#Hashlock.

Redbelly Network (Layer 1)

SMART CONTRACT

Security Audit

Platform

ETH

hashlock.com.au

MAY 2024

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT 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 THE USE OF THE CLIENT.

Executive Summary

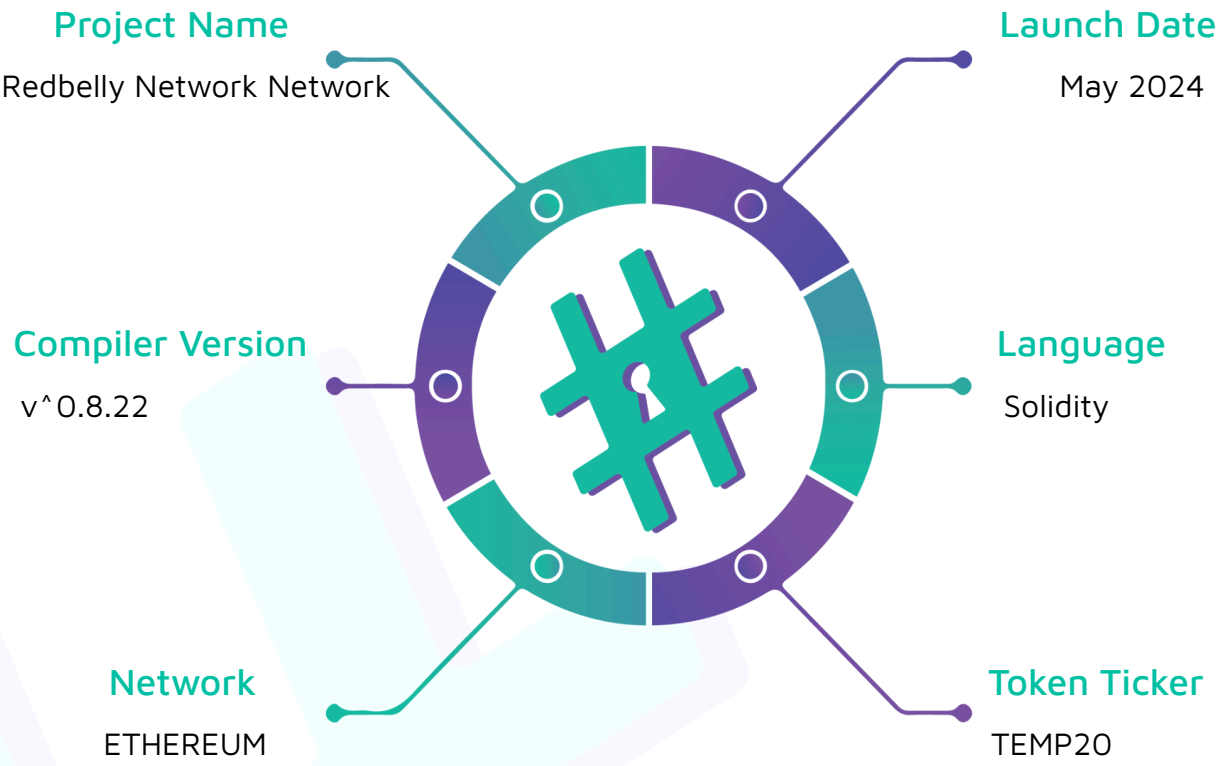
The Redbelly Network team partnered with Hashlock to conduct a security audit of their Sio2Adapter.sol smart contract. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts were secure.

Project Context

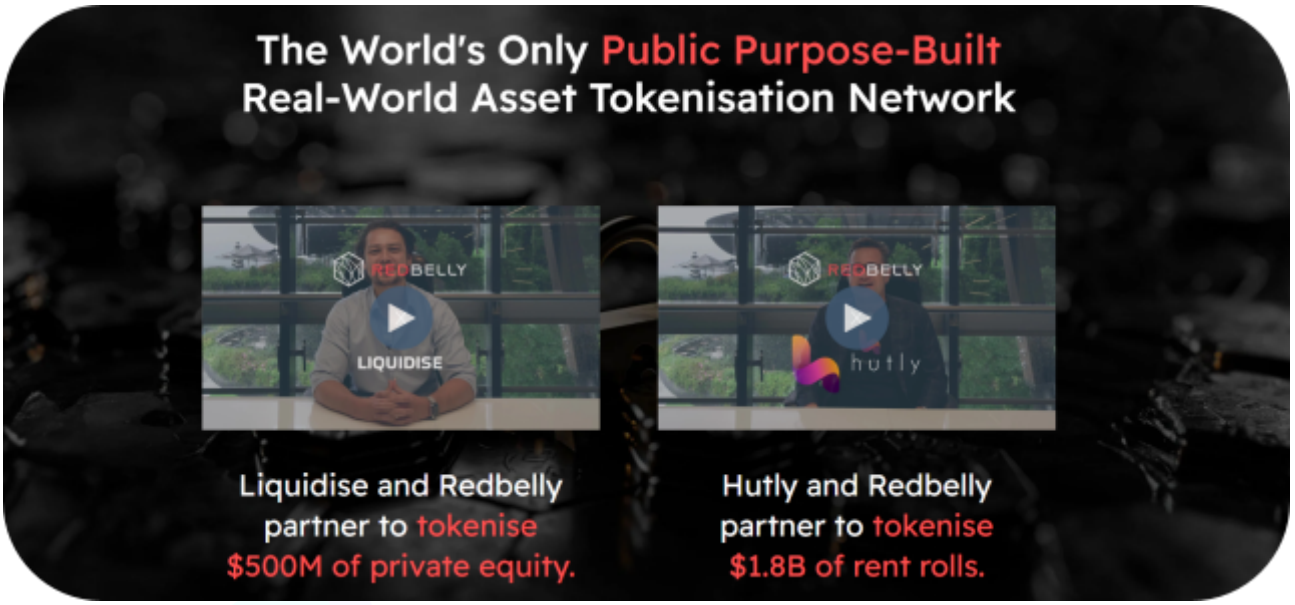
Redbelly Network Network is a Layer 1 Network that focuses on real-world assets, KYC, institutional use cases, and compliance. The Redbelly Network Network was born out of the University of Sydney and the CSIRO and has grown into a fully-fledged team that has made innovations in the technology and verified them via formal verification and extensive auditing. The Redbelly Network Network team engaged Hashlock to ensure that their protocol was ready for launch.

Project Name: Redbelly Network Network
Compiler Version: 0.8.22
Website: www.Redbelly.Network
Logo:

Visualised Context:



Project Visuals:



Audit scope

We at Hashlock audited the solidity code within the Redbelly Network 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	**Redbelly Network Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**May, 2024**
ActivityMonitor.sol	02b1ae41bdd90fad77d0966c30abcd03
BootstrapContractsRegistry.sol	7b9776414e580a5b00c7efb7d1f89070
ContractRoleAuth.sol	f93736e099bcdae4c11a5af4fe180aaf
Cornerstone.sol	eee15a98e3e65228075bb0931479d550
GasFees.sol	a70133cdb3ea2530f16e082005df8012
NetworkConfiguration.sol	bde621dff51263bc4826b8098f005e96
Permission.sol	c698d2fe0fa29b828671e369ec2aaead
PriceFeedContract.sol	9d0860554b30c26608e62acc0290f7f1
RBAC.sol	467f4c3f2541338288efbe8155ea47fc
Reconfiguration.sol	54a87cf8ec3f82ec4515ed0be63cc39f
NodeConfigStorage.sol	def92f1b435c3139a24826b484f70852
NodeRecordStorage.sol	48240040dc3ed3ed6eebbbfd3a085ee7
ActivityMonitorUpgradeable.sol	86e01d97355d70e06467616170fe968e
ContractRoleAuthUpgradeable.sol	e040b9b44aadfbabe84dde8ba0d7e028
IDPRegistryUpgradeable.sol	d5fe8a6b9596cdb0cf6f5976667b3f10
JailedNodesUpgradeable.sol	c4514018336d2826a6ac43909b9002df

NetworkConfigurationUpgradeable.sol	1acf635fc4fb06d7582d3bdca96db680
PermissionUpgradeable.sol	dce628e78ab057b2793ff49858a202b5
RBACUpgradeable.sol	4f8b3eba5079562eb8089993e6126416
ReconfigurationUpgradeable.sol	ced60adc75223b39c7d5d4ea6c1e3155
SignupBonusVestingUpgradeable.sol	b0fdc6120673d0736bd5e0e95fd96b5a
StakingDepositUpgradeable.sol	15808d4930dc26e39ee57dae6e2bbbca
StakingEscrowUpgradeable.sol	8ba75f4bf9df2e1f1aa2d41fe2263461
TombstonedNodesUpgradeable.sol	14b885dd53eebe72a23ffcc6cf47d070
JsonFormatter.sol	7502b945f5a6102859358b293e255340
MathUtils.sol	840d24bfd1ff9a2a16b70b93b029b016
PseudoRandomNumberGenerator.sol	b7a27d1d17cdb26c3e23af9c2e077f0a
StringToAddress.sol	2c7f980be93255cd898ed7d2b1e605a9
TimeParserUtils.sol	308c8b52039105e7f8b52855f34ff865
VotesMinHeap.sol	0ae6b0885de878263927be655df12f30
BusinessIdentifier	9079be48536e066086f32f477b61ce8e
BusinessIdentifierFactory	6f8f5a973fd8105833d1656e823f6669
RedbellyContractRegistry	85b0214fb97116e3e14e21ad0404ec7d
FTUAccountFunding	fa665c5a37b5afee94a8abd393691974
TransactionRefundable	bdd384a593bde51dad5bde2aa6692232

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Not Secure"**. The contracts contain self-contradictory, broken logic, and insufficient care when handling money. We have identified some significant vulnerabilities as well as malicious code that need to be addressed prior to launch.

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.

All vulnerabilities initially identified have yet to be resolved or acknowledged.

Hashlock found:

12 High-severity vulnerabilities

6 Medium-severity vulnerabilities

11 Low-severity vulnerabilities

7 Gas Optimisations

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

Intended Smart Contract Behaviours

Claimed Behaviour	Actual Behaviour
FTUAccountFunding.sol • **Initialize Contract:** Sets the initial values, including a refund limit of 2 ether. • **Release Signed Funds:** Transfers funds to a verified account, ensuring the signature matches the message hash. • **Release Funds (Unverified):** Transfers funds directly to an account after checking if it's already funded. • **Set Fund Amount:** Allows an administrator to specify the fund amount to be released to accounts. • **Withdraw Funds:** Enables the administrator to withdraw a specified amount if the contract has enough balance.	**Contract achieves this functionality.**
ContractRoleAuthUpgradeable.sol • **Initialize Contract:** Calls the base initialization function to set up roles. • **Only Staking Deposit Contract:** Restricts access to functions for those with the "STAKING_DEPOSIT_CONTRACT_ROLE." • **Only Jailed Nodes Contract:** Limits function access to accounts with the "JAILED_NODES_CONTRACT_ROLE." • **Only Network Configuration Contract:** Grants function access exclusively to "NETWORK_CONFIG_CONTRACT_ROLE" holders. • **Only Activity Monitor or Redbelly Role:** Permits function calls for "ACTIVITY_MONITOR_CONTRACT_ROLE" or "REDBELLY_ROLE." • **Only Jailed Nodes or Redbelly Role:** Allows access to those with "JAILED_NODES_CONTRACT_ROLE" or "REDBELLY_ROLE."	**Contract achieves this functionality.**
IDPRegistryUpgradeable.sol • **Behavior:**	**Contract achieves this functionality.**

○ Manages a registry of Identity Providers (IDPs) with various attributes, proofs, and permissions. ○ Supports adding, updating, and removing IDPs via their unique UID or issuer DID. ○ **register:** Adds a new IDP to the registry. ○ **getAll:** Returns all non-deleted IDPs. ○ **getByUid / getByIssuerDid:** Fetches IDP data by UID or issuer DID. ○ **updateByUid / updateByIssuerDid:** Updates IDP information with validation. ○ **removeByUid / removeByIssuerDid:** Marks an IDP as deleted by UID or issuer DID. ○ **getIssuersByProofType:** Returns issuers for a specific proof type. ○ **getSupportedProofTypes:** Lists all supported proof types.	
JailedNodesUpgradeable.sol ● **Initialization**: ○ initialize configures essential parameters and authorizes users. ● **Jailing & Releasing:** ○ **jail:** Imposes penalties on misbehaving nodes. ○ **freeServedGuilty / freeInnocent:** Handles the release of jailed nodes based on the penalty served. ● **Configuration:** ○ setSlashPrcnt and setDaysToServe: Adjust jailing policies. ● **Utilities:** ○ getRecentJailTenure, getJailCountForNode, isJailedNode: Retrieve information about node status. ○ getJailedDuration calculates the total time a node has served.	**Contract achieves this functionality.**
TransactionRefundable.sol 1. **Pause and Unpause:** ○ Allows holders of the REFUND_PAUSER_ROLE to pause or unpause the contract's operations.	

2. **Refund Gas:** ○ Automatically calculates and transfers the appropriate gas refund amount to eligible wallets (with the TXFEE_REFUNDS_ELIGIBLE_ROLE). 3. **Setters for Contract Parameters:** ○ **setShouldRevert:** Toggles whether the contract should revert transactions involving non-eligible wallets. ○ **setOffset:** Adjusts the offset value used in gas refund calculations. ○ **setMaxRefund:** Sets the maximum allowable refund value. 4. **Payment Reception:** ○ Implements a receive function to accept incoming Ether payments, emitting the PaymentReceived event.	
ActivityMonitor.sol 1. **Validation of Configuration Parameters:** ○ **InvalidInactivityThresholdForRounds:** Thrown if the threshold for inactivity rounds is set to zero or below. ○ **InvalidInactivityThresholdForMessage:** Thrown if the threshold for inactivity messages is set to zero or below. ○ **InvalidRecentVoteInterval:** Thrown if the recent vote interval is set to zero or below.	**Contract achieves this functionality.**
BootstrapContractsRegistry.sol 1. **Registry:** ○ A mapping storing contract names to their corresponding addresses. 2. **Events:** ○ **RegistryUpdated:** Emitted when a contract address is registered or updated. 3. **Errors:** ○ **EmptyContractName:** If the provided name is empty.	

○ **InvalidAddress:** If the provided address is zero. 4. **Methods:** ○ **register:** Updates/overwrites the mapping for a contract name. Restricted to authorized users. ○ **getContractAddress:** Returns the address for a given contract name.	
NetworkConfigurationContractUpgradeable.sol ○ **initialize:** Initializes with bootstrap nodes and contracts. ○ **getNetworkSize, isGovernor**, etc.: Retrieve node information. ○ addCandidate, removeCandidate, addGovernor, removeGovernor, etc.: Add/remove governors and candidates. ○ **register:** Registers new nodes with proper validation. ○ removeNode, removeGovernorFromNetwork: Manages node removal and replacement.	**Contract achieves this functionality.**
ContractRoleAuth.sol • **Purpose:** Implements access control mechanisms for specific contract roles using role-based access control (RBAC).	**Contract achieves this functionality.**
PermissionUpgradeable.sol **Purpose:** • Provides user permission management and identity verification within a decentralized network. • Supports credential verification for network access, manages authorized user addresses, and allows toggling of permissioned access for enhanced security.	**Contract achieves this functionality.**
Cornerstone.sol	**Contract achieves this functionality.**

Purpose: • Manages a collection of employee records and provides lookup functionality. • Enables initialization with a set of initial employees and allows retrieving employee details based on name or index. • Offers features to fetch comments associated with employees and supports error handling if an employee is not found.	
GasFeesContract • Allows managing gas fees with role-based access control (RBAC). • Facilitates updates to gas fee, currency, and decimal values by authorized users, ensuring accurate accounting of fees. • **Access Control:** ○ Inherits from RBAC to enforce permissions for updating gas fee parameters, ensuring only authorized users can make changes.	**Contract achieves this functionality.**
RBACUpgradeable • Provides Role-Based Access Control (RBAC) for managing permissions in a modular and extensible way. • Defines key roles, such as OWNER_ROLE and REDBELLY_ROLE, and includes modifiers to restrict access to functions based on roles. • Initializes default roles, like DEFAULT_ADMIN_ROLE and OWNER_ROLE, to the contract deployer.	**Contract achieves this functionality.**

NetworkConfigurationContract.sol
 	
• **Governor Management:**
 	
  ○ Allows retrieval of all governor node addresses to facilitate network governance.
• **Node Configuration:**
 	
  ○ Retrieves node configurations based on provided addresses for better insight into node
	Contract achieves this functionality.

properties. • **Node Role Verification:** ○ Checks whether a given node is a boot node, ensuring node roles are appropriately enforced. • **Candidate Nodes:** ○ Provides a placeholder implementation for getting active candidate nodes, returning an empty array for now.	
Permission • **Identity Management:** Provides a basic implementation for identity verification by allowing or denying access. • **Permission Check:** ○ isAllowed: Returns true unconditionally, meaning access is always granted. • **Access Control Status:** ○ isPermissionedAccessEnabled: Returns false, indicating permissioned access is not enforced.	**Contract achieves this functionality.**
SignupBonusVestingUpgradeable.sol **Purpose:** • Manages a signup bonus vesting schedule for node operators while enforcing access control. • **Signup Bonus Logic:** ○ Provides a mechanism for registering nodes and tracking their vesting periods and eligibility. ○ Implements a tiered signup bonus system based on node IDs. • **Vesting Management:** ○ Allows users to: ■ Register nodes for vesting. ■ Check the current vesting status using status(). ■ Claim eligible tokens with claim() if they	**Contract achieves this functionality.**

meet eligibility requirements.
 	
• **Vesting Logic:**
  ○ Determines signup bonus tiers based on node IDs.
  ○ Calculates claimable tokens by considering jailed periods and the total vesting period.	

StakingDepositUpgradeable
 	
Purpose:
• Manages staking deposits for nodes, including operations such as stake initialization, reward management, slashing, and unstaking. Implements role-based access control to manage permissions.
 	
Key Features:
• **Staking Management:**
  ○ Allows nodes to deposit stakes and manage their deposits over time.
  ○ Includes functionality for nodes to claim rewards based on their staking duration and behavior within the network.
• **Slashing Mechanism:**
  ○ Implements rules for slashing a portion of the stake if nodes misbehave, contributing to network security.
• **Cool-Off Periods:**
  ○ Enforces a cool-off period before unstaking can occur, enhancing network stability
	Contract achieves this functionality.

RBAC.sol **Purpose:** • A role-based access control (RBAC) contract that uses AccessControl from OpenZeppelin to provide a hierarchical permission system for smart contracts.	**Contract achieves this functionality.**

Role Management: ● Defines and manages roles for different levels of access within the Redbelly ecosystem.	
ReconfigurationContract.sol **Purpose:** ● A simple contract to manage and provide access to a "reconfiguration buffer" value. **Key Features:** 1. **State Variable:** ● _reconfigurationBuffer: Holds the buffer value as a uint256. 2. **Constructor**: ● Initializes _reconfigurationBuffer with a value provided during deployment. 3. **Public Getter Function:** ● getReconfigurationBuffer: Allows anyone to read the value of _reconfigurationBuffer.	**Contract achieves this functionality.**
BusinessIdentifierFactory.sol **Purpose:** ● Deploys BusinessIdentifier contracts that manage company information. ● Facilitates updating wallet addresses linked to businesses tract.	**Contract achieves this functionality.**
RedbellyContractRegistry.sol **Purpose:** ● Manages a core registry for different contracts by their names, allowing authorized users to add and access contracts.	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the Redbelly Network 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 and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Redbelly Network projects smart contract code in the form of GitHub access.

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

Dependencies

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

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies

Audit Findings

High

[H-01] TransactionRefundable - Built-in wallet draining backdoor

Description

The deployer of the contract has the full ability to drain the wallet in one go, making the contract fully centralized and trivially vulnerable to rug pulls from the contract deployer.

Vulnerability Details

The deployer of the contract is granted the role `DEFAULT_ADMIN_ROLE`. This role can control the variables `maxRefund` and `offset`. This in turn means that when calling `refundGas()`, any amount can be drained from the contract, including all of it, in one go.

This is a classical rug pull mechanic and does not only facilitate criminal activity but is also vulnerable to disgruntled employees and other situations where even a closely guarded passphrase is liable to eventually leak.

Even if assumed to be well-meaning, this poses a fatal security vulnerability.

Impact

The contract is not trustworthy for potential users.

Recommendation

Remove any such capabilities.

Status

Resolved

[H-02] FTUAccountFunding#releaseFunds - Users who are granted the role TXFEE_REFUNDS_ELIGIBLE_ROLE can essentially drain FTUAccountFunding

Description

Any user with TXFEE_REFUNDS_ELIGIBLE_ROLE can repeatedly call releaseFunds, with a new account for each call, to essentially drain the FTUAccountFunding contract. We consider this a high-severity vulnerability because the documentation for the TransactionRefundable contract (inherited by FTUAccountFunding) states that it is "A Refunding capability to fund back the tx fee to a calling wallet", indicating that users will be granted the role TXFEE_REFUNDS_ELIGIBLE_ROLE, and hence the ability to essentially drain FTUAccountFunding.

Vulnerability Details

For context, the refundGas modifier on the releaseFunds function is restricting msg.sender to only the addresses with role TXFEE_REFUNDS_ELIGIBLE_ROLE and it refunds the gas cost of calling releaseFunds to msg.sender. The function is defined by

```
function releaseFunds(address payable account) external refundGas {
    _requireNotPaused();
    if (payments[account] == true) {
        revert TransferBalanceError("Already funded");
    }
    payments[account] = true;
    account.transfer(fundAmount);
    emit ReleasedFund(fundAmount, account);
}
```

There is a restriction that each account will be able to receive the funding payment at most once, but an attacker with role TXFEE_REFUNDS_ELIGIBLE_ROLE can call releaseFunds an unlimited number of times, with a different newly created account for each call, and receive funding payment an indefinite number of times. If the funding payment fundAmount is non-zero, then it is possible to essentially drain the FTUAccountFunding contract like this.

Proof of Concept

A Foundry test that demonstrates how a user with role `TXFEE_REFUNDS_ELIGIBLE_ROLE` can exploit this is

```
function test_drainWithReleaseFunds() public {
    // The test setup:
    FTUAccountFuncing ftu = new FTUAccountFunding();
    ftu.initialize();
    ftu.grantRole(ftu.TXFEE_REFUNDS_ELIGIBLE_ROLE(), address(this));
    ftu.setFundAmount(1 ether);
    vm.deal(address(ftu), 5 ether);
    vm.deal(address(this), 0);

    // Accounts, owned by the attacker, to receive funding payments:
    address[] memory accounts = new address[](5);
    for (uint256 i; i < accounts.length; ++i) {
        accounts[i] = vm.addr(i + 1);
    }

    assertTrue(ftu.hasRole(ftu.TXFEE_REFUNDS_ELIGIBLE_ROLE(), address(this)));
    for (uint256 i; i < accounts.length; ++i) {
        assertEq(address(accounts[i]).balance, 0);
    }
    assertEq(address(ftu).balance, 5 ether);

    for (uint256 i; i < accounts.length; ++i) {
        ftu.releaseFunds(payable(address(accounts[i])));
    }

    // The ether are now in the attackers possession:
    for (uint256 i; i < accounts.length; ++i) {
        assertEq(accounts[i].balance, 1 ether);
    }
    assertEq(address(ftu).balance, 0);
}
```

Impact

Under the assumption that some users are granted the role `TXFEE_REFUNDS_ELIGIBLE_ROLE`, one user can steal all of the ether in the `FTUAccountFunding` contract.

Recommendation

Allow `msg.sender` to only successfully call one of `releaseFunds` or `releaseSignedFunds` at most once. Alternatively, change `releaseFunds` and `releaseSignedFunds` to internal functions.

Status

Resolved

[H-03] FTUAccountFunding#releaseSignedFunds - Users who are granted the role TXFEE_REFUNDS_ELIGIBLE_ROLE can essentially drain FTUAccountFunding

Description

Any user with `TXFEE_REFUNDS_ELIGIBLE_ROLE` can repeatedly call `releaseSignedFunds`, with a new account, hash, and signature for each call, to drain the `FTUAccountFunding` contract. Finding H-02 is similar and contains more details.

Vulnerability Details

One can employ an analogous procedure for finding H-02. The difference being that the attacker needs to call `releaseSignedFunds` and create a new hash and signature together with the new account for each call.

Proof of Concept

Refer to the proof of concept of finding H-02. The difference here is that instead of addresses, the attacker can create wallets. In a Foundry test, this can be achieved with

```
Vm.Wallet[] memory wallets = new Vm.Wallet[](5);
for (uint8 i; i < wallets.length; ++i) {
    wallets[i] = vm.createWallet(string(abi.encodePacked(i + 65)));
}
```

One can create a valid hash and signature for the wallet at index `i` in Foundry and call `releaseSignedFunds` with

```
bytes32 hash = keccak256("hash");
(uint8 v, bytes32 r, bytes32 s) = vm.sign(wallets[i], hash);
bytes memory signature = abi.encodePacked(r, s, v);
ftu.releaseSignedFunds(payable(wallets[i].addr), signature, hash);
```

Impact

Under the assumption that some users are granted the role `TXFEE_REFUNDS_ELIGIBLE_ROLE`, one user can steal all of the ether in the `FTUAccountFunding` contract.

Recommendation

Allow `msg.sender` to only successfully call one of `releaseFunds` or `releaseSignedFunds` at most once. Alternatively, change `releaseFunds` and `releaseSignedFunds` to internal functions.

Status

Resolved

[H-04] NodeConfigStorage#remove - The value of self.nodeIdToConfig needs to be updated

Description

The `remove` function does not update `self.nodeIdToConfig` at index `i` correctly.

Vulnerability Details

The `remove` function is removing a node at a certain index `i` by moving the node at index `self.nodeCounter - 1` to index `i`, which is a good approach, also employed by Open Zeppelin for the EnumerableSet implementation. However, the current implementation of `remove` does not update `self.nodeIdToConfig` at index `i` correctly.

Proof of Concept

A Foundry test demonstrating this is

```
using NodeConfigStorage for NodeConfigStorage.NodeConfig;
NodeConfigStorage.NodeConfig self;
function test_getConfigForAddress() public {
    self.insert(NodeConfigStorage.NodeParameters(
        0, 0, 0, 0, 0, address(1), address(1), "", 0));
    self.insert(NodeConfigStorage.NodeParameters(
        0, 0, 0, 0, 0, address(2), address(2), "", 0));

    // First verify that the config for node at address 2 is correct:
    assertEq(self.getConfigForAddress(address(2)).nodeAddress, address(2));

    // Remove node with address 1
    self.remove(address(1));

    // Now config for node at address 2 is not correct anymore:
    assertEq(self.getConfigForAddress(address(2)).nodeAddress, address(0));
}
```

Impact

The immediate impact of the incorrect `self.nodeIdToConfig` at index `i` is that `getConfigForAddress` will return an incorrect empty/zero value for the new node at index `i`. There are multiple critical implications of this. One critical implication is that the node at index `i` will withdraw the staking rewards for the node at index `0` when calling `StakingDepositUpgradeable.withdrawStakingRewards`.

Recommendation

Suggestions for updating the `remove` function:

- Set `self.nodeIdToConfig[nodeIndex] = self.nodeIdToConfig[self.nodeCounter - 1]`
- Set `self.nodeIdToConfig[nodeIndex].id = nodeIndex`
- `self.signingAddressToNodeId` needs to be updated to the right signing address, otherwise function `getConfigForSignerAddress` will not work. This can be achieved by adding an inverse mapping `self.nodeIdToSigningAddress`, so that remove can update `self.signingAddressToNodeId[self.nodeIdToSigningAddress[self.counter - 1]] = nodeIndex`

  The new inverse mapping `nodeIdToSigningAddress` needs to be updated as well,

`self.nodeIdToSigningAddress[nodeIndex] = self.nodeIdToSigningAddress[self.nodeCounter - 1]`

- The mapping `self.nodeIndxToAddress` is not needed, because one can instead use `self.nodeIdToConfig[nodeIndex].nodeAddress`.

Note that updating `self.nodeIdToConfig[nodeIndex].id` will have an impact on other parts of the code. For example, contract `SignupBonusVestingUpgradeable` cannot use `nodeParameters.id` to determine whether the node is eligible for sign up bonus.

Status

Resolved

[H-05] NetworkConfigurationUpgradeable#swapNodesWithIndexesAndUpdateRoles - Any governor can be swapped out for a candidate

Description

The function `swapNodesWithIndexesAndUpdateRoles` allows anyone to freely select the set of governors from the registered nodes.

Proof of Concept

```
function test_swapOutBootNode() external {
        networkConfiguration.register{value: stakingDeposit.minimumStakeAmount()}(
                address(100), address(100), "100", 0, 0, 0);


        assertEq(
                networkConfiguration.getGovernorAddressAtIndex(0),
                address(1)
        );
        assertTrue(networkConfiguration.isBootNode(address(1)));
        assertEq(
                networkConfiguration.getCandidateAddressAtIndex(0),
                address(100)
        );
```

```
        vm.startPrank(address(0)); // Any address will work
        networkConfiguration.swapNodesWithIndexesAndUpdateRoles(0, 0);
        vm.stopPrank();

        assertFalse(networkConfiguration.isGovernor(address(1)));
        assertTrue(networkConfiguration.isGovernor(address(100)));
}
```

Impact

Any governor (even a bootstrap governor) can be swapped out at any time. This mitigates the effect of `ReconfigurationUpgradeable.triggerReconfiguration` and allows for swapping eliminating all bootstrap governors.

Recommendation

Only allow `ReconfigurationUpgradeable` to call `swapNodesWithIndexesAndUpdateRoles`.

Status

Resolved

[H-06] SignupBonusVestingUpgradeable#initialize - Why are signup bonuses 400k and 150k ETH?

Description

The `initialize` function is initialising the signup bonuses `_first200Bonus` and `_next400Bonus` to `400000 * 1e18` and `150000 * 1e18`, respectively. This means that users among the first 200 can expect to receive 400k ether and the next 400 users expect to receive 150k ether, over a 4 year period.

Proof of Concept

Earn more than 2700 ether signup bonus in just 10 days:

```
function test_signupBonus() external {
        User user = new User();
        uint256 registerValue = stakingDeposit.minimumStakeAmount();
        networkConfiguration.register{value: registerValue}(
```

```
        address(user), address(user), "user", 0, 0, 0);

        // Transfer tokens to SignupBonusVestingUpgradeable contract:
        vm.deal(address(signupBonusVesting), 4000 ether);

        // Initially user has no ether:
        assertEq(address(user).balance, 0);

        // Claim signup bonus every day for 10 days:
        vm.startPrank(address(user));
        for (uint256 i = 0; i < 10; ++i) {
        vm.warp(block.timestamp + 1 days);

        uint256 balanceBefore = address(user).balance;
        signupBonusVesting.claim();
        uint256 balanceAfter = address(user).balance;

        // Daily signup bonus is more than 273 ether:
        assertGt(balanceAfter, balanceBefore + 273 ether);

        vm.stopPrank();

        // User earned more than 2730 ether in 10 days:
        assertGt(address(user).balance, 2730 ether);
}
```

Impact

It seems that the signup bonuses are too large.

Recommendation

Initialise `_first200Bonus` and `_next400Bonus` to realistic values.

Status

Resolved / Invalid, since the target blockchain is Redbelly with RBNT as its native token.

[H-07] StakingDepositUpgradeable#withdrawStakingRewards - Incorrect computation of the number of staked reward cycles

Description

The function incorrectly calculates the number of reward cycles for which the node was staking using `timeStaked - int256(_rewardCycle)`.

Proof of Concept

Withdraw a million ether staking reward immediately after registering:

```
function test_withdraw() external {
    User user = new User();
    uint256 registerValue = stakingDeposit.minimumStakeAmount();
    networkConfiguration.register{value: registerValue}(
        address(user), address(user), "user", 0, 0, 0);

    // Transfer a healthy amount of ether to the StakingDepositUpgradeable contract
    vm.deal(address(stakingDeposit), 2_000_000 ether);

    // Initially user has no ether:
    assertEq(address(user).balance, 0);

    vm.startPrank(address(user));
    stakingDeposit.withdrawStakingRewards();
    vm.stopPrank();

    // User was able to claim 1 million ether for no time staked:
    assertGt(address(user).balance, 1_000_000 ether);
}
```

Impact

Two scenarios. One is that it is possible to underflow the value of variable `computeNum` when `_rewardCycle < timeStaked`, making the value of `computeNum` excessively large. This in turn makes the value of the variable `rewardsToBeClaimed` too large, and the function will likely revert when attempting to transfer a large amount of ether. The other scenario is

when `timeStaked` > `_rewardCycle`, in which case it seems that the value of `computeNum` likely becomes too large, due to counting the number of weeks minus reward cycle length instead of counting the number of reward cycles.

Recommendation

Calculate the number of reward cycles for which the node was staking with

```
timeStaked / int256(_rewardCycle)
```

Status

Resolved / Invalid

[H-08] Committer email address - Email address is not anonymized

Description

Commit author email addresses are not anonymized using the standard GitHub format users.noreply.github.co, which opens said developers to spearphishing and social engineering attacks, making a supply chain exploit a closer reality. This is a vector that has been used in past supply chain attacks. Many of those addresses are Gmail and other personal addresses, in which case a takeover of such an email address will lead to a compromise of accounts on other platforms where those addresses are registered as recovery email accounts.
This would lead to a compromise of dev environments and infrastructure. Git is a source of personal information about developers that is often overlooked and can lead to well-orchestrated attacks that developers do not expect in general, leading to their effectiveness which could be seen in recent times.

Proof of Concept

```
git log | grep '^Author' | sort | uniq -c
```

Impact

Developer machine could become compromised. Ultimately (see H-09, H-10) this could translate to a compromise of the whole live infrastructure.

Recommendation

Rewrite git history to use github style email addresses.

Status

Unresolved

[H-09] Insufficient care in using git and related toolkit and commit review

Description

The commit history contains commits that have not been properly created. This is visible because the commit message contains code comments from the text editor used in interactive commit mode. These commits have not been reviewed and have been made by developers who need to be taught how to use these tools properly.

Improper use of git can lead to endless multiple critical scenarios, some of which are:

- Incorrect, broken, or known vulnerable code making it back into the code base after it has been fixed
- Code making it into the code base that shouldn't be there, e.g. malicious code, code planted by supply chain exploits, etc

Every time the code is committed, utmost care must be taken that the commit is absolutely perfect. Self-review of commits made to a code base is a crucial element to creating safe code.

The fault does not lie with any one developer, as all contributing developers use git and should be reviewing git history and fixing it if necessary.

Proof of Concept

Multiple commits in rbn-core-sc, eg:

- 903f26fb5cd874b404bbf2a74317f24d49f7a4a2
- a33a090dd0bf06b2ccfcdd92f1f17b7c8f44268e

Impact

Higher incidence of exploits in code base. Code review as a defense against supply chain issues becomes ineffective. Ultimately, a foothold on a developer's machine could translate to compromise of the whole project and all associated platforms, cloud accounts, etc.

Recommendation

Developer training, courses, books, certification.

Status

Resolved

[H-10] Commits are not signed - Code provenance cannot be established

Description

Commits are not signed, which means that there is no positive sign that a git repository has not been exploited to include malicious code.

Vulnerability Details

Various kinds of exploits such as supply chain attacks, spearphishing, viruses, and other malware make it possible for a foothold on the developer's machine to get translated to a foothold on any infrastructure they contribute code. However, this translation can be precluded by using a secure signing method, which can ensure the veracity of the code in question.

Exploits against developers are becoming increasingly common, especially because there are now malware task forces specializing in exploiting these situations. When a malware operator gains a foothold on a new machine, they assess the value of the machine and who the main user might be. Are there credit card numbers? Are there insurance papers? Etc. Depending on the information found, that foothold then gets sold to the right kinds of people who can extract the most money out of that specific information. It is known that there exist teams that specialize in attacking developers. The turnaround of such an operation is mere hours, which could happen while the developer is asleep - sleeping hours can be inferred from git commit times.

Impact

A foothold on the developer's machine translates into an undetectable foothold on the whole infrastructure.

Recommendation

1. Review the git commit log using "git commit -p" and ensure the commits present do not contain malware

2. Re-sign all commits using the following procedure: https://andrewmatveychuk.com/how-to-sign-you-commits-with-gpg-git-and-yubikey/
3. Ensure you are never storing private keys on your developer machine, only in a Yubikey, so that even if a foothold is gained on the developer machine, it cannot expand beyond that
4. Store your private keys that Yubikey keys derive from a cold storage on a USB stick that is stored without power. Plug it in at least once a year.

Status

Unresolved

[H-11] package.json - Version constraints use ^ operator

Description

Package version constraints allow the resolution of dependencies to package versions that have not been audited, checked, or even seen by the developer.

Vulnerability Details

JavaScript libraries are a common target for supply chain attacks, therefore with highly sensitive projects that handle money, it is important to validate whether the code being provided is not malicious.

Code installed via those packages has full ability to execute on the developer's machine as the code base is being built, enabling an RCE exploit.

Since supply chain exploits on popular libraries are generally found relatively quickly, it is recommended to use a version that is at least a month old.

Impact

The developer's machine could become compromised. Ultimately (see H-09, H-10) this could translate to a compromise of the whole live infrastructure.

Recommendation

1. Use exact version number selectors in the dependencies list
2. Check the git log for exploits when choosing a new dependency or new version of an existing dependency
3. Only use versions that have been released for more than a month

Status

Unresolved

[H-12] package.json - Dependencies are not vendored

Description

Packages are not vendored, which allows for a same-day spread of supply chain exploits to the developer's machine.

Vulnerability Details

JavaScript libraries are a common target for supply chain attacks, therefore with highly sensitive projects that handle money, it is important to validate whether the code being provided is not malicious.

Code installed via those packages has full ability to execute on the developer's machine as the code base is being built, enabling an RCE exploit.

Since supply chain exploits on popular libraries are generally found relatively quickly, it is recommended to use a version that is at least a month old.

While using exact version numbers for packages is a good idea to improve security, the code being served for an existing package version can easily be replaced by code which contains malicious additions.

Impact

Developer machine could become compromised. Ultimately (see H-09, H-10) this could translate to a compromise of the whole live infrastructure.

Recommendation

1. For every dependency, check out its git code to a local disk
2. Specify dependencies to be loaded from your local disk drive using a specific commit hash by using the absolute path format
   "git+file:///path/to/git/repo#commithash" (three slashes at the beginning)
   or alternatively the relative path format
   "git+file://../../path/to/git/repo#commithash" (two slashes at the beginning).
3. Always use commit hash, and never a semver. Faking code that's at a specific commit hash requires marginally more effort (use of rainbow tables).
4. Only use commit hashes that have existed on your disk specifically for more than a month.

Status

Unresolved

Medium

[M-01] NodeConfigStorage#getAllByAddresses - Config for node at index 0 is returned if address is non-existent

Description

Function `getAllByAddresses` returns an incorrect array of configs if the input contains a non-existent address at index `i` for which `self.nodeExists(addresses[i])` is `false`. This is because `self.nodeAddressToIndx[addresses[i]] == 0`, so `nodeConfigs[i] == self.nodeIdToConfig[0]`, which is incorrect, because `self.nodeIdToConfig[0]` is not the config of node `addresses[i]`.

Recommendation

Revert if `self.nodeExists(addresses[i])` is `false` or `addresses[i]` is `address(0)`.

Status

Resolved

[M-02] NodeConfigStorage#insert/update - Inserting a duplicate signing address should not be allowed

Description

Currently the functions `insert` and `update` allow for insertion of `NodeParameters parameters` for which `parameters.signingAddress` is already in the `self.signingAddressToNodeId` mapping. This is problematic for two reasons.

1. It overrides the original value returned by `getConfigForSignerAddress`.
2. When removing one of the nodes with the signing address, future calls to `getConfigForSignerAddress` will return empty/zero on the signing address, which is not correct, because one node still remains with that signing address.

Recommendation

Revert in functions `insert` and `update` if `self.signingAddressToNodeId` contains the signing address.

Status

Resolved

[M-03] JsonFormatter#base64ToBytes - Incorrect definition of byteLength

Description

The `base64ToBytes` function seems to define the `byteLength` variable incorrectly.

Vulnerability Details

The `byteLength` is defined by `(length * 3) / 4 - padding`, but that is potentially too small. For example, if the input is `"AB=="`, then `byteLength` is `4 * 3 / 4 - 2 == 1` byte, which is not enough to store the two base 64 digits A and B, because two base 64 digits require 12 bits.

Proof of Concept

The function application

```
JsonFormatter.base64ToBytes("AB==")
```

returns the byte string `0x00`

Impact

The impact of this seems limited because only the `PermissionUpgfradeable.request` function is known to rely on the `base64ToBytes` function. It applies the `base64ToBytes` to an ED25519 public key, which presumably has a length which is a multiple of 4, for which the `byteLength` happens to be defined correctly.

Recommendation

Define the `byteLength` variable correctly. This may be as simple as not subtracting the padding, ie `byteLength = (length * 3) / 4` may work, but the right definition entirely depends

on the function PermissionUpgfradeable.request, so the implementation of base64ToBytes should be tested against PermissionUpgfradeable.request.

Status

Acknowledged

[M-04] StakingDepositUpgradeable#withdrawStakingRewards - Potentially incorrect definition of stakedCoins

Description

The variable stakedCoins is defined to be minimumStakeAmount, which is not necessarily the stake amount of msg.sender.

Vulnerability Details

The setMinimumStakeAmount function can change the value of minimumStakeAmount, which implies that defining the value of stakedCoins to be minimumStakeAmount, in the function withdrawStakingRewards, is not always the correct amount of coins staked by msg.sender.

Recommendation

Define stakedCoins with uint256 stakedCoins = deposits[nodeConfig.id].depositedAmount

Status

Unresolved

[M-05] Contracts - Mistakenly sent funds can be Stuct.

Description

Mistakenly sent ETH will be locked in the contract

Vulnerability Details

The codebase has several receive() functions without any withdrawal mechanism. This can lead to a scenario where any Ether (ETH) sent to the contract address becomes

permanently locked within the contract, as there is no method to transfer or withdraw these funds.

```
receive() external payable {}
```

There are a total of 5 receive functions in

1. TransactionRefundable.sol
2. JailedNodesUpgradeable.sol
3. NetworkConfigurationUpgradeable.sol
4. SignupBonusVestingUpgradeable.sol
5. StakingDepositUpgradeable.sol

But only the one in SignupBonusVestingUpgradeable has a modifier so if the user unintentionally sent ETH it will revert, the rest of the functions don't have anything associated with them. TransactionRefundable.sol does have an event being emitted that can be used for off-chain servers or nodes. But let's consider a scenario where a user accidentally or due to any reason sends funds to these contracts then due to the presence of the 'receive' function the contracts will receive the funds but the user will not be able to withdraw them

According to slither analyzer detection documentation
"Possible functions that receive funds with the payable attribute must have a withdraw function to secure that funds can be sent out from the function or remove payable attribute"

Recommendation

Implement a secure withdrawal function that allows the contract owner or another authorized party to transfer Ether out of the contracts.

Status

Resolved

[M-06] StakingDepositUpgradeable#slashStake - use .call to send ETH

Description

The .transfer() function can revert

Vulnerability Details

The .transfer() function intends to transfer an ETH amount with a fixed amount of 2300 gas .transfer() functions which may supply different amounts of gas in the future. Additionally, if the recipient implements a fallback function containing some sort of logic, this may inevitably revert.

```
if (slashAmount != 0) {

        payable(_msgSender()).transfer(slashAmount); // use .call
```

A good article about it

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Recommendation

Consider using .call() instead with the checks-effects-interactions pattern implemented correctly. Careful consideration needs to be made to prevent reentrancy.

Status

Resolved

Low

[L-01] TransactionRefundable#initialize - Missing onlyInitializing modifier

Description

The `initialize` function is missing an `onlyInitializing` modifier to prevent calling `initialize` multiple times. This is not currently an issue because `initialize` is overridden by `FTUAccountFunding`, but may be an issue for other parent contracts.

Recommendation

Add `onlyInitializing` modifier to the `initialize` function.

Status

Resolved

[L-02] BusinessIdentifierFactory#deployContract/changeBusinessWalletAddress - It is possible to override the existing business identifier at any business wallet address

Description

The functions `deployContract` and `changeBusinessWalletAddress` allow for updating `businessContracts`, even when there is an existing business identifier deployed. This may be undesirable since an IDP can mistakenly or purposely override the business identifier associated with a wallet.

Recommendation

Revert if `businessContracts[walletAddress]` is not `address(0)`.

Status

Resolved

[L-03] BusinessIdentifierFactory#BusinessWalletddressUpdated - Event identifier typo

Description

The event identifier contains a typo.

Recommendation

Rename from `BusinessWalletddressUpdated` to `BusinessWalletAddressUpdated`

Status

Acknowledged

[L-04] NodeRecordStorage/NodeConfigStorage#nodeExists - Address 0 is always reported as existing

Description

Function `nodeExists` returns `true` on `address(0)`, because `self.nodeAddressToIndx[0] == 0` and `self.nodeIndxToAddress[0] == 0`. We classify this as low severity, because it looks like `nodeExists` is never actually applied to `address(0)`, but may be an issue in the future.

Recommendation

Start indices at 1 instead of 0. Then `nodeExists` can be implemented by

```
function nodeExists(
    NodeRecord storage self,
    address nodeAddr
) public view returns (bool) {
    return self.nodeAddressToIndx[nodeAddr] != 0;
}
```

Alternatively, to make minimal changes, keep the current implementation, but return `false` if `nodeAddr` is `address(0)`.

Status

Acknowledged

[L-05] JailedNodesUpgradeable#freeInnocent - It is not possible to free an innocent node which was tombstoned

Description

If a node is jailed and tombstoned then it is not possible to release the node with the `freeInnocent` function because of the `notTombstoned` modifier.

Recommendation

Allow nodes to be found innocent when they are tombstoned.

Status

Acknowledged

[L-06] ReconfigurationUpgradeable#_calculateProbability - Potentially incorrect logic for calculating probability

Description

The function `_calculateProbability` is potentially returning too early when both

- `networkConfigurations[d].totalCandidates == 0` and
- `pie == 0`.

This looks like a potential error, because the value of `pie` is a value of the previous loop iteration, while `networkConfigurations[d].totalCandidates` is a value of the current loop iteration.

Recommendation

Double-check the logic. Consider whether the check

```
if (pie == 0) {
    break;
}
```

should have been at the bottom of the for-loop.

Status

Resolved

[L-07] Contracts- Important Functions lacking event emission.

Description

There are some functions in the codebase that do not have any kind of event emitting them and some of the functions are setting state variables or crucial values. An example of this is 'SetThresholdForNextUpdate' function in 'PriceFeedContract.sol'

```
function setThresholdForNextUpdate(
    uint256 newThresholdForNextUpdate
) external override onlyRedbelly {
    thresholdForNextUpdate = newThresholdForNextUpdate;
}
```

Recommendation

It is recommended to add events with the critical functions.

Status

Acknowledged

[L-08] StakingDepositUpgradeable#slashStake - Dead ether balance check

Description

Function `slashStake` seems to check for sufficient ether balance after performing the ether transfer.

Recommendation

Move the check

```
if (address(this).balance < slashAmount) {
        revert InsufficientBalance("Insufficient contract balance");
```

```
}
```

so that is happens right before the transfer

```
payable(_msgSender()).transfer(slashAmount);
```

Status

Acknowledged

[L-09] Contracts - Use _msgSender() instead of msg.sender

Description

Some contracts are using `msg.sender` rather than `_msgSender()`, which can become problematic for meta transaction support.

Recommendation

Use `_msgSender()` instead of `msg.sender`

Status

Acknowledged

[L-10] NetworkConfigurationUpgradeable - The role GOVERNOR_ROLE seems unused

Description

The `NetworkConfigurationUpgradeable` contract is granting (and revoking) the role `GOVERNOR_ROLE` to nodes that are governors. It looks like the contract directly queries the `_governors` node record storage field instead of checking for role `GOVERNOR_ROLE`.

Recommendation

Consider whether granting and revoking the role `GOVERNOR_ROLE` can be removed from the contract. Note that, in case this cannot be removed, it looks like the function `swapNodesWithIndexesAndUpdateRoles` is revoking `GOVERNOR_ROLE` of the wrong candidate.

Status

Acknowledged

[L-11] ActivityMonitorUpgradeable#isNotGovernor - Misleading modifier identifier

Description

The modifier is called `isNotGovernor`, but the effect is the opposite, to only allow governor nodes to call certain functions.

Recommendation

Rename `isNotGovernor` more appropriately, for instance, `isGovernorNodeAddress` seems more appropriate.

Status

Acknowledged

Gas

[G-01] BusinessIdentifier#removeSmartContract - Use a more efficient data structure to store smart contracts addresses

Description

The asymptotic runtime of function `removeSmartContract` is $O(n)$, where n is the length of the `smartContracts` array.

Recommendation

This can be reduced to $O(1)$ runtime by using a data structure like Open Zeppelin AddressSet for `smartContracts` instead of an array. For example

```
AddressSet smartContracts; // Declare as AddressSet
function removeSmartContract(
    address _address
```

```
) external onlyAuthorisedDelegte {
    smartContracts.remove(_address);
    emit SmartContractRemoved(_address, _msgSender());
}
```

Status

Resolved

[G-02] NodeRecordStorage/NodeConfigStorage#getAll - Function is slow when there are many nodes

Description

The asymptotic runtime and memory usage of function `getAll` are both $O(n)$, where n is the value of `self.nodeCounter`. This makes the system scale badly in relation to the number of nodes. It may become a problem if the system reaches tens of thousands of nodes.

Recommendation

Implement algorithms that use node indices 0 to $n - 1$ instead of using the array of addresses. Only access addresses when required using the function `getAddressAtIndex`.

Status

Resolved

[G-04] Contracts- Nesting if-statements is cheaper than using &&

Description

Nesting if-statements avoids the stack operations of setting up and using an extra jumpdest, and saves 6 gas, so it is recommended to use nested if-else statements to save gas.

Status

Resolved

[G-05] Contracts- Cache array length outside of loop

Description

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first)..

Status

Resolved

[G-06] Contracts- Using bools for storage incurs overhead

Description

Use uint256(1) and uint256(2) for true/false to avoid Gwarmaccess (100 gas) and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having 'true' in the past, see it here

Status

Resolved

[G-07] Contracts- Using private rather than public for constants, saves gas

Description

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves 3406-3606 gas in deployment gas due to the

compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Status

Resolved

Centralisation

The Redbelly Network project values security and utility over decentralisation.

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

Conclusion

After Hashlocks analysis, the Redbelly Network project seems to have a sound and well tested code base, however, our findings need to be resolved to achieve security. 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 are to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior 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, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyse the feasibility of an attack in a live system.

Suggested Solutions:

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

Disclaimers

Hashlock's Disclaimer

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

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

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

Technical Disclaimer

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

About Hashlock

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

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

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


#Hashlock.