFromRegistry can Cause Denial of Service

Description

The GovernanceModule's `_removeFromRegistry` function contains an unbounded loop that iterates through all proposals, potentially causing out-of-gas errors when the number of proposals grows large, blocking critical governance operations.

Vulnerability Details

```
function _removeFromRegistry(bytes32 proposalId) internal {

    uint256 count = proposalCount;

    uint256 i;

    for (; i < count;) {

        if (proposalRegistry[i] == proposalId) {

            if (i != --count) {

                proposalRegistry[i] = proposalRegistry[count];

            }

            delete proposalRegistry[count];

            proposalCount = count;

            break;

        }
```

```
        unchecked { ++i; }

    }

}
```

This function is called in crucial governance operations such as when executing Proposals, cleaning up expired proposals, and handling proposal expiry

The unbounded loop means gas costs grow linearly with the proposal count. With enough proposals, the function could exceed block gas limits.

Impact

The unbounded loop in `_removeFromRegistry` creates significant operational risks for the governance system. As proposal count increases, gas costs for proposal execution and cleanup grow linearly, eventually hitting block gas limits. This could permanently block critical governance operations since there's no way to partially clean the registry. The impact is particularly severe because this function is called during essential operations like proposal execution and cleanup, meaning the entire governance system could become unusable once a certain number of proposals is reached.

Recommendation

1. Add pagination to registry cleanup:

```
function cleanupRegistry(uint256 startIndex, uint256 endIndex) external {

    require(endIndex <= proposalCount && endIndex > startIndex, "Invalid range");

    for(uint256 i = startIndex; i < endIndex;) {

        if(proposals[proposalRegistry[i]].config.expiryTime <= block.timestamp) {

            _removeFromRegistry(proposalRegistry[i]);

        }

        unchecked { ++i; }

    }

}
```

2. Or implement batch removal with gas limit:

```
function _removeFromRegistry(bytes32 proposalId) internal {

    uint256 gasLeftStart = gasleft();

    uint256 i;



    while(i < proposalCount && gasleft() > GAS_BUFFER) {

        if (proposalRegistry[i] == proposalId) {

            // Existing removal logic

            break;

        }

        unchecked { ++i; }

    }

}
```

Status

Resolved

[M-03] Treasury - Admin Role revocation Can permanently lock contract Functionality

Description

The Treasury contract relies solely on OpenZeppelin's `AccessControl` for role management but lacks protection against complete admin role removal, which could permanently lock all admin-controlled functions.

Vulnerability Details

In the constructor, here's what the actual code looks like:

```
constructor(address _storageToken, address _admin) {
```

```
    if(_admin == address(0) || _storageToken == address(0))
        revert F(0);
    storageToken = _storageToken;
    _grantRole(DEFAULT_ADMIN_ROLE, _admin);
}
```

The critical issue is that `DEFAULT_ADMIN_ROLE` holders can: Revoke their own role(using `renounceRole` function), Remove all other admins and leave the contract without any admin.

This affects `withdrawFees`:

```
function withdrawFees(address t, uint256 a)
    external
    nonReentrant
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    // Function becomes permanently inaccessible
    // if all admins are removed
}
```

Impact

If the admin role is revoked either accidentally or maliciously, the Treasury becomes permanently unusable. This would lock all fee management functionality, including the ability to withdraw accumulated fees. There is no recovery mechanism possible without contract redeployment, making any funds in the Treasury permanently trapped.

Recommendation

1. Add minimum admin check:

```
function revokeRole(bytes32 role, address account)
```

```
    public

    override

{

    if (role == DEFAULT_ADMIN_ROLE) {

        require(getRoleMemberCount(role) > 1, "Cannot remove last admin");

    }

    super.renounceRole(role, account);

}
```

2. Or implement a recovery mechanism:

```
function recoverAdmin(address newAdmin) external {

    require(msg.sender == storageToken, "Only storage token");

    _grantRole(DEFAULT_ADMIN_ROLE, newAdmin);

}
```

Status

Resolved

Low

[L-01] Lock#Constructor - Trapped ETH During Deployment

Description

The Lock contract can accidentally receive and permanently trap ETH during contract deployment.

Vulnerability Details

```
contract Lock {

    constructor(uint _unlockTime) payable {

        // ETH sent during deployment is permanently trapped

        // No fallback/receive functions

        // No withdrawal functionality

    }

}
```

Impact

As the constructor only deploys one time, deployment ETH will be trapped and as there is no respective withdrawal functions available, the ETH will be considered as lost.

Recommendation

Add ETH rejection or recovery mechanism:

```
receive() external payable {

    revert("ETH not accepted");

}

// OR

function withdrawDeploymentETH() external {

    require(msg.sender == owner);
```

```
    owner.transfer(address(this).balance);

}
```

Status

Resolved

[L-02] TokenDistributionEngine#_removeWallet - Unused Internal Function Increases Contract Size

Description

The `_removeWallet` function in `TokenDistribution` contract is defined but never used, unnecessarily increasing contract size

Recommendation

Either remove the unused function or implement its usage if functionality is needed.

Status

Resolved

[L-03] GovernanceModule#_checkUpgrade - Assignment vs Comparison Operator in Status Check

Description

In `_checkUpgrade`, the status comparison uses == instead of =, preventing the status update.

```
function _checkUpgrade(address newImplementation) internal virtual returns (bool) {

    // ...

    currentProposal.config.status == 1;  // Comparison instead of assignment

    // ...

}
```

Recommendation

Change operator to assignment:

```
currentProposal.config.status = 1;
```

Status

Resolved

Centralisation

The Functionland project values security and efficiency over decentralisation.

Functionland 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 Functionland 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