Security Audit

ParagonsDAO (Staking)

July 2024 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 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

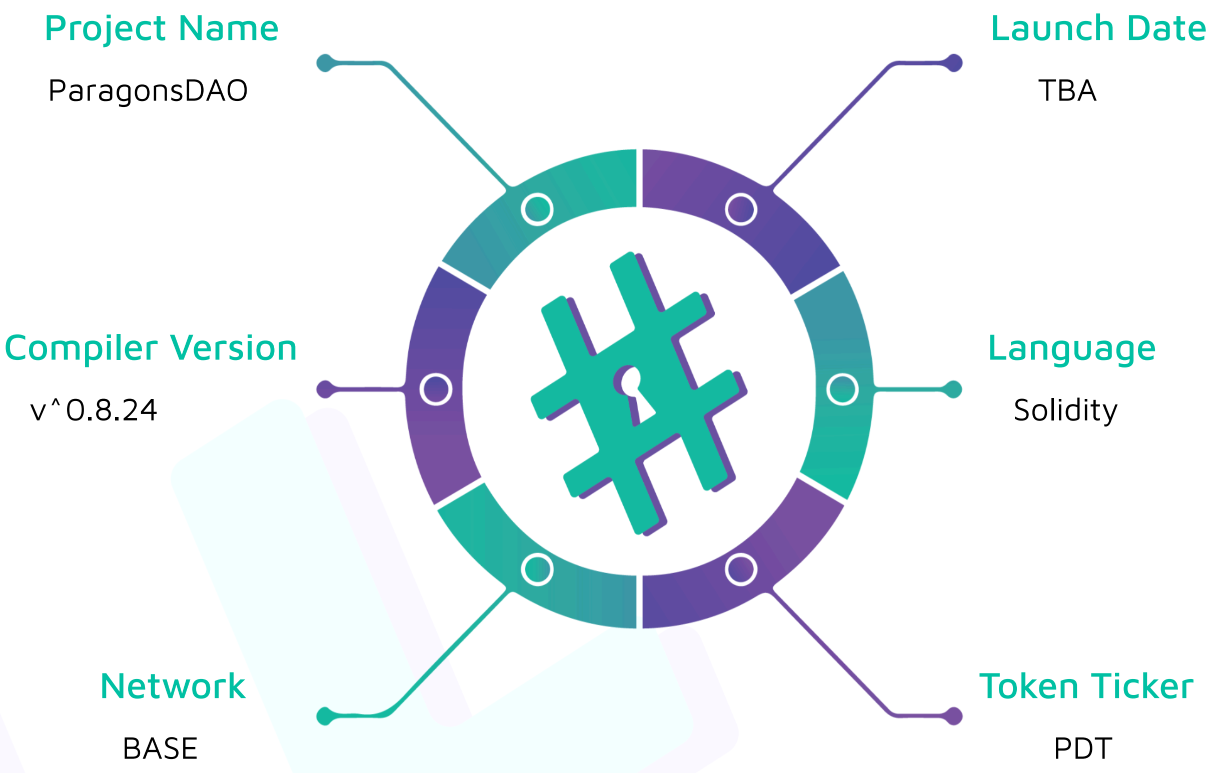
The ParagonsDAO team partnered with Hashlock to conduct a security audit of their StakedPDT.sol, IForkedPDTStakingV2.sol, IStakedPDT.sol smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

ParagonsDAO is a web3 gaming community focused on enabling players and guilds to compete and maximize their rewards through financial tools, shareable assets, edutainment, and competitive opportunities. Paragons is reducing the financial barriers of web3 gaming, to create an ecosystem where everyone wins—players, protocols, and guilds alike.

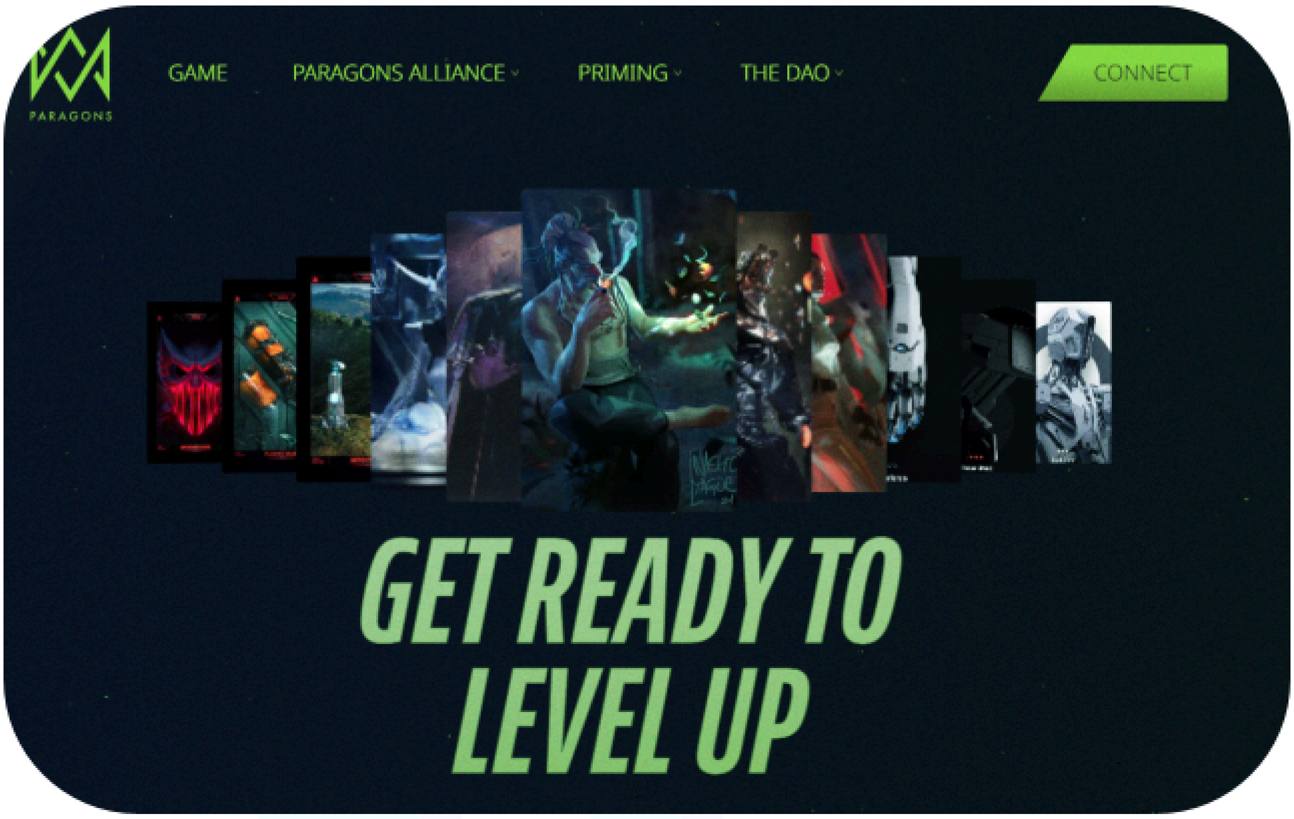
Project Name: ParagonsDAO
Compiler Version: ^0.8.24
Website: www.paragonsdao.com
Logo:

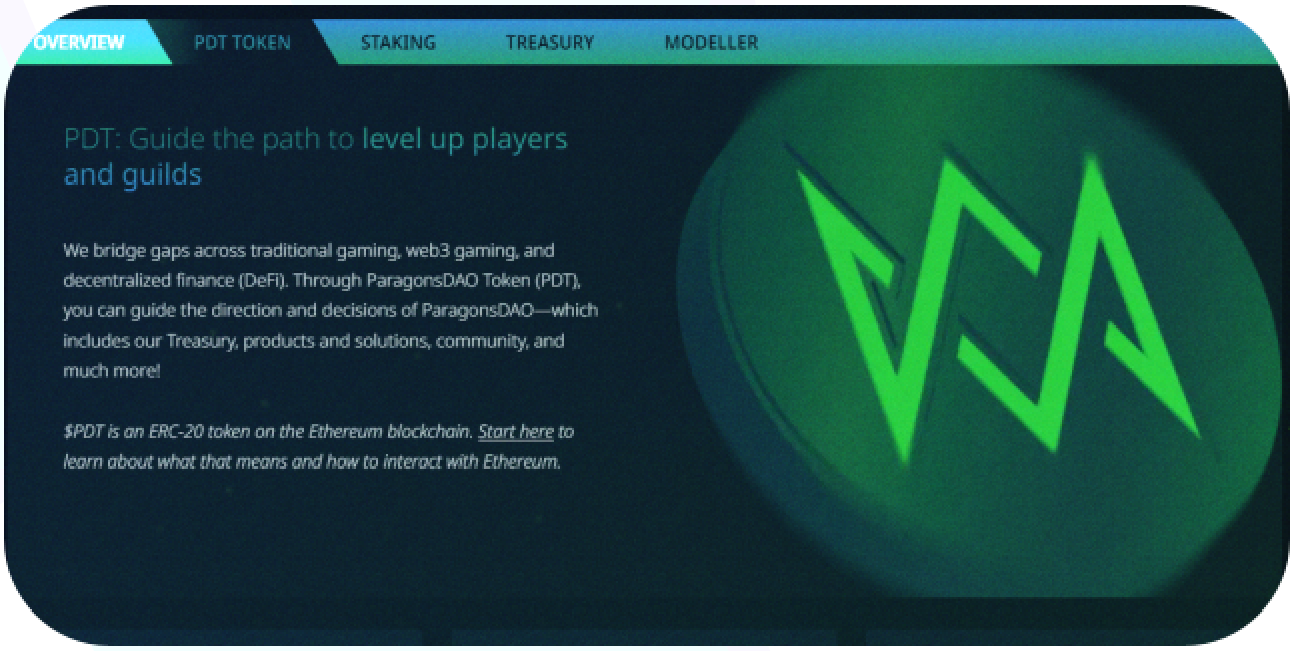
Visualised Context:



Project Name	ParagonsDAO
Launch Date	TBA
Compiler Version	v^0.8.24
Language	Solidity
Network	BASE
Token Ticker	PDT

Project Visuals:





PDT: Guide the path to level up players and guilds

We bridge gaps across traditional gaming, web3 gaming, and decentralized finance (DeFi). Through ParagonsDAO Token (PDT), you can guide the direction and decisions of ParagonsDAO—which includes our Treasury, products and solutions, community, and much more!

$PDT is an ERC-20 token on the Ethereum blockchain. Start here to learn about what that means and how to interact with Ethereum.

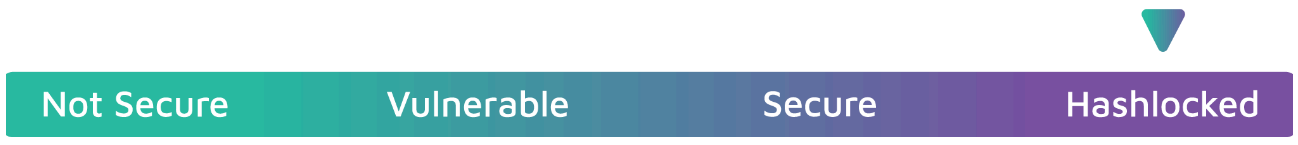
Audit scope

We at Hashlock audited the solidity code within the ParagonsDAO 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	**ParagonsDAO Protocol Smart Contracts**
Platform	**Base / Solidity**
Audit Date	**July, 2024**
Contract 1	StakedPDT.sol
Contract 1 MD5 Hash	608346ede58b91b5b8d48fdcf51a1bb9
Contract 2	IForkedPDTStakingV2.sol
Contract 2 MD5 Hash	a8426736e3d1a9a7251a5f3ac0037544
Contract 3	IStakedPDT.sol
Contract 3 MD5 Hash	a7aed75577f8432302ec7ea3582934e3
GitHub Commit Hash	5b60196206fc20ff7963a3f1a2466e18f60deac2

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.



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.

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

Hashlock found:

1 High-severity vulnerabilities

3 Medium-severity vulnerabilities

4 Low-severity vulnerabilities

5 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
StakedPDT.sol - Allows users to: - Stake PDT tokens and mint StakedPDT tokens in return - Unstake PDT tokens and burn their StakedPDT tokens - Earn rewards based on how long they have staked for - Transfer StakedPDT tokens to whitelisted addresses - Allows privileged roles to: - Update epoch length - Update which epochs will be excluded from rewards - Register new reward tokens - Move onto the next epoch - Withdraw reward tokens in the contract - Whitelist addresses to receive StakedPDT tokens	**Contract achieves this functionality.**
IForkedPDTStakingV2.sol - Interface	**Contract achieves this functionality.**
IStakedPDT.sol - Interface	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the ParagonsDAO project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices 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 ParagonsDAO project's smart contract code in the form of GitHub access.

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

Dependencies

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] StakedPDT::transfer, transferFrom - Transferring StakedPDT tokens breaks reward calculations due to not calculating weight properly

Description

The transfer and transferFrom functions allow users to transfer their StakedPDT tokens to whitelisted addresses. However, transfering staked tokens can break the reward and weight calculation.

Vulnerability Details

The transfer and transferFrom functions allow a user to send their StakedPDT token to another address, however these functions do not calculate the user weight.

```
    function transfer(address to, uint256 value) public override returns (bool) {
        if (!whitelistedContracts[to]) revert InvalidStakesTransfer();


        super._transfer(msg.sender, to, value);
        return true;
    }
      function transferFrom(address from, address to, uint256 value) public override
returns (bool) {
        if (!whitelistedContracts[to]) revert InvalidStakesTransfer();


        super._spendAllowance(from, msg.sender, value);
        super._transfer(msg.sender, to, value);
        return true;
    }
```

Consider a scenario where a user has staked at the end of an epoch, their weight is adjusted for the fact that they have staked late and their rewards will be calculated with their weight compared to the contract weight.

```
function stake(address to, uint256 amount) external nonReentrant {

        //rest of the function

        StakeDetails memory _stake = stakeDetails[to];
        uint256 _amountStaked = balanceOf(to);

        if (_stake.lastInteraction > _epoch.startTime) {
            uint256 _additionalWeight = _weightIncreaseSinceInteraction(
                block.timestamp,
                _stake.lastInteraction,
                _amountStaked
            );
            _stake.weightAtLastInteraction += _additionalWeight;
        } else {
            _stake.weightAtLastInteraction = _weightIncreaseSinceInteraction(
                block.timestamp,
                _epoch.startTime,
                _amountStaked
            );
        }
        //rest of the function
    }
```

However, when tokens are transferred, the transfer function does not consider what the user weight was. When a user receives tokens, their weight will not be calculated correctly while the contract weight is calculated correctly. This will result in combined user weights being more than the contract weight which means that users that claim first will benefit from more rewards while there won't be enough rewards left for users that claim later.

Proof of Concept

Implement and run the following test in the StakedPDT_claim.t.sol file.

```
function test_transferTokenAndClaim() public {
     uint256 POOL_SIZE = 1e18;
     uint256 initialBalance = 1e18;
     bPDTOFT.mint(staker1, initialBalance * 2);
     bPDTOFT.mint(staker2, initialBalance * 2);
     bPDTOFT.mint(staker3, initialBalance * 2);
     bPDTOFT.mint(staker4, initialBalance * 2);

     //update staker2 as whitelisted
     vm.startPrank(tokenManager);
     bStakedPDT.updateWhitelistedContract(staker2, true);
     vm.stopPrank();

     //advance to epoch1
     _creditPRIMERewardPool(POOL_SIZE);
     _moveToNextEpoch(0);

     //advance to epoch2
     _creditPRIMERewardPool(POOL_SIZE);
     _moveToNextEpoch(1);

     //staker3 and staker4 stake at epoch2 start
     (uint256 epochStartTime, uint256 epochEndTime, ) = bStakedPDT.epoch(2);
     vm.warp(epochStartTime);
```

```
        vm.startPrank(staker3);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker3, initialBalance);
        vm.stopPrank();

        vm.startPrank(staker4);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker4, initialBalance);
        vm.stopPrank();

        //staker1 stakes right before epoch end
        vm.warp(epochEndTime - 1 days);
        vm.startPrank(staker1);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker1, initialBalance);
        vm.stopPrank();

        //staker1 transfers StakedPDT to staker2
        vm.startPrank(staker1);
        bStakedPDT.transfer(staker2, initialBalance);
        vm.stopPrank();
        console.log("Contract weight:", bStakedPDT.contractWeight());
        console.log("Staker1 weight:", bStakedPDT.userTotalWeight(staker1));
        console.log("Staker2 weight:", bStakedPDT.userTotalWeight(staker2));
        console.log("Staker3 weight:", bStakedPDT.userTotalWeight(staker3));
        console.log("Staker4 weight:", bStakedPDT.userTotalWeight(staker4));

        //advance to epoch3
        _creditPRIMERewardPool(POOL_SIZE);
        _moveToNextEpoch(2);

        //all stakers claim
        vm.startPrank(staker1);
        bStakedPDT.claim(staker1);
        vm.stopPrank();

        vm.startPrank(staker2);
        bStakedPDT.claim(staker2);
        vm.stopPrank();

        vm.startPrank(staker3);
        bStakedPDT.claim(staker3);
        vm.stopPrank();

        vm.startPrank(staker4);
        bStakedPDT.claim(staker4);
        vm.stopPrank();

        //logs
        console.log("Staker1 PRIME Reward Tokens:", bPRIME.balanceOf(staker1));
        console.log("Staker2 PRIME Reward Tokens:", bPRIME.balanceOf(staker2));
        console.log("Staker3 PRIME Reward Tokens:", bPRIME.balanceOf(staker3));
        console.log("Staker4 PRIME Reward Tokens:", bPRIME.balanceOf(staker4));
    }
```

Observing the logs shown below will show that the combined user weight is greater than the contract weight and that staker4 does not receive the rewards that they are entitled to.

```
Contract weight: 54000000000000000000
Staker1 weight: 0
Staker2 weight: 27000000000000000000
Staker3 weight: 27000000000000000000
Staker4 weight: 27000000000000000000
Staker1 PRIME Reward Tokens: 0
Staker2 PRIME Reward Tokens: 491228070175438596
Staker3 PRIME Reward Tokens: 491228070175438596
```

```
    Staker4 PRIME Reward Tokens: 17543859649122808
```

Impact

Reward and weight calculation breaks, users will miss out on rewards they should get while other users will earn rewards that they shouldn't.

Recommendation

Calculate the user weight on token transfers. When a user receives tokens, their weight should be adjusted according to the weight of the token sender.

Alternatively, remove the transfer functionalities.

Status

Resolved

Medium

[M-01] StakedPDT::updateEpochLength - Function lacks proper checks for the new epoch length this can cause a temporary DoS of contract functionality.

Description

The updateEpochLength function allows the privileged EPOCH_MANAGER role to change the length of epochs. Lack of checks in this function can lead to temporary DoS, making it impossible to move to the next epoch until the length is updated again.

Vulnerability Details

The updateEpochLength function shown below updated the length of epochs.

```
    function updateEpochLength(uint256 newEpochLength) external onlyRole(EPOCH_MANAGER)
{
        require(newEpochLength > 0, "Invalid new epoch length");


        uint256 previousEpochLength = epochLength;
        epochLength = newEpochLength;
```

```
                  epoch[currentEpochId].endTime  =  epoch[currentEpochId].startTime  +
newEpochLength;
        emit UpdateEpochLength(currentEpochId, previousEpochLength, newEpochLength);
    }
```

As seen in this function, there are no checks done to make sure that the new epoch end time is greater than contractLastInteracted.

If the new epoch.endTime is less than the contractLastInteracted, when the distribute function is called, this function will make a call to the contractWeight function which will call the _weightIncreaseSinceInteraction function to calculate contract weight.

```
   function distribute() external onlyRole(EPOCH_MANAGER) {
        uint256 _currentEpochId = currentEpochId;
        Epoch memory _currentEpoch = epoch[_currentEpochId];

        if (block.timestamp >= _currentEpoch.endTime) {
            epoch[_currentEpochId].weightAtEnd = contractWeight();
       //rest of the function
    }
    function contractWeight() public view returns (uint256 contractWeight_) {
        Epoch memory _epoch = epoch[currentEpochId];
        uint256 _weightIncrease = _weightIncreaseSinceInteraction(
            Math.min(block.timestamp, _epoch.endTime),
            Math.max(contractLastInteraction, _epoch.startTime),
            totalSupply()
        );
        contractWeight_ = _weightIncrease + _contractWeight;
    }
    function _weightIncreaseSinceInteraction(
        uint256 timestamp,
        uint256 lastInteraction,
        uint256 baseAmount
    ) internal pure returns (uint256 additionalWeight_) {
        uint256 _timePassed = timestamp - lastInteraction;
        uint256 _multiplierReceived = (1e18 * _timePassed) / 1 days;

        additionalWeight_ = (baseAmount * _multiplierReceived) / 1e18;

    }
```

However, since the timestamp parameter in this function is less than the lastInteraction, this call will revert due to underflow.

Proof of Concept

Implement and run the following test in the StakedPDT_claim.t.sol file.

```
    function test_cannotMoveToNextEpoch() public {

        uint256 POOL_SIZE = 1e18;

        uint256 initialBalance = 1e18;

        bPDTOFT.mint(staker1, initialBalance * 2);

        /// EPOCH 0

        _creditPRIMERewardPool(POOL_SIZE);

        _moveToNextEpoch(0);

        // stake in epoch 1

        (uint256 epochStartTime, uint256 epochEndTime, ) = bStakedPDT.epoch(1);

        vm.warp(epochStartTime);

        // advance time

        vm.warp(epochStartTime + 2 weeks);

        //user stakes

        vm.startPrank(staker1);

        bPDTOFT.approve(bStakedPDTAddress, initialBalance);

        bStakedPDT.stake(staker1, initialBalance);

        vm.stopPrank();

        //update epoch lenght

        vm.startPrank(epochManager);

        bStakedPDT.updateEpochLength(1 weeks);

        vm.stopPrank();

        //move to next epoch

        _creditPRIMERewardPool(POOL_SIZE);

        _moveToNextEpoch(1);
    }
```

This test will fail due to an underflow error when the distribute function is called.

Impact

It will be impossible to move on to the next epoch until epoch length is updated again, causing a DoS of the contract.

Recommendation

Implement a check in the updateEpochLength function to make sure that the new epoch.endTime is greater than the contractLastInteraction.

Status

Resolved

[M-02] StakedPDT::unstake - Users that unstake at the epoch end time will lose out on rewards even if they have staked for the whole epoch length

Description

The unstake function will revert if the current block time is greater than the epoch end time. However, this function does not revert if the block time is the same as the epoch end time. Users that unstake right as the epoch ends will miss out on rewards. Additionally, these rewards won't be distributed to other stakers and will be left in the contract.

Vulnerability Details

The unstake function shown below reverts when the block.timestamp is greater than the epoch.endTime.

```
    function unstake(address to, uint256 amount) external nonReentrant {
        Epoch memory _epoch = epoch[currentEpochId];
        if (block.timestamp > _epoch.endTime) revert OutOfEpoch();
```

As seen in this function, if block.timestamp is equal to the epoch.endTime, the function will not revert. Users that staked the whole duration of the epoch can unstake just as the epoch ends, this will result in them losing out on their rewards even though they have staked the full duration of an epoch. Rewards that these users should have earned will be stuck in the contract until they are withdrawn by a privileged role.

Proof of Concept

Implement and run the following test in the StakedPDT_claim.t.sol file.

```
function test_unstakeAtEpochEnd() public {
     uint256 POOL_SIZE = 1e18;
     uint256 initialBalance = 1e18;
     bPDTOFT.mint(staker1, initialBalance);
     bPDTOFT.mint(staker2, initialBalance);

     /// EPOCH 0

     _creditPRIMERewardPool(POOL_SIZE);
     _moveToNextEpoch(0);

     /// EPOCH 1
     // users stake
     vm.startPrank(staker1);
     bPDTOFT.approve(bStakedPDTAddress, initialBalance);
     bStakedPDT.stake(staker1, initialBalance);
     vm.stopPrank();

     vm.startPrank(staker2);
     bPDTOFT.approve(bStakedPDTAddress, initialBalance);
     bStakedPDT.stake(staker2, initialBalance);
     vm.stopPrank();

     // advance time to the end of epoch 1
     (, uint256 epochEndTime, ) = bStakedPDT.epoch(1);
     vm.warp(epochEndTime);

```

```
        //staker1 unstakes at epoch end
        vm.startPrank(staker1);
        bStakedPDT.unstake(staker1, initialBalance);
        vm.stopPrank();

        // move to epoch 2
        _creditPRIMERewardPool(POOL_SIZE);
        _moveToNextEpoch(1);

        // stakers claim rewards
        vm.startPrank(staker1);
        bStakedPDT.claim(staker1);
        vm.stopPrank();

        vm.startPrank(staker2);
        bStakedPDT.claim(staker2);
        vm.stopPrank();

        // logs
        console.log("Staker1 Reward Tokens:", bPRIME.balanceOf(staker1));
        console.log("Staker2 Reward Tokens:", bPRIME.balanceOf(staker2));
                              console.log("Reward    tokens    in    the    contract:",
bPRIME.balanceOf(bStakedPDTAddress));
    }
```

Observing the logs we can see that the staker1 address has not received any rewards even though they have staked for the full epoch duration. The rewards they have earned are not distributed to other stakers and they are left in the staking contract.

```
Staker1 Reward Tokens: 0
Staker2 Reward Tokens: 5000000000000000000
Reward tokens in the contract: 15000000000000000000
```

Impact

Users that have staked for the whole epoch duration can lose out on rewards. These rewards are not distributed to other users and are left in the staking contract.

Recommendation

Change the if check as shown below in the unstake function.

```
    function unstake(address to, uint256 amount) external nonReentrant {
        Epoch memory _epoch = epoch[currentEpochId];
        if (block.timestamp >= _epoch.endTime) revert OutOfEpoch();
```

This change will make sure that the unstaking is not possible when epoch ends.

Update the contract to make sure that the rewards unstaking users should've gotten are distributed among other stakers.

Status

Resolved

[M-03] StakedPDT::updateRewardsExpiryThreshold - Updating reward expiry threshold applies to epochs that happened before the update

Description

The updateRewardsExpiryThreshold function updates the rewardsExpiryThreshold variable which changes which epochs are eligible for rewards. Users that do not expect this change will lose out on rewards when this variable is updated.

Vulnerability Details

The updateRewardsExpiryThreshold function updates the rewardsExpiryThreshold variable as shown below:

```
    function updateRewardsExpiryThreshold(
        uint256 newRewardsExpiryThreshold
     ) external onlyRole(TOKEN_MANAGER) {
        if (newRewardsExpiryThreshold == 0) revert InvalidRewardsExpiryThreshold();
        rewardsExpiryThreshold = newRewardsExpiryThreshold;
        emit UpdateRewardDuration(newRewardsExpiryThreshold);
    }
```

This function is backwards compatible, meaning that it will affect older epochs. Taking a look at where the rewardsExpiryThreshold variable is used:

```
    function claim(address to) external nonReentrant {
        _setUserWeightAtEpoch(msg.sender);

        uint256 _currentEpochId = currentEpochId;

        uint256 _claimLeftOff = claimLeftOff[msg.sender];
             if  (_claimLeftOff  ==  _currentEpochId  ||  _currentEpochId  ==  1)  revert
ClaimedUpToEpoch();

        uint256 _rewardsExpiryThreshold = rewardsExpiryThreshold;
        uint256 _startActiveEpochId = _currentEpochId > _rewardsExpiryThreshold
            ? _currentEpochId - _rewardsExpiryThreshold
            : 1;
        //rest of the function
```

This variable will make older epochs that are less than the currentEpochId - rewardsExpiryThreshold ineligible for reward claims. For example: if the currentEpochId is 25 and rewardsExpiryThreshold is 15, epochs 1-10 will not be claimable.
Having no snapshots of older epochs and what the rewardsExpiryThreshold was when users staked in these epochs will mean that this variable will apply to epochs before the update.

Proof of Concept

Implement and run the following test in the StakedPDT_claim.t.sol file.

```
    function test_updateRewardExpiryAfterStake() public {
        uint256 POOL_SIZE = 1e18;
        uint256 initialBalance = 1e18;
        bPDTOFT.mint(staker1, initialBalance * 2);
        bPDTOFT.mint(staker2, initialBalance * 2);
        bPDTOFT.mint(staker3, initialBalance * 2);

        //advance to epoch1
        _creditPRIMERewardPool(POOL_SIZE);
        _moveToNextEpoch(0);

        //staker1 stake at epoch1, reward expiry threshold is 24
        vm.startPrank(staker1);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker1, initialBalance);
        vm.stopPrank();
        //as the only staker in this epoch, staker1 should get all the epoch1 rewards.

        //advance to epoch2
        _creditPRIMERewardPool(POOL_SIZE);
        _moveToNextEpoch(1);

        //staker2 and staker3 stake at epoch2
        vm.startPrank(staker2);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker2, initialBalance);
        vm.stopPrank();

        vm.startPrank(staker3);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker3, initialBalance);
        vm.stopPrank();

        //advance to epoch3
        _creditPRIMERewardPool(POOL_SIZE);
        _moveToNextEpoch(2);

        //tokenManager updates reward expiry threshold to 1
        vm.startPrank(tokenManager);
        bStakedPDT.updateRewardsExpiryThreshold(1);
        vm.stopPrank();

        //all stakers claim
        vm.startPrank(staker1);
        bStakedPDT.claim(staker1);
        vm.stopPrank();

        vm.startPrank(staker2);
        bStakedPDT.claim(staker2);
        vm.stopPrank();

        vm.startPrank(staker3);
        bStakedPDT.claim(staker3);
        vm.stopPrank();

        //showing that all claimed rewards are equal
        uint256 staker1Rewards = bPRIME.balanceOf(staker1);
        uint256 staker2Rewards = bPRIME.balanceOf(staker2);
        uint256 staker3Rewards = bPRIME.balanceOf(staker3);
        assertEq(staker1Rewards, staker2Rewards);
        assertEq(staker1Rewards, staker3Rewards);
    }
```

This test will pass, proving that staker1 that staked when the rewardsExpiryThreshold was 24, lost their rewards for epoch1 when rewardsExpiryThreshold was updated to 1.

Impact

Users that have staked before the rewardsExpiryThreshold change, will lose out on rewards since they do not expect this change.

Additionally this introduces a centralization risk, malicious or careless tokenManager can cause the users to lose out on rewards.

Recommendation

Keep track of what the rewardsExpiryThreshold was when that epoch happened, in order to not make this variable backwards compatible. rewardsExpiryThreshold change should not apply to epochs that have happened before the change.

Note:

The ParagondsDAO team acknowledges that if, through a successful governance proposal, their community decides to shorten the rewards expiry threshold, they would need to provide sufficient lead time and do a large communication push to ensure stakers with unclaimed rewards from previous epochs claim them before the change is implemented.

For any stakers who did not claim their rewards before the change was implemented, the rewards they earned and did not claim beyond the new threshold would be considered expired.

Status

Acknowledged

Low

[L-01] StakedPDT::withdrawRewardTokens - Centralization risk for privileged function

Description

The withdrawRewardTokens function allows the privileged tokenManager role to withdraw any amount of reward tokens from the contract.

```
    /**
     * @notice Withdraw idle reward tokens. Idle reward amount
     * should be calculated from off-chain side.
     * @param rewardToken The address of the reward token
     * @param amount The amount of the reward tokens to withdraw
     *
     * Requirements:
     *
     * - Only TOKEN_MANAGER can withdraw reward tokens
     * - `rewardToken` should be already registered
     * - `amount` shouldn't be zero
     *
     * Emits a {WithdrawRewardToken} event.
     */
    function withdrawRewardTokens(
        address rewardToken,
        uint256 amount
    ) external onlyRole(TOKEN_MANAGER) {
        if (rewardToken == address(0)) {
            revert InvalidRewardToken();
        }
        if (amount == 0) {
            revert InvalidWithdrawAmount();
        }

        address[] memory _tokenList = rewardTokenList;
        uint256 _tokenListSize = _tokenList.length;
        uint8 isRegistered = 0;

        for (uint256 itTokenIndex; itTokenIndex < _tokenListSize; ) {
            if (_tokenList[itTokenIndex] == rewardToken) {
                isRegistered = 1;
                break;
            }

            unchecked {
                ++itTokenIndex;
            }
        }
        if (isRegistered == 0) {
            revert InvalidRewardToken();
        }

        IERC20(rewardToken).safeTransfer(msg.sender, amount);

         emit WithdrawRewardToken(rewardToken, amount);
    }
```

Comments state that the amount that will be withdrawn is calculated off-chain. However this still introduces a centralization risk.
Off-chain system should never calculate the incorrect amount.
Malicious or careless tokenManager role can withdraw any amount of tokens from the contract which can result in users losing out on rewards they have earned.

Recommendation

Implement the correct idle reward calculation in the function itself or make sure that the off-chain calculation will never be incorrect.

Status

Resolved

[L-02] StakedPDT - Users' tokens will be stuck in the contract if privileged role does not manually start a new epoch

Description

The unstake function does not allow users to unstake their PDT tokens if epoch has ended:

```
    function unstake(address to, uint256 amount) external nonReentrant {
        Epoch memory _epoch = epoch[currentEpochId];
        if (block.timestamp > _epoch.endTime) revert OutOfEpoch();
        //rest of the function
```

This is done to make sure that users do not lose out on their rewards in case they unstake after an epoch ends and another one starts. However, this means that staked tokens will be stuck in the contract if the EPOCH_MANAGER role does not call the distribute function to manually move onto the next epoch.

This is a useability issue and a centralization risk.

Recommendation

Implement a mechanism that keeps track if users have staked till the end of an epoch and make them eligible for rewards according to this mechanism. This way the function can be modified to allow users to stake/unstake after an epoch ends and before a new one starts manually.

Status

Acknowledged

ParagonsDAO Team Comment:
This is working as designed.
We will design a script that begins the new epoch automatically.

[L-03] StakedPDT - Contract implementation does not match the documentation

Description

The documents given for the StakedPDT.sol state that:

- `Rewards can be claimed at the end of each epoch using an epoch_id`

However, the claim function does not take any epoch_id parameter from the user, instead it calculates the rewards for all epochs.

- `Up to two different ERC20 tokens can be distributed as rewards in a single epoch.`

More than two different ERC20 tokens can be distributed as rewards in a single epoch.

Recommendation

Update the documentation or the contract as it is intended to avoid mismatch.

Status

Resolved

[L-04] StakedPDT - No token locking mechanism allows users to receive rewards even if they staked for 1 block

Description

In the staking mechanism, there is no lock for users' staked tokens and users are eligible for rewards no matter when in epoch they have staked. This will cause users to stake right before an epoch ends and unstake right after epoch ends, benefiting from rewards.

Vulnerability Details

The rewards in the StakedPDT.sol contract are sent to the contract at each epoch's start. Users that stake during an epoch earn rewards based on how long they have staked in that epoch for.

```
    function stake(address to, uint256 amount) external nonReentrant {
        Epoch memory _epoch = epoch[currentEpochId];

        if (block.timestamp > _epoch.endTime) {
            revert OutOfEpoch();
        }
        if (amount == 0) {
            revert InvalidStakeAmount();
        }

        _setUserWeightAtEpoch(to);
        _adjustContractWeight();

        StakeDetails memory _stake = stakeDetails[to];
        uint256 _amountStaked = balanceOf(to);

        if (_stake.lastInteraction > _epoch.startTime) {
            uint256 _additionalWeight = _weightIncreaseSinceInteraction(
                block.timestamp,
                _stake.lastInteraction,
                _amountStaked
            );
            _stake.weightAtLastInteraction += _additionalWeight;
        } else {
            _stake.weightAtLastInteraction = _weightIncreaseSinceInteraction(
                block.timestamp,
                _epoch.startTime,
                _amountStaked
            );
        }

        _stake.lastInteraction = block.timestamp;
        stakeDetails[to] = _stake;

        _mint(to, amount);
        IERC20(pdt).safeTransferFrom(msg.sender, address(this), amount);

        emit Stake(to, amount, currentEpochId);
    }
```

Taking a look at the unstake function:

```
    function unstake(address to, uint256 amount) external nonReentrant {
        Epoch memory _epoch = epoch[currentEpochId];
        if (block.timestamp > _epoch.endTime) revert OutOfEpoch();

        uint256 _amountStaked = balanceOf(msg.sender);
        if (amount == 0 || amount > _amountStaked) revert InvalidUnstakeAmount();
        _setUserWeightAtEpoch(msg.sender);
        _adjustContractWeight();

        StakeDetails memory _stake = stakeDetails[msg.sender];

        if (_stake.lastInteraction > _epoch.startTime) {
            uint256 _additionalWeight = _weightIncreaseSinceInteraction(
                block.timestamp,
                _stake.lastInteraction,
                _amountStaked
            );
            _stake.weightAtLastInteraction += _additionalWeight;
        } else {
            _stake.weightAtLastInteraction = _weightIncreaseSinceInteraction(
                block.timestamp,
                _epoch.startTime,
                _amountStaked
            );
        }

        _stake.lastInteraction = block.timestamp;
        stakeDetails[msg.sender] = _stake;

        _burn(msg.sender, amount);
        IERC20(pdt).safeTransfer(to, amount);

        emit Unstake(msg.sender, amount, currentEpochId);
    }
```

We can observe that users can unstake their tokens at any point within an epoch. This allows users to stake right before an epoch ends, unstake right after the epoch ends and claim rewards.

Proof of Concept

Implement and run the following test in the StakedPDT_claim.t.sol file.

```
function test_StepwiseJumpOfRewards() public {
      uint256 POOL_SIZE = 1e18;
      uint256 initialBalance = 1e18;
      bPDTOFT.mint(staker1, initialBalance * 100);
      bPDTOFT.mint(staker2, initialBalance * 2);
      bPDTOFT.mint(staker3, initialBalance * 2);
      bPDTOFT.mint(staker4, initialBalance * 2);

      /// EPOCH 0

      _creditPRIMERewardPool(POOL_SIZE);
      _moveToNextEpoch(0);

      /// EPOCH 1

      _creditPRIMERewardPool(POOL_SIZE);
      _moveToNextEpoch(1);

      //EPOCH 2
      //staker2, 3 and 4 stake in start of epoch2
      (uint256 epochStartTime2, uint256 epochEndTime2, ) = bStakedPDT.epoch(2);
      vm.warp(epochStartTime2);

      vm.startPrank(staker2);
      bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
      bStakedPDT.stake(staker2, initialBalance);
      vm.stopPrank();

      vm.startPrank(staker3);
      bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
      bStakedPDT.stake(staker3, initialBalance);
      vm.stopPrank();
```

```
        vm.startPrank(staker4);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 2);
        bStakedPDT.stake(staker4, initialBalance);
        vm.stopPrank();

        //staker1(whale) stakes right before epoch2 ends
        uint256 staker1BalanceBefore = bPDTOFT.balanceOf(staker1);
        vm.warp(epochEndTime2 - 1);
        vm.startPrank(staker1);
        bPDTOFT.approve(bStakedPDTAddress, initialBalance * 100);
        bStakedPDT.stake(staker1, initialBalance * 100);
        vm.stopPrank();

        //move to epoch3
        _creditPRIMERewardPool(POOL_SIZE);
        _moveToNextEpoch(2);

        // EPOCH 3

        vm.startPrank(staker1);
        bStakedPDT.unstake(staker1, initialBalance * 100);
        vm.stopPrank();
        uint256 staker1BalanceAfter = bPDTOFT.balanceOf(staker1);

        vm.startPrank(staker1);
        bStakedPDT.claim(staker1);
        vm.stopPrank();

        vm.startPrank(staker2);
        bStakedPDT.claim(staker2);
        vm.stopPrank();

        vm.startPrank(staker3);
        bStakedPDT.claim(staker3);
        vm.stopPrank();

        vm.startPrank(staker4);
        bStakedPDT.claim(staker4);
        vm.stopPrank();

        assertEq(staker1BalanceBefore, staker1BalanceAfter);
        assertGt(bPRIME.balanceOf(staker1), 0);
        console.log("Staker1 PRIME Reward Tokens:", bPRIME.balanceOf(staker1));
        console.log("Staker2 PRIME Reward Tokens:", bPRIME.balanceOf(staker2));
        console.log("Staker3 PRIME Reward Tokens:", bPRIME.balanceOf(staker3));
        console.log("Staker3 PRIME Reward Tokens:", bPRIME.balanceOf(staker4));
    }
```

Observing the logs, staker1 has staked for only 1 block and received rewards while diluting the reward pool for other users.

```
Staker1 PRIME Reward Tokens: 13778469763148
Staker2 PRIME Reward Tokens: 333328740510078950
Staker3 PRIME Reward Tokens: 333328740510078950
Staker3 PRIME Reward Tokens: 333328740510078950
```

Impact

Users that have staked for only 1 block can benefit from rewards, diluting the reward pool for honest stakers.

Recommendation

Do not let users stake at the end of epochs, for example if there’s 3 days left for an epoch end, revert stake calls.

Alternatively, introduce a token locking mechanism that does not allow unstaking before a certain time has passed since staking.

Status

Acknowledged

ParagonsDAO Team Comment:

This is working as designed.

A staker cannot earn more than they are eligible for in an epoch. This is controlled by a daily multiplier.

Gas

[G-01] StakedPDT::unstake - Unnecessary revert can be avoided

Description

The highlighted part in the function shown below causes the function to revert if the amount entered by the user is greater than their balance.

```
function unstake(address to, uint256 amount) external nonReentrant {

    //rest of the function

    uint256 _amountStaked = balanceOf(msg.sender);

    if (amount == 0 || amount > _amountStaked) revert InvalidUnstakeAmount();

    //rest of the function
```

This can be modified in order for the function to not revert.

Recommendation

Update the function as shown below so that the amount will be set to users balance if inputted amount is too big:

```
function unstake(address to, uint256 amount) external nonReentrant {

    //rest of the function

    uint256 _amountStaked = balanceOf(msg.sender);
```

```
        if (amount == 0) revert InvalidUnstakeAmount();

        if (amount > _amountStaked){

            amount = _amountStaked; }

        //rest of the function
```

Status

Acknowledged

Paragons DAO Team Comment:

Controls for “max” withdrawal will be implemented on front-end.

[G-02] StakedPDT - Prevent setting a state variable with the same value

Description

Not only is wasteful in terms of gas, but this is especially problematic when an event is emitted and the old and new values set are the same, as listeners might not expect this kind of scenario. Functions shown below can be updated to revert if the updated value is the same as the existing value.

```
function updateEpochLength()

function updateRewardsExpiryThreshold()

function updateWhitelistedContract()
```

Recommendation

Implement checks in these functions to avoid setting state variables to their existing values. Prior to updating a state variable, compare the new value with the current value and proceed with the assignment only if they differ. Additionally, ensure that events related to state variable updates are emitted only when actual changes occur. This approach not only saves gas but also prevents confusion and unnecessary triggers in event listeners.

Status

Resolved

[G-03] IForkedPDTStakingV2 - Event declared but not emitted

Description

An event within the contract is declared but not utilised in any of the contract's functions or operations. Having unused event declarations can consume unnecessary space and may lead to misunderstandings for developers or users expecting this event as part of the contract's functionality.

```
event PushBackEpoch0()

event UpsertRewardToken()

event TransferStakes()
```

Recommendation

Consider removing the unused event declaration to optimise the contract and enhance clarity. If there is an intent for this event to be part of certain operations, ensure it is emitted appropriately. Otherwise, for the sake of clean and efficient code, it is advisable to remove any unused declarations.

Status

Resolved

[G-04] StakedPDT - State variables that are used multiple times in a function should be cached

Description

When performing multiple operations on a state variable in a function, it is recommended to cache it first. Multiple reads to a state variable can save gas by caching it.

```
function updateEpochLength() //State variable `currentEpochId` is used multiple times
function stake() //State variable `currentEpochId` is used multiple times

function unstake() //State variable `currentEpochId` is used multiple times
```

Recommendation

Cache state variables in stack or local memory variables within functions when they are used multiple times.

Status

Resolved

[G-05] StakedPDT - Automated unchecked arithmetic is generated after solidity version 0.8.22

Description

It is not needed to add the unchecked block to for loop expressions after Solidity version 0.8.22.

According to Solidity documentation: Solidity 0.8.22 introduces an overflow check optimization that automatically generates an unchecked arithmetic increment of the counter of for loops. This new optimization removes the need for poor unchecked increment patterns in for loop bodies

Recommendation

Do not use the unchecked keyword for loop expressions after Solidity version 0.8.22.

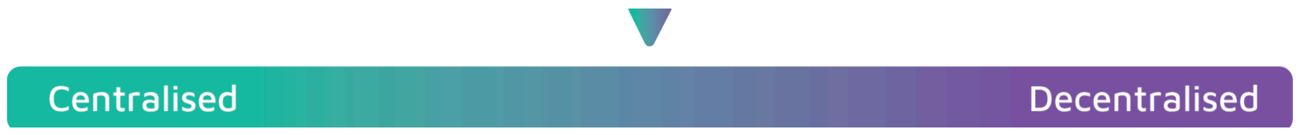
Status

Resolved

Centralisation

The ParagonsDAO 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 ParagonsDAO 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 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 still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to 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 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.

Hashlock.