BLOCKLORDS
Audit Report

Wed Nov 20 2024

contact@bitslab.xyz https://twitter.com/scalebit_

ScaleBit

BLOCKLORDS Audit Report

1 Executive Summary

1.1 Project Information

Description	BLOCKLORDS is a player-driven MMO medieval grand strategy game where your decisions and skills shape the world and narrative
Type	Game
Auditors	ScaleBit
Timeline	Thu Nov 07 2024 - Wed Nov 20 2024
Languages	Solidity
Platform	Base
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/blocklords/lordchain-smartcontracts
Commits	b7d38514827e86d870544293dee27414bd4305f9 3cc14ddd1cd2ff2dd78bdb811ce3c242322a8107 c0c6b75420d53627556db92e0de4acc84ce4e46f a27ef7e75966a956d772df19490f72add88f8c62

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
VAL	contracts/Validator.sol	879d4e75973d88daed53c364b10befddc6681402
VFA	contracts/ValidatorFactory.sol	c8789af8ea00cdeca3916176772b6b7c23f9ec74
IVF	contracts/interfaces/IValidatorFactory.sol	6180e79caf279315ab9a0014b17e1cd9736e3ca7
IVA	contracts/interfaces/IValidator.sol	dea2e48f39cb880a1b4cdd2a948caa8f96b8ac45
IGO	contracts/interfaces/IGovernance.sol	3b3326bf4e052ec5cbc186f5b761606f7c8f2b2e
GOV	contracts/Governance.sol	e89f2e0b6be3fe242794057377d1798e24c85933
VFE	contracts/ValidatorFees.sol	c0c815e2a30e10d20a30c442736c65c3f00f5776

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	24	24	0
Informational	2	2	0
Minor	11	11	0
Medium	4	4	0
Major	7	7	0
Critical	0	0	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow
- Number of rounding errors
- Unchecked External Call
- Unchecked CALL Return Values
- Functionality Checks
- Reentrancy
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic issues
- Gas usage
- Fallback function usage
- tx.origin authentication
- Replay attacks
- Coding style issues

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by BLOCKLORDS to identify any potential issues and vulnerabilities in the source code of the BLOCKLORDS smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 24 issues of varying severity, listed below.

ID	Title	Severity	Status
GOV-1	The User can Claim Extra Boost Rewards	Major	Fixed
GOV-2	Incorrect Status Handling for Boost Proposals in `executeVoteRewardProposal` Function	Medium	Fixed
GOV-3	Single-step Ownership Transfer Can be Dangerous	Medium	Fixed
GOV-4	The Two-step Ownership Transfer Implementation Is Incorrect	Medium	Fixed
GOV-5	Inconsistent Proposal ID Generation in `createBoostPropose` Functions	Minor	Fixed
GOV-6	Invalid Fee Check in `setDepositFee` and `setClaimFee` Functions	Minor	Fixed
GOV-7	Missing Status Validation for Voting	Minor	Fixed
VAL-1	Lack of Access Control	Major	Fixed

VAL-2	The User's Rewards will be Lost	Major	Fixed
VAL-3	Due to an Overflow, the User is unable to Claim Rewards	Major	Fixed
VAL-4	The User can Repeatedly Claim Rewards from Previous Periods	Major	Fixed
VAL-5	The User's Voting Rewards will also be Included in the Calculation of Period Rewards	Major	Fixed
VAL-6	Missing BoostReward Distribution in `withdraw` Function	Major	Fixed
VAL-7	Possible Inability to Claim Fees	Medium	Fixed
VAL-8	Incorrect Condition in Quality Check	Minor	Fixed
VAL-9	Unnecessary Update of User Reward Debt	Minor	Fixed
VFA-1	Incorrect Condition in `SubTotalStakedWallet` Function Leading to Underflow Error	Minor	Fixed
VFA-2	Missing Check for `_endTime > _startTime` in AddTotalValidators Function	Minor	Fixed
VFA-3	Ineffective Condition in `createValidator` Function	Minor	Fixed
VFA-4	Invalid Global Variables	Minor	Fixed
VFA-5	Missing Zero Address Check in setVoter Function	Informational	Fixed

VAL-10	Updating the User's Debt after Deleting the User's Information is Meaningless	Minor	Fixed
VAL-11	Incorrect Withdraw Event Logs Due to Reset User Data	Minor	Fixed
VAL-12	Mismatch Between `MAX_LOCK` Value and Commented Description of Lock Duration	Informational	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the BLOCKLORDS Smart Contract :

Admin

- **createPropose**: Creates a new proposal for votin.
- **createBoostPropose**: Creates a new boost proposal for validators.
- **addBoostReward**: Distributes the boost rewards to the validators based on their votes.
- **resetVotes**: reset votes for users.
- **cancelProposal**: Cancels a proposal by marking it as cancelled.
- **cancelBoostProposal**: Cancels a boost proposal by marking it as cancelled.
- **setVoteReward**: Sets the reward token and reward amount for a given proposal.
- **executeVoteRewardProposal**: Executes the reward distribution for a given proposal based on the vote weight.
- **setRewardPeriod**: Configures a new reward period.
- **setName**: Updates the validator’s name.
- **setVerifier**: Sets the verifier address responsible for signature verification.
- **setMasterValidator**: Assigns the master validator's address for controlling validator functions.
- **setVoter**: Defines the address of the governance voter.
- **claimFees**: Claims accumulated fees in the contract on behalf of the admin.
- **setDepositFee**: Adjusts the deposit fee percentage, constrained by the maximum allowable limit.
- **setClaimFee**: Adjusts the claim fee percentage, up to the maximum allowed limit.
- **setPauseState**: Toggles the contract’s pause state, allowing or restricting certain operations as a safety measure.

User

- **vote**: Users can vote for a given proposal.
- **getUserVotesForAllChoices**: Retrieves the number of votes cast by a user for all available choices in a specific proposal.
- **getProposalOptionVotes**: Retrieves the total number of votes cast for each available choice in a specific proposal.
- **createLock**: Starts a staking lock with a specific amount and duration, enabling the user to earn rewards.
- **increaseAmount**: Adds to the user's staked amount within an active lock.
- **extendDuration**: Extends the duration of an existing lock period, allowing continued participation in staking rewards.
- **claim**: Claims any pending rewards based on the user's staked amount.
- **withdraw**: Withdraws the staked tokens once the lock period has expired.
- **setAutoMax**: Enables or disables the `autoMax` feature, which automatically sets the lock duration to the maximum allowed period.

4 Findings

GOV-1 The User can Claim Extra Boost Rewards

Severity: Major

Status: Fixed

Code Location:

contracts/Governance.sol#767-795

Descriptions:

The user can call the `claimBoostReward()` function to claim the boost reward.

```
function claimBoostReward() external nonReentrant whenNotPaused {
        UserInfo storage user = userInfo[msg.sender];

        uint256 totalBoostPending = _calculateBoostPending(user);

        if (totalBoostPending <= 0) revert InvalidBoostReward();

        // Transfer the total pending boost reward to the user
        IERC20(token).transfer(msg.sender, totalBoostPending);

        uint256 totalBoostDebt = 0;

        // Loop through each reward period from the user's last updated period to the
current
        for (uint256 i = 0; i < currentBoostRewardPeriodIndex; i++) {
            BoostReward storage boost = boostRewards[i];

            if(!IGovernance(governance).isBoostVote(i)) continue ;

            if (block.timestamp < boost.startTime) {
                break;
            }

            totalBoostDebt += (user.amount * boost.accTokenPerShare) / PRECISION_FACTOR;
        }

        boostRewardDebt[msg.sender] = totalBoostDebt;
```

```
    emit BoostRewardClaimed(msg.sender, totalBoostPending);
}
```

There is an issue where if the user deposits 100 in Period 1, then another 100 in Period 10, and calls `claimBoostReward()`, they are able to claim all boost rewards from Period 1 to Period 10.

Suggestion:

It is recommended to distribute the boost rewards to the user at the time of claiming rewards.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

GOV-2 Incorrect Status Handling for Boost Proposals in `executeVoteRewardProposal` Function

Severity: Medium

Status: Fixed

Code Location:

contracts/Governance.sol#472,535

Descriptions:

```
// Check the proposal status
if (proposals[_proposalId].status != FinalizationStatus.Pending) {
    revert("Proposal is not in Pending status");
}

// ... rest of the function logic ...

// Update the proposal status to Executed
proposals[_proposalId].status = FinalizationStatus.Executed;
```

Currently, the function only checks and updates the status of regular proposals using `proposals[_proposalId].status`. For boost proposals, whose statuses are stored separately in `boostProposals[_proposalId].status`, this check and update are not performed. As a result, after executing the reward distribution for a boost proposal, its status remains Pending, which can lead to inconsistencies in the contract's state.

Suggestion:

1. Check the Correct Status Field:

- For regular proposals, continue to check proposals[_proposalId].status.
- For boost proposals, check boostProposals[_proposalId].status.

2. Update the Correct Status Field After Execution:

- For regular proposals, update proposals[_proposalId].status to FinalizationStatus.Executed.

- For boost proposals, update boostProposals[_proposalId].status to FinalizationStatus.Executed.

GOV-3 Single-step Ownership Transfer Can be Dangerous

Severity: Medium

Status: Fixed

Code Location:

contracts/Governance.sol#5

Descriptions:

Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. If the admin permissions are given to the wrong address within this function, it will cause irreparable damage to the contract. Below is the official documentation explanation from OpenZeppelin https://docs.openzeppelin.com/contracts/4.x/api/access

`Ownable is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.`

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

Suggestion:

It is recommended to use a two-step ownership transfer pattern.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

GOV-4 The Two-step Ownership Transfer Implementation Is Incorrect

Severity: Medium

Status: Fixed

Code Location:

contracts/Governance.sol#279-288

Descriptions:

In the `acceptOwnership` function, `transferOwnership` is called to transfer ownership, but it should actually call `_transferOwnership` (OpenZeppelin Ownable.sol link) instead of `transferOwnership`.

```
function acceptOwnership() external nonReentrant {
    // Ensure that only the nominated address can accept ownership
    if (msg.sender != newOwner)  revert notNominatedAddress();

    // Transfer ownership to the nominated address
    transferOwnership(newOwner);

    // Reset the nominated address after transfer
    newOwner = address(0);
}
```

```
function transferOwnership(address _newOwner) public override  onlyOwner {
    if (_newOwner == address(0)) revert ZeroAddress();
    newOwner = _newOwner;
}
```

Suggestion:

It's recommended to refer to the implementation in OpenZeppelin's Ownable2Step.sol.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

GOV-5 Inconsistent Proposal ID Generation in `createBoostPropose` Functions

Severity: Minor

Status: Fixed

Code Location:

contracts/Governance.sol#128

Descriptions:

There is an inconsistency in the increment method of proposalCount between the `createPropose` and `createBoostPropose` functions, leading to potential discrepancies in proposal ID generation:

In the `createPropose` function:

```
uint256 proposalId = proposalCount++;
```

In the `createBoostPropose` function:

```
uint256 proposalId = ++proposalCount;
```

Since `proposalCount++` and `++proposalCount` increment operations differ in their order of execution, this discrepancy could result in inconsistent proposal IDs, potentially leading to tracking issues or unexpected behavior.

Suggestion:

To ensure consistent proposal ID generation, it is recom `proposalCount++` for both functions is advisable to maintain consistency.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

GOV-6 Invalid Fee Check in `setDepositFee` and `setClaimFee` Functions

Severity: Minor

Status: Fixed

Code Location:

contracts/Governance.sol#846,857

Descriptions:

```
if (_fee < 0) revert WrongFee();
```

In both the `setDepositFee` and `setClaimFee` functions, there is a check for `_fee < 0` to validate the input fee. However, since `_fee` is of type `uint256`, it can never be negative, making the condition `_fee < 0` redundant and ineffective.

Suggestion:

Remove the _fee < 0 check in both functions.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

GOV-7 Missing Status Validation for Voting

Severity: Minor

Status: Fixed

Code Location:

contracts/Governance.sol#167

Descriptions:

Voting should only be allowed when the proposal is in a pending state, but the contract does not perform this validation.

```
function vote(uint256 _proposalId, uint256[] calldata _choiceIds, uint256[] calldata
_weights) external nonReentrant {
        // Check if choiceIds and weights lengths match
        if (_choiceIds.length != _weights.length) revert UnequalLengths();


        // Declare the Proposal storage variable here after checking the type of proposal
        if (isBoostVote[_proposalId]) {
            // Boost proposal voting logic
            ValidatorBoostProposal storage boostProposal = boostProposals[_proposalId];

            // Common check for voting period
            _checkVotingPeriod(boostProposal.startTime, boostProposal.endTime);

            _vote(_proposalId, _choiceIds, _weights, proposalValidatorCounts[_proposalId],
true);
        } else {
            // Regular proposal voting logic
            Proposal storage proposal = proposals[_proposalId];

            // Common check for voting period
            _checkVotingPeriod(proposal.startTime, proposal.endTime);

            _vote(_proposalId, _choiceIds, _weights, proposal.totalChoices, false);
        }
    }
```

Suggestion:

It is recommended to add status validation to ensure voting is only allowed for pending proposals.

Resolution:

This issue has been fixed. The client has added status validation to ensure voting is only allowed for pending proposals.

VAL-1 Lack of Access Control

Severity: Major

Status: Fixed

Code Location:

contracts/Validator.sol#154-196

Descriptions:

The `createValidator` function in the ValidatorFactory contract lacks any access control. Is access control needed for this function?

```
    function createValidator(address _owner, uint256 _quality, address _verifier) public returns (address validator) {
        uint256 validatorId = allValidators.length;  // Use the length of allValidators array as the validatorId

        bytes32 salt = keccak256(abi.encodePacked(_quality, _owner, validatorId)); // salt includes stable as well, 3 parameters

        validator = Clones.cloneDeterministic(implementation, salt);

        IValidator(validator).initialize(msg.sender, _owner, validatorId, _quality, _verifier);

        allValidators.push(validator);

        _isValidator[validator] = true;

        emit ValidatorCreated(_owner, validator, allValidators.length);
    }
```

Suggestion:

It is recommended to implement access control.

Resolution:

This issue has been fixed. The client has implemented access control.

VAL-2 The User's Rewards will be Lost

Severity: Major

Status: Fixed

Code Location:

contracts/Validator.sol#506

Descriptions:

In the `_deposit()` function, the protocol updates the user's reward debt as follows:

```
_user.rewardDebt = (_user.amount *
rewardPeriods[getCurrentPeriod()].accTokenPerShare) / PRECISION_FACTOR;
```

However, before updating the user's `rewardDebt`, the protocol does not distribute the rewards previously earned by the user. This results in the loss of rewards.

For example, if the `lockDuration` is 30 days and the `RewardPeriod` is also 30 days:

1. On Day 1, the user deposits 200 tokens. `user.rewardDebt = 200 * accTokenPerShare`.
2. After 15 days, `accTokenPerShare` increases to `accTokenPerShare1`. The user then deposits an additional 300 tokens.
3. The protocol calculates `user.rewardDebt` as `(200 + 300) * accTokenPerShare1`.

In this calculation, the rewards accumulated during the first 15 days are effectively lost, as the protocol does not account for them before recalculating `user.rewardDebt`.

Suggestion:

It is recommended to distribute the user's previously accrued rewards before updating their `rewardDebt`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VAL-3 Due to an Overflow, the User is unable to Claim Rewards

Severity: Major

Status: Fixed

Code Location:

contracts/Validator.sol#672

Descriptions:

In the `claim()` function, the protocol calls the `_updateUserRewards()` function to update `_user.rewardDebt`, setting it to `accumulatedRewards`.

```
  function _updateUserRewards(UserInfo storage _user) internal {
    uint256 accumulatedRewards = _user.rewardDebt;

    // Loop through each reward period from the user's last updated period to the
current
    for (uint256 i = _user.lastUpdatedRewardPeriod; i < currentRewardPeriodIndex; i++)
{
      RewardPeriod memory period = rewardPeriods[i];

      if (period.endTime <= block.timestamp) {
        accumulatedRewards += (_user.amount * period.accTokenPerShare) /
PRECISION_FACTOR;
      }
    }

    _user.rewardDebt = accumulatedRewards;
  }
```

In the `_calculateTotalPending()` function, the protocol calculates the rewards for each period, then uses the formula `pendingReward - _user.rewardDebt`.

```
  function _calculatePending(UserInfo storage _user, uint256 _periodIndex) internal view
returns (uint256) {
    RewardPeriod memory period = rewardPeriods[_periodIndex];

    // If the user's staked amount is 0 or the current time is before the reward period
start time, return 0 pending reward
```

```
        if (_user.amount == 0 || block.timestamp < period.startTime) {
            return 0;
        }

        // Calculate the current accTokenPerShare for this reward period
        uint256 currentAccTokenPerShare = period.accTokenPerShare;
        if (block.timestamp <= period.endTime) {
            uint256 lrdsReward = _calculateLrdsReward(_periodIndex);
            currentAccTokenPerShare += (lrdsReward * PRECISION_FACTOR) / totalStaked;
        }

        // Calculate the pending reward based on the user's staked amount and the
period's accTokenPerShare
        uint256 pendingReward = (_user.amount * currentAccTokenPerShare) /
PRECISION_FACTOR;

        // Subtract the user's reward debt to get the actual pending reward for this period
        return pendingReward - _user.rewardDebt;
    }
```

However, since `_user.rewardDebt` holds the accumulated value from previous periods, this subtraction can result in an overflow.

Suggestion:

It is recommended to calculate the user's rewards first and then update the reward debt.

Resolution:

This issue has been fixed. The client transferred the rewards to the user before updating the reward debt.

VAL-4 The User can Repeatedly Claim Rewards from Previous Periods

Severity: Major

Status: Fixed

Code Location:

contracts/Validator.sol#586-606

Descriptions:

In the `_deposit()` function, the protocol distributes the previous earnings to the user.

```
    function _claim(uint256 _pending) internal returns (uint256 userClaimAmount, uint256
feeAmount) {
        // If there are no pending rewards, return zero values
        if (_pending == 0) return (0, 0);

        // Ensure the contract has enough reward tokens to cover the pending claim
        if (IERC20(token).balanceOf(address(this)) < _pending) revert
NotEnoughRewardToken();

        // Calculate the claim fee (claimFee is in basis points, e.g., 300 = 3%)
        feeAmount = (_pending * claimFee) / 10000;
        userClaimAmount = _pending - feeAmount;

        // Transfer the fee to the contract owner
        if (feeAmount > 0) {
            IERC20(token).safeTransfer(owner, feeAmount);
        }

        // Transfer the remaining rewards to the user
        IERC20(token).safeTransfer(msg.sender, userClaimAmount);

        emit Claim(msg.sender, userClaimAmount, feeAmount);
    }
```

However, unlike the `claim()` function, the protocol does not update `user.lastUpdatedRewardPeriod = currentPeriod` when distributing the earnings.

```
// Update the user's last updated reward period to the current period
user.lastUpdatedRewardPeriod = currentPeriod;
```

As a result, the user is able to claim rewards for the previous periods multiple times.

Suggestion:

It is recommended to update `user.lastUpdatedRewardPeriod` after the user claims the rewards.

Resolution:

This issue has been fixed. The client directly transferred the rewards to the user and then updated the user debt.

VAL-5 The User's Voting Rewards will also be Included in the Calculation of Period Rewards

Severity: Major

Status: Fixed

Code Location:

contracts/Validator.sol#744-751

Descriptions:

In the `Governance.claimAndLock()` function, the protocol transfers the reward to the validator and then calls `IValidator(masterValidator).stakeFor()` to increase the user's `user.amount`.

```
// Transfer the reward amount from the bank to the MasterValidator for staking
    IERC20(token).safeTransferFrom(bank, masterValidator, rewardAmount);

    // Stake the reward in the MasterValidator contract on behalf of the user
    IValidator(masterValidator).stakeFor(msg.sender, rewardAmount);
```

```
function stakeFor(address _user, uint256 _amount) external onlyGovernance {
    // Increase the user's staked amount
    UserInfo storage use = userInfo[_user];
    if (use.amount <= 0 ) revert NoLockCreated();
    use.amount += _amount;

    emit StakeForUser(_user, _amount);
}
```

There is an attack scenario here: if there are 10 periods during the staking lock-up, and the user deposits during period 1, then participates in governance voting until the proposal ends. The user first calls `Governance.claimAndLock()`, which triggers the `stakeFor()` function, directly updating the user's `user.amount += rewardAmount`. Then, when the `Validator.claim()` function is called, the user's reward for each period is calculated as

`pending = user.amount * accTokenPerShare`. Since the `user.amount` now includes the rewardAmount generated in governance, it will participate in the reward calculation for each period, leading the user to receive more rewards than intended.

Suggestion:

It is recommended to re-deposit the user's voting rewards back to the user.

Resolution:

This issue has been fixed. The client re-deposited the voting rewards back to the user.

VAL-6 Missing BoostReward Distribution in `withdraw` Function

Severity: Major

Status: Fixed

Code Location:

contracts/Validator.sol#293-334

Descriptions:

```
    function withdraw() external nonReentrant whenNotPaused {
        UserInfo storage user = userInfo[msg.sender];

        if (user.amount <= 0) revert ZeroAmount();
        if (block.timestamp < user.lockEndTime) revert TimeNotUp();
        if (user.autoMax == true) revert AutoMaxTime();

        // Update global reward state and user-specific rewards
        _updateValidator();
        _updateBoostReward();

        if (IERC20(token).balanceOf(address(this)) < user.amount) revert
NotEnoughRewardToken();

        // Calculate the total pending rewards
        uint256 totalPending = _calculateTotalPending(user);

        if (totalPending > 0) {
            // Call _claim to distribute the rewards
            _claim(totalPending);
        }

         // Transfer the user's staked amount back to them
        IERC20(token).safeTransfer(msg.sender, user.amount);

        // Reset votes associated with the user
        if (address(this) == masterValidator) {
            IGovernance(governance).resetVotes(msg.sender);
        }
```

```
        // Update the global staking total
        totalStaked -= user.amount;

        // Update the total staked amount and wallet count in the factory contract
        IValidatorFactory(factory).subTotalStakedAmount(user.amount);
        IValidatorFactory(factory).subTotalStakedWallet();

        delete userInfo[msg.sender];

        _updateUserDebt(user);

        emit Withdraw(msg.sender, user.amount);
    }
```

In the withdraw function of the BoostReward distribution is overlooked. After executing `delete userInfo[msg.sender]`, the user's information is reset, which prevents them from claiming the BoostReward.

Suggestion:

Modify the withdraw function to prioritize the distribution of BoostReward before reset userinfo. This ensures that users receive all pending rewards, when they exit.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VAL-7 Possible Inability to Claim Fees

Severity: Medium

Status: Fixed

Code Location:

contracts/Validator.sol#198

Descriptions:

In the `setRewardPeriod` function of the `validator` contract, the `token` in the `validatorFee` contract is set using the following code:

```
ValidatorFees(validatorFees).setToken(_stakeToken);
```

If `_stakeToken` in `setRewardPeriod` differs from the previous call, fees from the prior reward period may become unclaimable.

Suggestion:

It is recommended to change the `token` in `ValidatorFee` to an array.

Resolution:

This issue has been fixed. The client has specified that only one type of token is allowed.

VAL-8 Incorrect Condition in Quality Check

Severity: Minor

Status: Fixed

Code Location:

contracts/Validator.sol#123

Descriptions:

In the Validator contract, the following line is intended to validate that the _quality parameter is within the acceptable range:

```
if (_quality < 1 && _quality > 7) revert QualityWrong();
```

However, this condition will always evaluate to false because _quality cannot be both less than 1 and greater than 7 at the same time. As a result, this check does not effectively validate _quality, and invalid values outside the range 1–7 may pass through undetected.

Suggestion:

To properly enforce the range, the condition should use the logical OR (||) operator instead of AND (&&).

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VAL-9 Unnecessary Update of User Reward Debt

Severity: Minor

Status: Fixed

Code Location:

contracts/Validator.sol

Descriptions:

In the `_deposit()` function, the protocol calls the `_updateUserRewards()` function to update `_user.rewardDebt`.

```
function _deposit(uint256 _amount, uint256 _lockDuration, UserInfo storage _user)
internal {
    // Update global reward state and user-specific rewards
    _updateValidator();
    _updateUserRewards(_user);
    _updateBoostReward(currentBoostRewardPeriodIndex);
```

Later, within the same `_deposit()` function, the protocol updates `_user.rewardDebt` again.

```
    // If lock duration is provided but no amount is being deposited, just extend the lock
duration
    if (_lockDuration > 0 && _amount == 0) {
        _user.lockEndTime = block.timestamp < _user.lockEndTime ? _user.lockEndTime +
_lockDuration : block.timestamp + _lockDuration;
    }

    _user.rewardDebt = (_user.amount *
rewardPeriods[getCurrentPeriod()].accTokenPerShare) / PRECISION_FACTOR;
```

This second update will overwrite the first one, making the initial update performed by `_updateUserRewards()` unnecessary.

Suggestion:

It is recommended to remove the call to `_updateUserRewards()`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VFA-1 Incorrect Condition in `SubTotalStakedWallet` Function Leading to Underflow Error

Severity: Minor

Status: Fixed

Code Location:

contracts/ValidatorFactory.sol#88

Descriptions:

In the SubTotalStakedWallet function, there is an incorrect condition check:

```
if (totalStakedWallet < 0) revert NotEnoughWallet();
totalStakedWallet--;
```

Since totalStakedWallet is of type uint256, it can never be less than zero. This check is redundant and does not prevent underflow errors.

Suggestion:

We recommend replacing it with an explicit zero check.

```
if (totalStakedWallet == 0) revert NotEnoughWallet();
```

VFA-2 Missing Check for `_endTime > _startTime` in AddTotalValidators Function

Severity: Minor

Status: Fixed

Code Location:

contracts/ValidatorFactory.sol#93-114

Descriptions:

In the AddTotalValidators function, there is no check to ensure that `_endTime > _startTime`. This oversight can lead to two issues in `getTotalValidatorRewards()`:

1. Underflow

```
uint256 duration = totalValidators[i].endTime - totalValidators[i].startTime;
```

2. Division by Zero

```
uint256 duration = totalValidators[i].endTime - totalValidators[i].startTime;
totalValidatorRewards += (period * totalValidators[i].totalReward) / duration;
```

This could cause a Denial of Service (DoS) by reverting the `getTotalValidatorRewards` function whenever it is called.

Suggestion:

To prevent these issues, add a check in the AddTotalValidators function to ensure _endTime is greater than _startTime. This will help avoid potential underflow and division by zero errors.

```
if (_endTime <= _startTime) revert InvalidTimePeriod();
```

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VFA-3 Ineffective Condition in `createValidator` Function

Severity: Minor

Status: Fixed

Code Location:

contracts/ValidatorFactory.sol#158

Descriptions:

In the `createValidator` function, there is an ineffective check designed to verify if a validator already exists based on validatorId:

```
uint256 validatorId = allValidators.length;  // Use the length of allValidators array as the validatorId

// Check if validator already exists by checking if the validatorId already exists in allValidators array
if (validatorId < allValidators.length && allValidators[validatorId] != address(0)) revert PoolAlreadyExists();
```

However, this check will never execute due to the following reasons:

1. `validatorId < allValidators.length` will always be false: Since validatorId is defined as allValidators.length, this condition checks if allValidators.length < allValidators.length, which is impossible.
2. `allValidators[validatorId] != address(0)` will also always be false in this context

As a result, this check is ineffective, and PoolAlreadyExists() will never trigger.

Suggestion:

To improve code readability and maintainability, it is recommended to remove this ineffective condition, consider implementing a more reliable check.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VFA-4 Invalid Global Variables

Severity: Minor

Status: Fixed

Code Location:

contracts/ValidatorFactory.sol#23-24

Descriptions:

Several global variables in the ValidatorFactory contract are ineffective, such as feeManager and voter. Additionally, the `isPaused` variable does not successfully pause the validator.

```
mapping(address => bool) public isPaused;
```

Suggestion:

It is recommended to review and correct the implementation of these variables.

Resolution:

This issue has been fixed. The client has removed the global variables.

VFA-5 Missing Zero Address Check in setVoter Function

Severity: Informational

Status: Fixed

Code Location:

contracts/ValidatorFactory.sol#125

Descriptions:

The setVoter function currently lacks a zero address check, unlike similar functions in the contractsetPauser and setFeeManager, which include this validation.

Suggestion:

To enhance code consistency and ensure a valid voter address is always assigned, we recommend adding a zero address check in the `setVoter` function.

VAL-10 Updating the User's Debt after Deleting the User's Information is Meaningless

Severity: Minor

Status: Fixed

Code Location:

contracts/Validator.sol#331

Descriptions:

In the `withdraw()` function, the protocol first deletes the user's information, and then calls `_updateUserDebt()` to update the user's debt.

```
delete userInfo[msg.sender];

_updateUserDebt(user);
```

The issue here is that once the user's information is deleted, updating the debt with `_updateUserDebt()` becomes meaningless.

Suggestion:

It is recommended to remove the `_updateUserDebt()` function call.

Resolution:

This issue has been fixed. The client removed the `_updateUserDebt()` update.

VAL-11 Incorrect Withdraw Event Logs Due to Reset User Data

Severity: Minor

Status: Fixed

Code Location:

contracts/Validator.sol#333

Descriptions:

```
delete userInfo[msg.sender];

_updateUserDebt(user);

emit Withdraw(msg.sender, user.amount);
```

In the `withdraw` function, delete userInfo[msg.sender] resets userinfo before the Withdraw event is emitted. As a result, the event logs user.amount as 0, which is incorrect and does not reflect the actual withdrawn amount.

Suggestion:

Emit the Withdraw event before resetting `userInfo[msg.sender]` to ensure the correct user.amount is logged.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

VAL-12 Mismatch Between `MAX_LOCK` Value and Commented Description of Lock Duration

Severity: Informational

Status: Fixed

Code Location:

contracts/Validator.sol#50

Descriptions:

In the Validator contract, the constant MAX_LOCK is defined as follows:

```
uint256 public constant MAX_LOCK = 14560;    // Maximum lock duration (209 weeks - 1 second)
```

However, the value 14560 does not match the commented description "Maximum lock duration (209 weeks - 1 second)."

Suggestion:

To avoid potential confusion, please update the comment to match the intended `MAX_LOCK` value.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.