BLOCKSEC

Security Audit
Report for Story Protocol Staking

Date: February 25, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Meta Pool
Target	Story Protocol Staking

Version History

Version	Date	Description
1.0	February 25, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-02-26 02:24:50+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

This audit focuses on the code repositories of the Story Protocol Staking [1] for Meta Pool.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Story Protocol Staking	`Version 1`	`2a90f213d51f514f253feebf234af447fe75cdc1`
	`Version 2`	`5dadbb0e604ca17c879bc97bf4b34f0fd2518369`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] https://github.com/Meta-Pool/story-protocol-staking

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization

* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **five** potential security issues. Besides, we have **two** recommendations and **two** notes.

- Medium Risk: 4
- Low Risk: 1
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Potential DoS due to lack of override for function `_decimalsOffset()`	DeFi Security	Confirmed
2	Medium	Potential bypass of withdrawal limit check	DeFi Security	Confirmed
3	Medium	Bypass of slashing via early asset withdrawal	DeFi Security	Confirmed
4	Medium	Timely invoke function `sendRewardsAndFees()` before updating rewards fee	DeFi Security	Fixed
5	Medium	Lack of check for totalPendingWithdrawals in function `updateWithdrawal()`	DeFi Security	Fixed
6	-	Lack of unstake enforcement before validator removal	Recommendation	Fixed
7	-	Lack of balance check in function `coverWithdrawals()`	Recommendation	Fixed
8	-	Potential centralization risk	Note	-
9	-	Front-running opportunity in reward distribution	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential DoS due to lack of override for function `_decimalsOffset()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `StakedIP` contract, function `getStIPPrice()` relies on `convertToAssets(1 ether)`, which ultimately invokes the function `_convertToAssets()`. However, function `_convertToAssets()` includes `_decimalsOffset()`, which is not overridden in contract `StakedIP` and the default return value is 0. This leads to a division by zero error when the total supply of shares is zero, causing the function `getStIPPrice()` to revert.

```
383   function getStIPPrice() public view returns (uint256) {
384       return convertToAssets(1 ether);
385   }
```

Listing 2.1: StakedIP.sol

```
126   function convertToAssets(uint256 shares) public view virtual returns (uint256) {
127       return _convertToAssets(shares, Math.Rounding.Floor);
128   }
```

Listing 2.2: ERC4626.sol

```
232   function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view virtual
          returns (uint256) {
233       return shares.mulDiv(totalAssets() + 1, totalSupply() + 10 ** _decimalsOffset(), rounding);
234   }
```

Listing 2.3: ERC4626.sol

Impact Function `getStIPPrice()` is unavailable when the share's total supply is 0.

Suggestion Override function `_decimalsOffset()` in contract `StakedIP` to ensure the return value is not zero, preventing division by zero errors.

Feedback from the project There's no need to use `getStIPPrice()` if the supply is zero. Given that there is already a supply in the current status and we have minted `stIP`, the supply will never reach zero.

2.1.2 Potential bypass of withdrawal limit check

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The contract enforces a structured withdrawal process where users burn shares in function `_withdraw()`, which then invokes function `requestWithdraw()` on the `Withdrawal` contract. After the lock-up period, users need to invoke the function `completeWithdraw()` to finalize the withdrawal and receive their funds accordingly. To limit the number of pending withdrawals per user, the contract sets `MAX_WITHDRAWALS_PER_USER`.

However, this restriction can be bypassed. Since function `_withdraw()` only verifies the shares balance of the sender, users can transfer shares to another account, which can then initiate its own withdrawal requests, effectively increasing the limit. Additionally, users can approve another account via function `approve()`, allowing that account to withdraw on their behalf, further circumventing the restriction.

```
530   function _withdraw(
531       address _caller,
532       address _receiver,
533       address _owner,
534       uint256 _assets,
535       uint256 _shares
536   ) internal override onlyFullyOperational {
537       if (_shares == 0) revert InvalidZeroAmount();
538       if (_caller != _owner) _spendAllowance(_owner, _caller, _shares);
```

```
539
540         _burn(_owner, _shares);
541         totalUnderlying -= _assets;
542
543         IWithdrawal(withdrawal).requestWithdraw(_assets, _caller, _receiver);
544
545         emit Withdraw(_caller, _receiver, _owner, _shares, _assets);
546     }
```

Listing 2.4: StakedIP.sol

```
86  /// @notice Queue IP withdrawal
87  /// @dev Multiples withdrawals are accumulative, but will restart the unlock timestamp and
        override the receiver. Shares used for this request should be already burned in the
        calling function (StakedIP._withdraw)
88  /// @param _amountOut IP amount to withdraw
89  /// @param _user Owner of the withdrawal
90  function requestWithdraw(uint256 _amountOut, address _user, address _receiver) external
        onlyStaking {
91      if (_amountOut == 0) revert InvalidRequest();
92
93      uint256 unlockTimestamp = block.timestamp + validatorsDisassembleTime;
94      uint256 request_id = _findEmptySlot(_user);
95
96      _userPendingWithdrawals[_user][request_id] = WithdrawRequest({
97          amount: _amountOut,
98          unlockTimestamp: unlockTimestamp,
99          receiver: _receiver
100     });
101     totalPendingWithdrawals += _amountOut;
102
103     emit RequestWithdraw(_user, request_id, _amountOut, _receiver, unlockTimestamp);
104 }
```

Listing 2.5: Withdrawal.sol

```
164 function _findEmptySlot(address _user) private view returns (uint256) {
165     for (uint256 i = 0; i < MAX_WITHDRAWALS_PER_USER; i++) {
166         if (_userPendingWithdrawals[_user][i].amount == 0) return i;
167     }
168     revert UserMaxWithdrawalsReached(_user);
169 }
```

Listing 2.6: Withdrawal.sol

Impact Users can bypass the withdrawal limit by transferring or approving shares to other users.

Suggestion Revise the logic accordingly.

Feedback from the project The `MAX_WITHDRAWALS_PER_USER` exists to manage a limited amount of withdraws in the logic, and not a mapping pointing to a dynamic array producing higher gas costs. This bypass is not a problem.

2.1.3 Bypass of slashing via early asset withdrawal

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description When a `validator` is slashed, the `operator` invokes the function `reportSlash()` to synchronize the protocol's `totalUnderlying` to reflect the loss of `validator` deposit. This update decreases `totalUnderlying`, leading to a lower share price. However, before this update is recorded on-chain, users can invoke the `withdraw()` function and redeem their shares at the previous, higher share price. Since `totalUnderlying` has not yet been adjusted, the withdrawal calculation does not account for the slashing event, allowing users to receive more underlying tokens than they should.

```
459   function reportSlash(uint256 _amount, bytes calldata _validatorCmpPubkey) external
          onlyOperator checkValidSlash(_amount) checkValidatorExists(_validatorCmpPubkey) {
460       totalUnderlying -= _amount;
461       emit ReportSlash(msg.sender, _amount, _validatorCmpPubkey);
462   }
```

Listing 2.7: StakedIP.sol

```
219   function withdraw(uint256 assets, address receiver, address owner) public virtual returns (
          uint256) {
220       uint256 maxAssets = maxWithdraw(owner);
221       if (assets > maxAssets) {
222           revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
223       }
224
225       uint256 shares = previewWithdraw(assets);
226       _withdraw(_msgSender(), receiver, owner, assets, shares);
227
228       return shares;
229   }
```

Listing 2.8: ERC4626.sol

```
530   function _withdraw(
531       address _caller,
532       address _receiver,
533       address _owner,
534       uint256 _assets,
535       uint256 _shares
536   ) internal override onlyFullyOperational {
537       if (_shares == 0) revert InvalidZeroAmount();
538       if (_caller != _owner) _spendAllowance(_owner, _caller, _shares);
539
540       _burn(_owner, _shares);
541       totalUnderlying -= _assets;
542
543       IWithdrawal(withdrawal).requestWithdraw(_assets, _caller, _receiver);
544
```

```
545         emit Withdraw(_caller, _receiver, _owner, _shares, _assets);
546     }
```

Listing 2.9: StakedIP.sol

```
 90     function requestWithdraw(uint256 _amountOut, address _user, address _receiver) external
            onlyStaking {
 91         if (_amountOut == 0) revert InvalidRequest();
 92
 93         uint256 unlockTimestamp = block.timestamp + validatorsDisassembleTime;
 94         uint256 request_id = _findEmptySlot(_user);
 95
 96         _userPendingWithdrawals[_user][request_id] = WithdrawRequest({
 97             amount: _amountOut,
 98             unlockTimestamp: unlockTimestamp,
 99             receiver: _receiver
100         });
101         totalPendingWithdrawals += _amountOut;
102
103         emit RequestWithdraw(_user, request_id, _amountOut, _receiver, unlockTimestamp);
104     }
```

Listing 2.10: Withdrawal.sol

```
109     function completeWithdraw(uint256 _request_id, bool _wrap) external {
110         WithdrawRequest memory _pendingUserWithdraw = _userPendingWithdrawals[msg.sender][
                _request_id];
111
112         if (_pendingUserWithdraw.amount == 0) {
113             revert InvalidRequestId(msg.sender, _request_id);
114         }
115
116         if (block.timestamp < _pendingUserWithdraw.unlockTimestamp) {
117             revert ClaimTooSoon(_pendingUserWithdraw.unlockTimestamp);
118         }
119
120         delete _userPendingWithdrawals[msg.sender][_request_id];
121         totalPendingWithdrawals -= _pendingUserWithdraw.amount;
122
123         if (_wrap) {
124             address wIP = StakedIP(stIP).asset();
125             IWIP(wIP).deposit{ value: _pendingUserWithdraw.amount }();
126             IERC20(wIP).safeTransfer(_pendingUserWithdraw.receiver, _pendingUserWithdraw.amount);
127         } else {
128             payable(_pendingUserWithdraw.receiver).sendValue(_pendingUserWithdraw.amount);
129         }
130
131         emit CompleteWithdraw(
132             msg.sender,
133             _request_id,
134             _pendingUserWithdraw.amount,
135             _pendingUserWithdraw.receiver,
136             _pendingUserWithdraw.unlockTimestamp
```

```
137         );
138     }
```

Listing 2.11: Withdrawal.sol

Impact Users can withdraw assets at a stale share price before `totalUnderlying` is updated, effectively avoiding the impact of the slashing event. This results in unfair asset distribution and potential losses for remaining stakeholders.

Suggestion Revise the logic to recalculate the amount of underlying tokens the user should receive based on the updated share price in the function `completeWithdraw()`.

Feedback from the project We decided not to apply the suggestion. Mainly because this approach involves a mechanism whereby once a user requests a withdrawal, they no longer receive rewards/yield but instead incur penalties, which doesn't sound fair.

2.1.4 Timely invoke function `sendRewardsAndFees()` before updating rewards fee

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `RewardsManager` contract, the protocol's `owner` can update `rewardsFeeBp` through the function `updateRewardsFee()`. This variable determines the percentage of the `treasuryFee` deducted from the accrued rewards. However, in the current implementation, if `rewardsFeeBp` is updated before invoking the function `sendRewardsAndFees()`, the newly updated fee rate will be applied to previously accrued rewards, which is incorrect.

```
62     function updateRewardsFee(uint256 _rewardsFeeBp) external onlyOwner checkRewards(_rewardsFeeBp
           ) {
63         rewardsFeeBp = _rewardsFeeBp;
64
65         emit UpdateRewardsFee(msg.sender, _rewardsFeeBp);
66     }
```

Listing 2.12: RewardsManager.sol

```
78     function sendRewardsAndFees() external nonReentrant {
79         (uint256 rewards, uint256 treasuryFee) = getManagerAccrued();
80         require(rewards > 0, NoRewards());
81
82         IStakedIP(stakedIP).injectRewards{ value: rewards }();
83
84         if (treasuryFee > 0) payable(treasury).sendValue(treasuryFee);
85
86         emit SendRewardsAndFees(msg.sender, rewards, treasuryFee);
87     }
```

Listing 2.13: RewardsManager.sol

```
68     function getManagerAccrued() public view returns (uint256 rewards, uint256 treasuryFee) {
69         uint256 balance = address(this).balance;
```

```
70        if (balance > 0) {
71            treasuryFee = balance.mulDiv(rewardsFeeBp, ONE_HUNDRED, Math.Rounding.Floor);
72            rewards = balance - treasuryFee;
73        }
74    }
```

Listing 2.14: RewardsManager.sol

Impact Users may be charged a higher fee than what was originally promised under the previous fee rate.

Suggestion Revise the logic to ensure that the function `sendRewardsAndFees()` is invoked in a timely manner before updating `rewardsFeeBp`.

2.1.5 Lack of check for totalPendingWithdrawals in function `updateWithdrawal()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `updateWithdrawal()` allows the protocol's `owner` to change the designated withdrawal address. However, if this update occurs while there are pending withdrawals (i.e., `totalPendingWithdrawals` is non-zero), it can lead to a situation where user assets are misdirected.

Specifically, if a user initiates a withdrawal and the withdrawal address is changed before the operator invokes the function `unstake()`, the assets intended for the user may be sent to the new withdrawal address. Since this new address lacks the corresponding `_userPendingWithdrawals` record, the user cannot retrieve their assets.

```
219    function withdraw(uint256 assets, address receiver, address owner) public virtual returns (
           uint256) {
220        uint256 maxAssets = maxWithdraw(owner);
221        if (assets > maxAssets) {
222            revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
223        }
224
225        uint256 shares = previewWithdraw(assets);
226        _withdraw(_msgSender(), receiver, owner, assets, shares);
227
228        return shares;
229    }
```

Listing 2.15: ERC4626.sol

```
530    function _withdraw(
531        address _caller,
532        address _receiver,
533        address _owner,
534        uint256 _assets,
535        uint256 _shares
536    ) internal override onlyFullyOperational {
```

```
537         if (_shares == 0) revert InvalidZeroAmount();
538         if (_caller != _owner) _spendAllowance(_owner, _caller, _shares);
539
540         _burn(_owner, _shares);
541         totalUnderlying -= _assets;
542
543         IWithdrawal(withdrawal).requestWithdraw(_assets, _caller, _receiver);
544
545         emit Withdraw(_caller, _receiver, _owner, _shares, _assets);
546     }
```

Listing 2.16: StakedIP.sol

```
 90     function requestWithdraw(uint256 _amountOut, address _user, address _receiver) external
            onlyStaking {
 91         if (_amountOut == 0) revert InvalidRequest();
 92
 93         uint256 unlockTimestamp = block.timestamp + validatorsDisassembleTime;
 94         uint256 request_id = _findEmptySlot(_user);
 95
 96         _userPendingWithdrawals[_user][request_id] = WithdrawRequest({
 97             amount: _amountOut,
 98             unlockTimestamp: unlockTimestamp,
 99             receiver: _receiver
100         });
101         totalPendingWithdrawals += _amountOut;
102
103         emit RequestWithdraw(_user, request_id, _amountOut, _receiver, unlockTimestamp);
104     }
```

Listing 2.17: Withdrawal.sol

```
109     function completeWithdraw(uint256 _request_id, bool _wrap) external {
110         WithdrawRequest memory _pendingUserWithdraw = _userPendingWithdrawals[msg.sender][
                _request_id];
111
112         if (_pendingUserWithdraw.amount == 0) {
113             revert InvalidRequestId(msg.sender, _request_id);
114         }
115
116         if (block.timestamp < _pendingUserWithdraw.unlockTimestamp) {
117             revert ClaimTooSoon(_pendingUserWithdraw.unlockTimestamp);
118         }
119
120         delete _userPendingWithdrawals[msg.sender][_request_id];
121         totalPendingWithdrawals -= _pendingUserWithdraw.amount;
122
123         if (_wrap) {
124             address wIP = StakedIP(stIP).asset();
125             IWIP(wIP).deposit{ value: _pendingUserWithdraw.amount }();
126             IERC20(wIP).safeTransfer(_pendingUserWithdraw.receiver, _pendingUserWithdraw.amount);
127         } else {
128             payable(_pendingUserWithdraw.receiver).sendValue(_pendingUserWithdraw.amount);
```

```
129         }
130
131         emit CompleteWithdraw(
132             msg.sender,
133             _request_id,
134             _pendingUserWithdraw.amount,
135             _pendingUserWithdraw.receiver,
136             _pendingUserWithdraw.unlockTimestamp
137         );
138     }
```

Listing 2.18: Withdrawal.sol

Impact The user will lose assets that have not been withdrawn.

Suggestion Add a check to ensure that the `withdrawal` address cannot be updated if `totalPendingWithdrawals` is not zero.

2.2 Recommendations

2.2.1 Lack of unstake enforcement before validator removal

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `StakedIP` contract, the owner can remove a `validator` by setting its `targetStakePercent` to 0 and invoking the function `bulkRemoveValidators()`. However, the current implementation does not guarantee that all assets staked to the `validator` are fully withdrawn before its removal.

```
310     function stake(
311         bytes calldata _validatorCmpPubkey,
312         uint256 _amount,
313         IIPTokenStaking.StakingPeriod _period,
314         bytes calldata _extraData
315     ) external onlyFullyOperational onlyOperator checkValidatorExists(_validatorCmpPubkey) returns
            (uint256 delegation_id) {
316         delegation_id = ipTokenStaking.stake{ value: _amount }(
317             _validatorCmpPubkey,
318             _period,
319             _extraData
320         );
321
322         emit Stake(msg.sender, _validatorCmpPubkey, delegation_id, _amount, _extraData);
323     }
```

Listing 2.19: StakedIP.sol

```
139     modifier checkValidatorExists(bytes calldata _validatorCmpPubkey) {
140         require(isValidatorListed(_validatorCmpPubkey), ValidatorNotListed(_validatorCmpPubkey));
141         _;
142     }
```

Listing 2.20: StakedIP.sol

```
262   function bulkRemoveValidators(bytes[] memory _validatorsCmpPubkey) external onlyOwner {
263       for (uint256 i = 0; i < _validatorsCmpPubkey.length; ++i) {
264           _removeValidator(_validatorsCmpPubkey[i]);
265       }
266   }
```

Listing 2.21: StakedIP.sol

```
576   unction _removeValidator(bytes memory _validatorCmpPubkey) private {
577       uint256 _validatorsLength = validatorsLength;
578
579       // The final validators array cannot be empty. Once initialized, it should be
580       // theoretically impossible for it to become empty.
581       require(_validatorsLength > 1, ValidatorsEmptyList());
582
583       uint256 _index = getValidatorIndex(_validatorCmpPubkey);
584       Validator memory validator = _validators[_index];
585
586       require(validator.targetStakePercent == 0, ValidatorHasTargetPercent(validator));
587
588       if (_index != _validatorsLength - 1) {
589           // Replace the element at the index with the last element in the array
590           _validators[_index] = _validators[_validatorsLength - 1];
591           delete _validators[_validatorsLength - 1];
592       } else {
593           // If the element is the last one, just remove it
594           delete _validators[_index];
595       }
596
597       _validatorExists[keccak256(_validatorCmpPubkey)] = false;
598       validatorsLength--;
599
600       emit RemoveValidator(_validatorCmpPubkey);
601   }
```

Listing 2.22: StakedIP.sol

Suggestion Revise the logic to ensure that all assets are unstaked from the specified validator before it is removed.

2.2.2 Lack of balance check in function `coverWithdrawals()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `coverWithdrawals()` function is designed to improve capital efficiency by using newly deposited funds to directly cover pending withdrawals, avoiding the need for costly stake and unstake operations. However, the transferred amount (`_assets`) may exceed

the actual shortfall in the `Withdrawal` contract. In this case, the function could overfund the `Withdrawal` contract, reducing capital efficiency and potentially introducing unnecessary risks.

```
299    function coverWithdrawals(uint256 _assets) external onlyOperator {
300        payable(withdrawal).sendValue(_assets);
301        emit CoverWithdrawals(msg.sender, _assets);
302    }
```

Listing 2.23: StakedIP.sol

Suggestion Add a check to ensure that `_assets` is less than or equal to the shortfall of the contract `Withdrawal`.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operations (e.g., parameter setting, pause/unpause, and granting roles). Additionally, the `owner` also has the ability to upgrade the implementation. If the private keys of these privileged roles are lost or maliciously exploited, it could potentially lead to losses for users.

Feedback from the project In next stages functions ownership and upgrades ownership will be transferred to a multisig.

2.3.2 Front-running opportunity in reward distribution

Introduced by `Version 1`

Description The `RewardsManager` contract's `sendRewardsAndFees()` function distributes staking rewards to the `StakedIP` contract, increasing the share price. Users can monitor and front-run the system by depositing funds just before the reward distribution. This allows them to benefit from the higher share price even if they have staked for a shorter duration compared to other participants. The impact is limited due to the withdrawal lock time, which prevents immediate redemption and ensures a minimum staking period for all users. This behavior is an inherent part of the design and does not pose significant risks to the system.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS