Security Review Report
NM-0434-METAPOOL

NETHERMIND
SECURITY

(Feb 10, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the smart contracts of MetaPool. MetaPool protocol is a multi-chain liquidity-staking ecosystem that allows users to stake tokens and earn yield. Staking users will also get voting rights for the DAO of the protocol. This audit scope comprises smart contracts related to integration of MetaPool with Story Protocol.

Anyone can deposit either native IP tokens or wrapped IP tokens in the protocol. Only the entitled operator role will be able to stake the deposited tokens with the validator. Users can initiate withdrawal of their tokens any time with a waiting period of 14 days. The yield earned is given to the user at the time of withdrawal along with their deposited tokens.

The operator is responsible for provisioning tokens for withdrawal request either by un-staking funds from the validator or routing the funds from new deposits. Operators can choose to restake the accrued rewards for compounding.

This security review focuses exclusively on the smart contracts listed in Section 2 (*Audited Files*).

The audited code comprises of 703 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the codebase and (b) creation of test cases. **Along this document, we report** 15 points of attention, where two are classified as `Critical`, three are classified as `High`, one is classified as `Medium`, one is classified as `Low`, and 8 are classified as `Informational` and `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the overview of the system. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

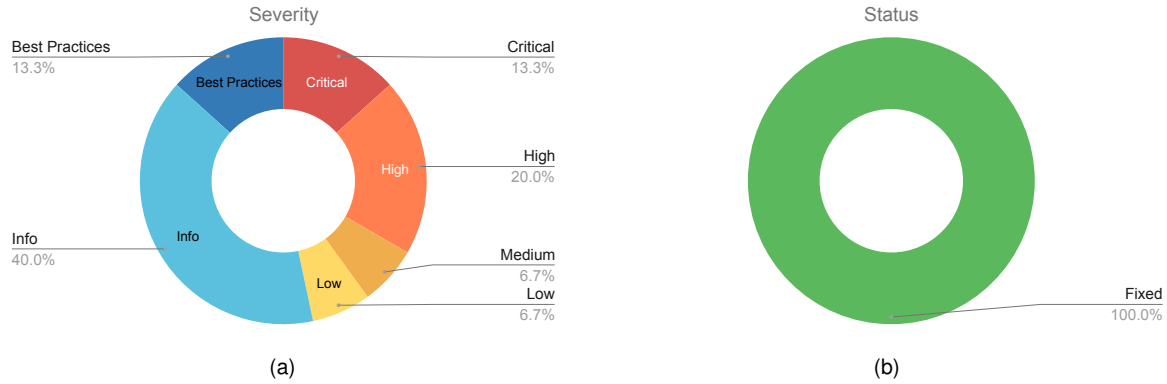

Fig. 1: Distribution of issues: Critical (2), **High** (3), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (6), **Best Practices** (2). **Distribution of status: Fixed** (15), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Feb 07, 2025
Response from Client	Regular responses during audit engagement
Final Report	Feb 10, 2025
Repository	MetaPool-Story Protocol
Commit (Audit)	24806b73c4e328cb2b65fc15b1281ecd90227f82
Commit (Final)	cdd68063ff2857093d1f1c4187c053a7d933a2af
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	StakedIP.sol	391	86	22.0%	113	590
2	RewardsManager.sol	61	6	9.8%	16	83
3	Withdrawal.sol	99	17	17.2%	32	148
4	interfaces/IStakedIP.sol	6	3	50.0%	4	13
5	interfaces/IWIP.sol	10	1	10.0%	7	18
6	interfaces/IRewardsManager.sol	5	1	20.0%	2	8
7	interfaces/IWithdrawal.sol	5	1	20.0%	2	8
8	interfaces/IStakedIPVaultOperations.sol	5	1	20.0%	2	8
9	interfaces/story/IIPTokenStakingFull.sol	6	1	16.7%	3	10
10	interfaces/story/IIPTokenStaking.sol	115	157	136.5%	30	302
	Total	**703**	**274**	**39.0%**	**211**	**1188**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect implementation of `StakedIP::mint`	Critical	Fixed
2	Users cannot deposit with WIP	Critical	Fixed
3	Excessive IP tokens in `Withdrawal` contract cannot be restaked to `StakedIP`	High	Fixed
4	Remove validator operation is buggy, creates duplicate entry in the array	High	Fixed
5	RewardsManager cannot accept Native tokens	High	Fixed
6	user Staked tokens could be locked in withdraw contract	Medium	Fixed
7	Use of `transfer` when sending native tokens	Low	Fixed
8	Emit event only when rewards are disbursed	Info	Fixed
9	IStakedIPVaultOperations interface is not used	Info	Fixed
10	Unused imports/Error in Withdrawal contract	Info	Fixed
11	`StakedIP` is not fully ERC4626-compatible	Info	Fixed
12	`getValidatorIndex` should iterate over `validatorsLength`	Info	Fixed
13	injectRewards should be callable only from RewardManager contract	Info	Fixed
14	Ownership is more secure with two-step transfers	Best Practices	Fixed
15	Unlocked and multiple solidity versions	Best Practices	Fixed

4 System Overview

MetaPool is a multi-chain liquidity stake ecosystem. The current scope of smart contracts are related to integration with the Story protocol. StakedIP is the main interfacing contract that extends the ERC4626 specification. StakedIP is upgradeable. The Story Protocol expects a RewardManager contract and a Withdrawal contract to be configured.

The tokens deposited by the users are staked and allocated to the validators. Any withdrawal request from the user will undergo a waiting period. The Story Protocol sends the yield to the RewardManager contract. Staker will get their yield at the time of withdrawal along with their deposited tokens.

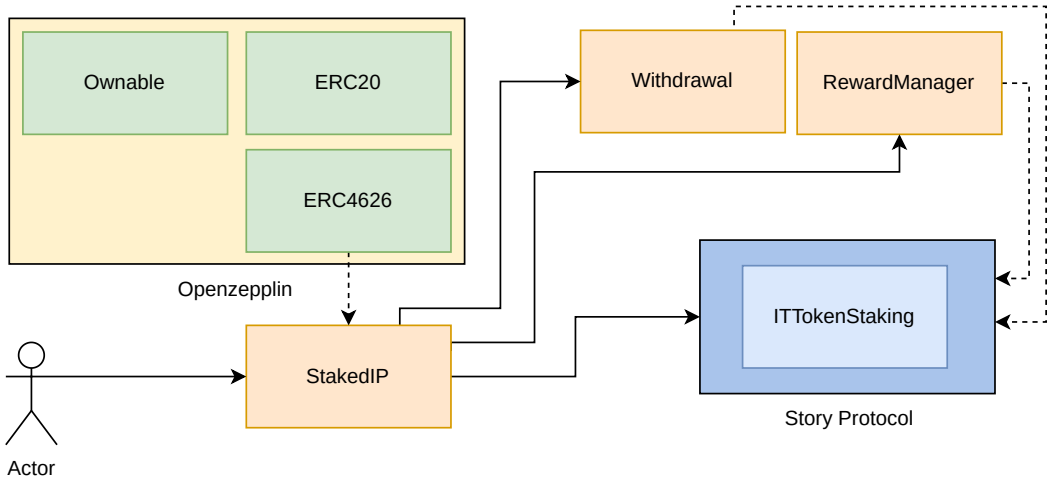

Fig. 2: MetaPool-Story Integration overview

4.1 Deposit

Anyone can deposit native or wrapped tokens to MetaPool in return for a yield. The deposited liquidity is managed by an operator role. The operator chooses the amount, the period for staking, and the validator to stake the available funds.

4.2 Withdraw

Users can initiate a request to withdraw the deposited funds anytime. As the deposited funds are staked in the Story Protocol, there is a waiting period of 14 days for the request to be processed. The operator can cover the funds required for withdrawal from new deposits or by un-staking funds from the validator. The earned yield for the user is computed at the time of withdrawal request, which is sent along with deposited funds on completion of the request.

4.3 Rewards

Story Protocol sends the rewards to the configured Reward Manager contract. The operator can choose to re-stake the earned rewards into the story protocol for compounding.

4.4 Owner functions

The owner of the protocol can perform admin functions to configure the values in the protocol.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Incorrect implementation of `StakedIP::mint`

File(s): `StakedIP.sol`

Description: The `mint` function is intended to accept `shares` as its first argument and return the required number of `assets`. Its purpose is to mint the exact number of `shares` for the receiver by depositing the corresponding amount of underlying `assets`. However, the function is incorrectly overridden:

```
function mint(uint256 _assets, address _receiver) public override returns (uint256) {
    _transferAndUnwrap(msg.sender, _assets);


    uint256 shares = previewMint(_assets);
    _deposit(msg.sender, _receiver, _assets, shares);


    return shares;
}
```

The implementation contains several critical issues:

1. The function incorrectly accepts `assets` instead of `shares` as input, contradicting its intended behavior;
2. It transfers the received `_assets`(which are the `shares`) instead of calculating and transferring the required assets for minting the specified `shares`;
3. It incorrectly computes `shares` using `previewMint`, whereas `previewMint` actually returns the amount of `assets` needed to mint the exact number of `shares`;

Recommendation(s): Update the `mint` function to correctly accept `shares` as input and ensure it transfers the required `assets` from the user:

```
function mint(uint256 shares, address receiver) public override returns (uint256) {
    uint256 assets = previewMint(shares);

    _transferAndUnwrap(msg.sender, assets);


    _deposit(msg.sender, receiver, assets, shares);


    return assets;
}
```

Status: Fixed

Update from the client: Fixed in commit d91cdae

6.2 [Critical] Users cannot deposit with WIP

File(s): `StakedIP.sol`

Description: The `StakedIP` contract allows users to deposit into the vault using WIP. The deposit function is designed to transfer WIP to the contract and subsequently unwrap it into IP:

```
function depositIP(address _receiver) public payable returns (uint256) {
    uint256 shares = previewDeposit(msg.value);


    _deposit(address(this), _receiver, msg.value, shares);

    return shares;
}

function _transferAndUnwrap(address _from, uint256 _amount) private {
    address asset = asset();
    IERC20(asset).safeTransferFrom(_from, address(this), _amount);
==>     IWIP(asset).withdraw(_amount);
}
```

However, the `StakedIP` contract incorrectly prevents the WIP contract from sending native tokens due to a restrictive condition in the `receive` fallback function:

```
receive() external payable {
==>     require(msg.sender != asset()
            && msg.sender != withdrawal, InvalidDepositSender(msg.sender));
    depositIP(msg.sender);
}
```

Since WIP must send native tokens when unwrapping, this restriction causes the withdraw function to fail, leading to a permanent failure of deposits using WIP. As a result, users cannot deposit through WIP

Recommendation(s): Revise the revert condition in the receive fallback function to allow the WIP contract to send native tokens to `StakedIP`.

Status: Fixed

Update from the client: Fixed in commit b9fc12d

6.3 [High] Excessive IP tokens in `Withdrawal` contract cannot be restaked to `StakedIP`

File(s): `StakedIP.sol` , `Withdrawal.sol`

Description: When `withdrawal` requests are made, the operator can cover withdrawals by calling `coverWithdrawals`, which transfers IP tokens to the `Withdrawal` contract. The `Withdrawal` contract may hold more IP tokens than required for withdrawals. In such cases, these excess tokens should be restaked back into the `StakedIP` vault, hence the utility of `sendIPToRestake` function. There are two key implementation issues that prevent this from happening:

1. The `sendIPToRestake` function is restricted to being called only by the `StakedIP` contract:

```
    function sendIPToRestake(uint256 _amount) external onlyStaking {
        if (address(this).balance <= totalPendingWithdrawals) revert NotEnoughIPtoStake(_amount, 0);


        uint256 ipRemaining = address(this).balance - totalPendingWithdrawals;


        if (_amount > ipRemaining) revert NotEnoughIPtoStake(_amount, ipRemaining);


        stIP.sendValue(_amount);
    }
```

   This restriction is problematic because the `StakedIP` contract cannot directly call the `Withdrawal` contract to retrieve excess tokens. Instead, the `Withdrawal` contract should be responsible for transferring these tokens back to `StakedIP`.

2. The `receive` fallback function in `StakedIP` incorrectly blocks the `Withdrawal` contract from sending IP tokens:

```
    receive() external payable {
        require(msg.sender != asset() && msg.sender != withdrawal, InvalidDepositSender(msg.sender));
        depositIP(msg.sender);
    }
```

   This restriction prevents the `Withdrawal` contract from restaking its excess IP tokens by calling `sendIPToRestake`.

Due to these implementation flaws, any excess IP tokens held by the `Withdrawal` contract are stuck and cannot be restaked.

Recommendation(s): Revisit the revert condition of `receive` fallback in `StakedIP`. Also, remove the `onlyStaking` modifier from `sendIPToRestake`.

Status: Fixed

Update from the client: Fixed in commit 13e6f70

6.4 [High] Remove validator operation is buggy, creates duplicate entry in the array

File(s): StakedIP.sol

Description: `_removeValidator` function in `StakedIPVaultOperations` does not correctly delete the item from the `validators` array. As a result, the same item could result in duplicate entries for the same validator.

```
function _removeValidator(bytes memory _validatorCmpPubkey) private {
   uint256 _validatorsLength = validatorsLength;


   require(_validatorsLength > 1, ValidatorsEmptyList());


   uint256 _index = getValidatorIndex(_validatorCmpPubkey);
   Validator memory validator = _validators[_index];


   require(validator.targetStakePercent == 0, ValidatorHasTargetPercent(validator));


   if (_index != _validatorsLength - 1) {
==>    _validators[_index] = _validators[_validatorsLength - 1];
   } else {
       delete _validators[_index];
   }
  //...
}
```

In the case where, the element being deleted is not the last element of the array, the last element of the array is copied into the index of the element being deleted. But, as the last element is not popped from the array, this will result in two instances of the same validator.

```
Example:
validators[
          {index:0,pubkey:"A"},
          {index:1,pubkey:"B"},
          {index:2,pubkey:"C"}
        ];


// Remove pubkey=="B" would result as below

validators[
          {index:0,pubkey:"A"},
          {index:2,pubkey:"C"},
          {index:2,pubkey:"C"}
        ];
```

This will impact how the `validators` array is used in the contract. For example, `redistribute` function will return incorrect results.

Recommendation(s): Pop the last element of the validators array.

Status: Fixed

Update from the client: Fixed in commit 16ab74e

6.5 [High] RewardsManager cannot accept Native tokens

File(s): RewardsManager.sol

Description: RewardsManager contract tracks the accrued rewards which are then restaked as part of compounding. A treasury fee is deducted from the rewards which is sent to the treasury address.

```
function sendRewardsAndFees() external {
   (uint rewards, uint treasuryFee) = getManagerAccrued();


   IStakedIP(stakedIP).injectRewards{ value: rewards }();
   payable(treasury).sendValue(treasuryFee);


   emit SendRewards(msg.sender, rewards);
   emit SendFees(msg.sender, treasuryFee);
}

function getManagerAccrued() public view returns (uint rewards, uint treasuryFee) {
   uint balance = address(this).balance;
   treasuryFee = (balance * rewardsFeeBp) / ONE_HUNDRED;
   rewards = balance - treasuryFee;
}
```

The issue arises from how the accrued amount is computed in the contract. As the RewardsManager contract does not have a receive or payable functions, there seems no way in how the balance of native tokens in the contract will accrue for distribution.

This issue impacts the working of reward accruals for the protocol.

Recommendation(s): Add receive function so that rewards can be received into the contract from story protocol.

Status: Fixed

Update from the client: Fixed in commit 0be2869

6.6 [Medium] User staked tokens could be locked in Withdraw contract

File(s): StakedIP.sol

Description: StakedIP contract overrides the _withdraw function of ERC4626 library contract and implements a withdrawal queue that has a waiting period before the request can be completed. On _withdraw call, the accounting for shares and assets are updated and a request is submitted to the Withdrawal contract. As a result, the withdrawal will not be completed in a single transaction.

```
function _withdraw(
   address _caller,
   address _receiver,
   address _owner,
   uint256 _assets,
   uint256 _shares
) internal override onlyFullyOperational {
   if (_shares == 0) revert InvalidZeroAmount();
   if (_caller != _owner) _spendAllowance(_owner, _caller, _shares);


   _burn(_owner, _shares);
   totalUnderlying -= _assets;

   IWithdrawal(withdrawal).requestWithdraw(_assets, _caller, _receiver);

   emit Withdraw(msg.sender, _receiver, _owner, _shares, _assets);
}
```

When the user tries to process the pending withdrawal request, the issues arises if the _receiver setup in the request is a smart contract that does not accept native tokens. In that case, the completeWithdraw will revert and the funds will remain locked in the Withdrawal contract.

```
function completeWithdraw(uint256 _request_id) external {
   WithdrawRequest memory _pendingUserWithdraw = _userPendingWithdrawals[msg.sender][_request_id];
   //...
   payable(_pendingUserWithdraw.receiver).sendValue(_pendingUserWithdraw.amount);
   //...
}
```

Recommendation(s): Add an option for the request owner to send WIP instead of IP tokens.

Status: Fixed

Update from the client: Fixed in commit d700889

6.7 [Low] Use of `transfer` when sending native tokens

File(s): `StakedIP.sol`

Description: In order to transfer native tokens, the `StakedIP` contract is currently using the `transfer` function. This is not a recommended function to move native tokens due to the fixed gas limit of 2300. As the gas cost is subject to change as part of infra upgrades.

Also, if the recipient is a smart contract like a multi-sign wallet, 2300 gas will not cover the gas required to complete the transfer leading to revert.

```
function mintIP(uint256 _shares, address _receiver) external payable returns (uint256) {
   //...
   uint256 change = msg.value - assets;
   if (change > 0) {
       payable(msg.sender).transfer(change);
   }
   //...
}
```

Recommendation(s): Use `.sendValue` function to move native tokens like in other places in the contract.

Status: Fixed

Update from the client: Fixed in commit 735a14f

6.8 [Info] Emit event only when rewards are disbursed

File(s): `RewardsManager.sol`

Description: In `RewardsManager` contact, `sendRewardsAndFees` function can be called by anyone to process the accrued rewards. Rewards are injected to the `StakedIP` contract and the applicable fee is transferred to the treasury address.

```
function sendRewardsAndFees() external nonReentrant {
   (uint256 rewards, uint256 treasuryFee) = getManagerAccrued();

   if (rewards > 0) IStakedIP(stakedIP).injectRewards{ value: rewards }();
   if (treasuryFee > 0) payable(treasury).sendValue(treasuryFee);

==> emit SendRewardsAndFees(msg.sender, rewards, treasuryFee);
}
```

The event `SendRewardsAndFees` should be emitted only when accrued `rewards` are processed. In the current implementation, the event is emitted even when there are no accrued rewards.

Recommendation(s): emit `SendRewardsAndFees` event only when `rewards` > 0.

Status: Fixed

Update from the client: Fixed in commit 0eed09c

6.9 [Info] IStakedIPVaultOperations interface is not used

File(s): The `IStakedIPVaultOperations.sol`

Description: Functions of `IStakedIPVaultOperations` interface are migrated from dedicated operations contract to `StakedIP` contract, but the contract does not derive from `IStakedIPVaultOperations`.

```
interface IStakedIPVaultOperations {
 function getValidatorIndex(bytes memory _validatorUncmpPubkey) external view returns (uint256);

 function isValidatorListed(bytes memory _validatorUncmpPubkey) external view returns (bool);
}

```

Recommendation(s): `StakedIP` contract should inherit from `IStakedIPVaultOperations` interface.

Status: Fixed

Update from the client: Removed the interface

6.10 [Info] Unused imports/Error in Withdrawal contract

File(s): `Withdrawal.sol`

Description: The `Withdrawal` contract has imports from other files, but they were never used in the contract.

- StakedIP;
- ERC4626Upgradeable;

Like wise, the below error was also not used in the contract.

- WithdrawAlreadyCompleted;

Recommendation(s): Remove redundant imports or objects from the contract.

Status: Fixed

Update from the client: Fixed in commit edf0024

6.11 [Info] `StakedIP` is not fully ERC4626-compatible

File(s): `StakedIP.sol`

Description: In the `_deposit` function, the first parameter is the caller of the function. But, in the case of `depositIP` and `mintIP`, the address of the contract is being passed as a caller instead of `msg.sender`.

```
function depositIP(address _receiver) public payable returns (uint256) {
   uint256 shares = previewDeposit(msg.value);

   _deposit(address(this), _receiver, msg.value, shares);

   return shares;
}
```

While the contract can indeed deposit IP in the staking setup via `setupStaking`, users can still call the function to deposit IP and in such cases, the depositor will be logged as the contract's address instead of the caller.

In addition, the first parameter of the `Withdraw` event in `_withdraw` is emitted with the `msg.sender` directly, while the function already accepts the caller in its first argument and that argument should be used instead.

Recommendation(s): Consider:

1. Passing `msg.sender` as caller for `_deposit` in the `depositIP` function.
2. Updating the first argument of the `Withdraw` event emitted in `_withdraw` from `msg.sender` to `_caller`.

Status: Fixed

Update from the client: Fixed in commit cdd6806

6.12 [Info] `getValidatorIndex` should iterate over `validatorsLength`

File(s): `StakedIP.sol`

Description: The `StakedIP` contract tracks the current number of validators using `validatorsLength`. While the max number of validators is 10, In the case where the current number is smaller, the `getValidatorIndex` function is still iterating over the full length of the array. Instead, while searching for validator index, the logic can iterate up to `validatorsLength` instead of `validators.length`.

```
function getValidatorIndex(bytes memory _validatorCmpPubkey) public view returns (uint256) {
   Validator[MAX_VALIDATORS] memory validators = _validators;

   bytes32 _validatorPubkeyHash = keccak256(_validatorCmpPubkey);
   for (uint256 i = 0; i < validators.length; ++i) {
       if (keccak256(validators[i].cmpPubkey) == _validatorPubkeyHash) {
           return i;
       }
   }

   revert ValidatorNotFount(_validatorCmpPubkey);
}
```

Recommendation(s): Update for loop to iterate on `validatorsLength` value instead of `validators.length`.

Status: Fixed

Update from the client: Fixed in commit 55916ab

6.13 [Info] injectRewards should be callable only from RewardManager contract

File(s): StakedIP.sol, RewardsManager.sol

Description: When sendRewardsAndFees function is called on the RewardManager contract, the reward funds are sent from the RewardManager contract to the StakedIP contract.

```
function sendRewardsAndFees() external nonReentrant {
   (uint256 rewards, uint256 treasuryFee) = getManagerAccrued();

   if (rewards > 0) IStakedIP(stakedIP).injectRewards{ value: rewards }();
   //...
}
```

But, in the StakedIP contract, there is no access control assigned to the injectRewards function. As a result any one can call the function.

```
function injectRewards() external payable {
   require(msg.value >= minDepositAmount, LessThanMinDeposit());
   totalUnderlying += msg.value;
}
```

Recommendation(s): Assign a modifier that will restrict the access for injectRewards to RewardManager contract only.

Status: Fixed

Update from the client: Fixed in commit 090f129

6.14 [Best Practices] Ownership is more secure with two-step transfers

File(s): RewardsManager.sol

Description: As owner is a critical role in managing admin functions of the protocol, it is recommended to implement a two step transfer process. In the first step, the current owner initiates the process for the new account to take over the ownership. As a second step, the new account will claim the ownership. This ensures that the caller of the claiming function has control on the new account.

```
contract RewardsManager is IRewardsManager, Ownable, ReentrancyGuard {
```

Recommendation(s): Consider using the Ownable2Step library contract of openzeppelin instead of Ownable. Likewise for StakedIP and Withdrawal contracts, consider using the Ownable2StepUpgradeable library contract.

Status: Fixed

Update from the client: Fixed in commit cfb9d4d

6.15 [Best Practices] Unlocked and multiple solidity versions

File(s): IWIP.sol, IIPTokenStaking.sol

Description: It is recommended to use locked and single solidity versions in the project. The below two interface files are using an unlocked and older version of Solidity.

- IWIP.sol;
- IIPTokenStaking.sol;

```
pragma solidity ^0.8.23;
```

Recommendation(s): It is recommended to use a single and locked version of Solidity.

Status: Fixed

Update from the client: Fixed in commit 41c0b29

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about MetaPool documentation

The `MetaPool` team was actively present in regular calls, effectively addressing concerns and questions raised by the Nethermind Security team. However, the code contained only minimal NatSpec documentation, and there was no comprehensive project documentation. The team could improve this by providing a well-structured written overview of the system and detailing the different design choices to enhance clarity and accessibility.

8 Test Suite Evaluation

8.1 Compilation Output

```
> npx hardhat compile
Compiled 41 Solidity files successfully (evm target: paris).
·-------------------------|--------------------------------|--------------------------------·
|  Solc version: 0.8.28   ·  Optimizer enabled: true       ·  Runs: 200                     |
··························|································|·································
|  Contract Name          ·  Deployed size (KiB) (change)  ·  Initcode size (KiB) (change)  |
··························|································|·································
|  Address                ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  EllipticCurve          ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  EnumerableSet          ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  Errors                 ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  IPTokenStaking         ·                 10.276 ()      ·                 10.731 ()      |
··························|································|·································
|  Math                   ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  Panic                  ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  RewardsManager         ·                  1.570 ()      ·                  1.964 ()      |
··························|································|·································
|  SafeCast               ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  SafeERC20              ·                  0.057 ()      ·                  0.085 ()      |
··························|································|·································
|  StakedIP               ·                 16.704 ()      ·                 16.915 ()      |
··························|································|·································
|  Token                  ·                  1.772 ()      ·                  2.735 ()      |
··························|································|·································
|  WIP                    ·                  1.772 ()      ·                  2.585 ()      |
··························|································|·································
|  Withdrawal             ·                  2.904 ()      ·                  3.115 ()      |
·-------------------------|--------------------------------|--------------------------------·
```

8.2 Tests Output

```
> npx hardhat test
·------------------------|--------------------------------|--------------------------------·
|  Solc version: 0.8.28  ·  Optimizer enabled: true       ·  Runs: 200                     |
·------------------------|--------------------------------|--------------------------------·
|  Contract Name         ·  Deployed size (KiB) (change)  ·  Initcode size (KiB) (change)  |
·------------------------|--------------------------------|--------------------------------·
|  Address               ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  EllipticCurve         ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  EnumerableSet         ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  Errors                ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  IPTokenStaking        ·                10.276 (0.000)  ·                10.731 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  Math                  ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  Panic                 ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  RewardsManager        ·                 1.570 (0.000)  ·                 1.964 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  SafeCast              ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  SafeERC20             ·                 0.057 (0.000)  ·                 0.085 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  StakedIP              ·                16.704 (0.000)  ·                16.915 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  Token                 ·                 1.772 (0.000)  ·                 2.735 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  WIP                   ·                 1.772 (0.000)  ·                 2.585 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
|  Withdrawal            ·                 2.904 (0.000)  ·                 3.115 (0.000)  |
·------------------------|--------------------------------|--------------------------------·
```

```
  Staked IP  - Stake IP tokens in Meta Pool ----
    Deploying Staked IP protocol
Warning: Potentially unsafe deployment of contracts/mocks/IPTokenStaking.sol:IPTokenStaking

    You are using the `unsafeAllow.state-variable-immutable` flag.

Warning: Potentially unsafe deployment of contracts/mocks/IPTokenStaking.sol:IPTokenStaking

    You are using the `unsafeAllow.constructor` flag.

       [T100]-1 StakedIPContract initial parameters are correct. (549ms)
       [T101]-1 RewardsManagerContract initial parameters are correct.
       [T102]-1 WithdrawalContract initial parameters are correct.
    Trigger all unit errors
       [T103]-1 RewardsManagerContract - InvalidAddressZero().
       [T104]-1 RewardsManagerContract - InvalidRewardsFee().
       [T105]-1 RewardsManagerContract - OwnableUnauthorizedAccount().
       [T106]-1 StakedIPContract - InvalidZeroAmount().
       [T107]-1 StakedIPContract - InvalidZeroAddress().
       [T108]-1 StakedIPContract - LessThanMinDeposit().
       [T109]-1 StakedIPContract - OperatorUnauthorized().
       [T110]-1 StakedIPContract - InvalidIPFee().
       [T111]-1 StakedIPContract - NotFullyOperational().
       [T112]-1 StakedIPContract - ValidatorNotListed().
       [T113]-1 StakedIPContract - OwnableUnauthorizedAccount().
       [T114]-1 StakedIPContract - ValidatorAlreadyListed().
       [T115]-1 StakedIPContract - ValidatorHasTargetPercent().
       [T116]-1 StakedIPContract - MaxValidatorsExceeded().
       [T117]-1 StakedIPContract - ShouldBeOneHundred().
       [T118]-1 StakedIPContract - ArraySizeMismatch().
       [T119]-1 StakedIPContract - ValidatorNotFount().
       [T120]-1 StakedIPContract - ValidatorsEmptyList().


  21 passing (646ms)
```

8.2.1 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8.2.2 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report 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 sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.