Security Audit Report for Metapool-ethereum

Date: Jul 10, 2023

Version: 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Metapool
Target	Metapool-ethereum

Version History

Version	Date	Description
1.0	July 10, 2023	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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

The repository that has been audited includes Metapool-ethereum [1].

The auditing process is iterative. Specifically, we will 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. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
Metapool	`Version 1`	`c448ad22a85596e72ecea75f25cc8fa1797e077a`
	`Version 2`	`8f4f9b179e2abe511ddffd9ab181744bff9addba`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **Metapool-ethereum/contracts** folder contract only. Specifically, the files covered in this audit include:

- contracts/interfaces/IDeposit.sol
- contracts/interfaces/IWeth.sol
- contracts/interfaces/IERC4626Router.sol
- contracts/ERC4626Router.sol
- contracts/LiquidUnstakePool.sol
- contracts/Staking.sol
- contracts/Withdrawal.sol

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

[1] https://github.com/Meta-Pool/metapool-ethereum/

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.

- **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
* Access control
* 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.

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.

Table 1.1: Vulnerability Severity Classification

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

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 find **six** potential issues. Besides, we have **three** recommendations and **four** notes as follows:

- High Risk: 0
- Medium Risk: 2
- Low Risk: 4
- Recommendations: 3
- Notes: 4

ID	Severity	Description	Category	Status
1	Low	Denial of Service by Uninitialized System Parameters	Software Security	Fixed
2	Medium	Lack of Check for rewardsFee	DeFi Security	Fixed
3	Medium	Denial of Service by Redundant Check	DeFi Security	Fixed
4	Low	Stolen User's Assets by Loss of Precision	DeFi Security	Fixed
5	Low	Lack of Check on Duplicate Nodes	DeFi Security	Fixed
6	Low	Incorrect Event Parameter	Software Security	Fixed
7	-	Incorrect Annotation in updateNodesBalance()	Recommendation	Fixed
8	-	Lack of Check on Address	Recommendation	Fixed
9	-	Failure to Adhere to Checks-Effects-Interactions Pattern	Recommendation	Fixed
10	-	Potential Centralization Problem	Note	Confirmed
11	-	Timely Triggering of Privileged Function pushToBeacon()	Note	Confirmed
12	-	Challenges in Achieving Real-time and Accurate Updates of Staking Rewards on Beacon Chain	Note	Confirmed
13	-	Withdrawals might be Delayed if the Ethereum Network is Congested	Note	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Denial of Service by Uninitialized System Parameters

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Staking` contract, state variables `withdrawal` and `liquidUnstakePool` are accessed in privileged functions such as `pushToBeacon()` and `updateNodeBalance()`. However, they are not initialized in the function `initialize()`. The values will be 0 by default if they are not configured in the functions `updateWithdrawal()` and `updateLiquidPool()`.

```
162    function updateWithdrawal(address payable _withdrawal)
```

```
163         external
164         onlyRole(DEFAULT_ADMIN_ROLE)
165     {
166         withdrawal = _withdrawal;
167     }
```

Listing 2.1: Staking.sol

```
171     function updateLiquidPool(address payable _liquidPool)
172         external
173         onlyRole(DEFAULT_ADMIN_ROLE)
174     {
175         if (_liquidPool == address(0)) revert ZeroAddress("liquidPool");
176         liquidUnstakePool = _liquidPool;
177     }
```

Listing 2.2: Staking.sol

Impact The user can still stake their funds, but withdrawals are not allowed. Besides, the whole protocol will not work properly.

Suggestion Configure the `withdrawal` and `liquidUnstakePool` properly in the function `initialize()`.

2.2 DeFi Security

2.2.1 Lack of Check for rewardsFee

Severity Medium

Status Fixed in in `Version 2`

Introduced by `Version 1`

Description In the function `updateNodesBalance()` of the contract `Staking` , if `nodesTotalBalance` increases, a certain proportion of `mpETH`, calculated based on `rewardsFee`, will be charged and sent to the `treasury`. This rewardsFee can be changed by the privileged role `DEFAULT_ADMIN_ROLE` via the function `updateRewardsFee()`. However, there is no check to limit the maximum value of this system parameter.

```
181  function updateRewardsFee(uint16 _rewardsFee)
182      public
183      onlyRole(DEFAULT_ADMIN_ROLE)
184  {
185      rewardsFee = _rewardsFee;
186  }
```

Listing 2.3: Staking.sol

Impact The user may get no farming rewards earned from the Beacon Chain if the `rewardsFee` is 100% .

Suggestion Add a check to ensure the `rewardsFee` can never exceed a reasonable maximum value in the function `updateRewardsFee()`.

2.2.2 Denial of Service by Redundant Check

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `getEthForValidator()` of the contract `LiquidUnstakePool`, there is a check `assert(previousTotalAssets == totalAssets())`.

If there is a precision loss during the `Staking(STAKING).depositETH` process, the checks in the assert statement will not pass, resulting in a revert.

```
237  function getEthForValidator(uint _amount) external nonReentrant onlyStaking {
238        uint currentETHPercentage = (ethBalance * 10000) / totalAssets();
239        uint newEthPercentage = ((ethBalance - _amount) * 10000) / totalAssets();
240        if (newEthPercentage < minETHPercentage) {
241            uint availableETH = ((currentETHPercentage - minETHPercentage) *
242                totalAssets()) / 10000;
243            revert RequestedETHReachMinProportion(_amount, availableETH);
244        }
245        uint previousTotalAssets = totalAssets();
246        ethBalance -= _amount;
247        Staking(STAKING).depositETH{value: _amount}(address(this));
248        assert(previousTotalAssets == totalAssets());
249        emit SendETHForValidator(block.timestamp, _amount);
250    }
```

Listing 2.4: LiquidUnstakePool.sol

Impact Once loss of precision occurred during `Staking(STAKING).depositETH`, the function `getEthForValidator()` will revert.

Suggestion Remove redundant checks.

2.2.3 Stolen User's Assets by Loss of Precision

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `redeem()` allows the liquidity provider of the `LiquidUnstakePool` to redeem their `Ethers` with rewards. However, if the variable `_shares` is too small, the `poolPercentage` may be rounded down to 0 due to the precision loss.

```
173  function redeem(
174        uint _shares,
175        address _receiver,
176        address _owner
177    ) public virtual override nonReentrant returns (uint ETHToSend) {
178        if (msg.sender != _owner) {
179            _spendAllowance(_owner, msg.sender, _shares);
180        }
181        uint poolPercentage = (_shares * 1 ether) / totalSupply();
```

```
182         ETHToSend = (poolPercentage * ethBalance) / 1 ether;
183         uint mpETHToSend = (poolPercentage *
184             Staking(STAKING).balanceOf(address(this))) / 1 ether;
185         _burn(msg.sender, _shares);
186         payable(_receiver).sendValue(ETHToSend);
187         IERC20Upgradeable(STAKING).safeTransfer(_receiver, mpETHToSend);
188         ethBalance -= ETHToSend;
189         emit RemoveLiquidity(msg.sender, _shares, ETHToSend, mpETHToSend);
190     }
```

Listing 2.5: LiquidUnstakePool.sol

Impact When the variable `_shares` is too small, the user's shares are burnt without receiving anything.

Suggestion Add a check to ensure that `poolPercentage` is greater than 0.

2.2.4 Lack of Check on Duplicate Nodes

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In Ethereum staking, depositing more than 32 `Ethers` to a single set of keys does not increase rewards potential, nor does accumulating rewards above 32 `Ethers`. However, the function `pushBeacon()` lacks a check to ensure if the node hasn't been deposited before.

```
250     function pushToBeacon(Node[] memory _nodes, uint256 _requestPoolAmount, uint256
            _requestWithdrawalAmount)
251     external
252     onlyOperational onlyRole(ACTIVATOR_ROLE)
253 {
254     uint32 nodesLength = uint32(_nodes.length);
255     uint256 requiredBalance = nodesLength * 32 ether;
256     // TODO: Check exact amount of ETH needed to stake
257     if (
258         stakingBalance + _requestPoolAmount + _requestWithdrawalAmount <
259         requiredBalance
260     )
261         revert NotEnoughETHtoStake(
262             stakingBalance,
263             _requestPoolAmount,
264             _requestWithdrawalAmount,
265             requiredBalance
266         );
267
268     if (_requestPoolAmount > 0)
269         LiquidUnstakePool(liquidUnstakePool).getEthForValidator(_requestPoolAmount);
270     if (_requestWithdrawalAmount > 0)
271         Withdrawal(withdrawal).getEthForValidator(_requestWithdrawalAmount);
272
273     uint32 _totalNodesActivated = totalNodesActivated;
274
275     for (uint256 i = 0; i != nodesLength; ++i) {
```

```
276         depositContract.deposit{value: 32 ether}(
277             _nodes[i].pubkey,
278             _nodes[i].withdrawCredentials,
279             _nodes[i].signature,
280             _nodes[i].depositDataRoot
281         );
282         _totalNodesActivated++;
283         emit Stake(_totalNodesActivated, _nodes[i].pubkey);
284     }
285
286     uint256 requiredBalanceFromStaking = requiredBalance - _requestWithdrawalAmount;
287     // Amount from Withdrawal isn't included as this amount was never substracted from
            nodesAndWithdrawalTotalBalance and never added to stakingBalance
288     stakingBalance -= requiredBalanceFromStaking;
289     nodesAndWithdrawalTotalBalance += requiredBalanceFromStaking;
290     totalNodesActivated = _totalNodesActivated;
291 }
```

Listing 2.6: Staking.sol

Impact If deposit `Ethers` are deposited to duplicate nodes, the staking rewards will be less as expected.

Suggestion Add a checks to prevent depositing `Ethers` to duplicate nodes.

2.2.5 Incorrect Event Parameter

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 2`

Description In the function `withdraw()` of the contract `LiquidUnstakePool`, The parameter within the `RemoveLiquidity` event is incorrect. Based on the given event definition, the first event parameter should be the `_owner` instead of the `msg.sender`.

```
150     function withdraw(
151         uint256 _assets,
152         address _receiver,
153         address _owner
154     ) public override returns (uint256 shares) {
155         shares = previewWithdraw(_assets);
156         if (msg.sender != _owner) _spendAllowance(_owner, msg.sender, shares);
157         uint256 poolPercentage = (_assets * 1 ether) / totalAssets();
158         if (poolPercentage == 0) revert AssetsTooLow();
159         uint256 ETHToSend = (poolPercentage * ethBalance) / 1 ether;
160         uint256 mpETHToSend = (poolPercentage * Staking(STAKING).balanceOf(address(this))) /
161             1 ether;
162         _burn(_owner, shares);
163         ethBalance -= ETHToSend;
164         IERC20Upgradeable(STAKING).safeTransfer(_receiver, mpETHToSend);
165         payable(_receiver).sendValue(ETHToSend);
166         emit RemoveLiquidity(msg.sender, shares, ETHToSend, mpETHToSend);
167         emit Withdraw(msg.sender, _receiver, _owner, ETHToSend, shares);
168     }
```

Listing 2.7: LiquidUnstakePool.sol

Impact Misconfigured event parameters can potentially lead to confusion and misinformation.

Suggestion Change `emit RemoveLiquidity( msg.sender, ..)` to `emit RemoveLiquidity( _owner, ..)`.

2.3 Additional Recommendation

2.3.1 Incorrect Annotation in updateNodesBalance()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The annotation in the function `updateNodesBalance()` of the contract `Staking` is incorrect (line 188).

```
188    /// @notice Update Withdrawal contract address
189    /// @dev Updater function
190    /// @notice Updates nodes total balance
191    /// @param _newNodesBalance Total current ETH balance from validators
192    function updateNodesBalance(uint _newNodesBalance) external onlyRole(UPDATER_ROLE)
```

Listing 2.8: Staking.sol

Suggestion I Revise the incorrect annotation.

2.3.2 Lack of Check on Address

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Lack of zero address check before updating address variables in multiple places, such as function `initialize()` and `updateWithdrawal()`.

```
 91    function initialize(
 92        IDeposit _depositContract,
 93        IERC20MetadataUpgradeable _weth,
 94        address _treasury,
 95        address _updater,
 96        address _activator
 97    ) external initializer {
 98        __ERC4626_init(IERC20Upgradeable(_weth));
 99        __ERC20_init("MetaPoolETH", "mpETH");
100        __AccessControl_init();
101        require(
102            _weth.decimals() == 18,
103            "wNative token error, implementation for 18 decimals"
104        );
105        require(
106            address(this).balance == 0,
```

```
107             "Error initialize with no zero balance"
108         );
109         _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
110         _grantRole(UPDATER_ROLE, _updater);
111         _grantRole(ACTIVATOR_ROLE, _activator);
112         updateRewardsFee(500);
113         treasury = _treasury;
114         depositContract = _depositContract;
115         nodesBalanceUnlockTime = uint64(block.timestamp);
116     }
```

Listing 2.9: Staking.sol

```
162     function updateWithdrawal(address payable _withdrawal)
163         external
164         onlyRole(DEFAULT_ADMIN_ROLE)
165     {
166         withdrawal = _withdrawal;
167     }
```

Listing 2.10: Staking.sol

Suggestion I Add zero address check before updating address variable.

2.3.3 Failure to Adhere to Checks-Effects-Interactions Pattern

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `redeem()`, the operation of sending Ether to the `_receiver` occurs before the update of the `ethBalance`, which breaks the Checks-Effects-Interactions pattern.

```
173   function redeem(
174         uint _shares,
175         address _receiver,
176         address _owner
177     ) public virtual override nonReentrant returns (uint ETHToSend) {
178         if (msg.sender != _owner) {
179             _spendAllowance(_owner, msg.sender, _shares);
180         }
181         uint poolPercentage = (_shares * 1 ether) / totalSupply();
182         ETHToSend = (poolPercentage * ethBalance) / 1 ether;
183         uint mpETHToSend = (poolPercentage *
184             Staking(STAKING).balanceOf(address(this))) / 1 ether;
185         _burn(msg.sender, _shares);
186         payable(_receiver).sendValue(ETHToSend);
187         IERC20Upgradeable(STAKING).safeTransfer(_receiver, mpETHToSend);
188         ethBalance -= ETHToSend;
189         emit RemoveLiquidity(msg.sender, _shares, ETHToSend, mpETHToSend);
190     }
```

Listing 2.11: LiquidUnstakePool.sol

Suggestion I Send Ether to the `_receiver` after the `ethBalance` has been updated.

2.4 Notes

2.4.1 Potential Centralization Problem

Status Confirmed
Introduced by version 1

Description This project has potential centralization problems. The privileged role DEFAULT_ADMIN_ROLE can change the contract address of liquidUnstakePool and withdrawal at any time. Meanwhile, the privileged role ACTIVATOR_ROLE is able to call depositContract.deposit() to activate specified validators, by calling the pushToBeacon() function while the privileged role UPDATER_ROLE can upload a new nodesTotalBalance to update the farming rewards received from Beacon Chain. We suggest these roles should be in multi-signature.

Feedback from the Project The admin (i.e., DEFAULT_ADMIN_ROLE) will be managed by a multisig. Given the architecture of Ethereum Beacon and consensus chain, there's no way to do the calculation on-chain, so UPDATER_ROLE needs to be assigned to an automated monitor bot collecting rewards and penalties in the beacon chain.

2.4.2 Timely Triggering of Privileged Function pushToBeacon()

Status Confirmed
Introduced by version 1

Description The function pushToBeacon() allows the privileged role ACTIVATOR_ROLE to timely deposit the staked funds into the Beacon Chain for earning. This function should be triggered timely. Otherwise, the rewards will be less as expected and the withdrawals from users may also get stuck.

Feedback from the Project The call is automated and monitored. Timely execution is of utmost interest to the protocol, so it is expected to be executed timely out of self interest.

2.4.3 Challenges in Achieving Real-time and Accurate Updates of Staking Rewards on Beacon Chain

Status Confirmed
Introduced by version 1

Description The update of staking rewards earned from the Beacon Chain is not real-time, and has a 4-hour delay, which requires the UPDATER_ROLE to actively invoke the function updateNodesBalance() for the update. In this case, the user's rewards may differ from their expectations. If the UPDATER_ROLE fails to update on time (i.e., every four hours), the rewards will be further reduced. The synchronization of the amount rewards on the Beacon Chain is exclusively performed off-chain, as there are no alternatives available within the current architecture of Ethereum to facilitate on-chain calculations. All these require users to have complete trust on the UPDATER_ROLE. However, a validation process is also implemented to ensure that the updated balance does not deviate by more than +/-0.1%, which minimizing the aforementioned error to a negligible extent.

Feedback from the Project Given the tx cost of Ethereum, **It is not viable to update staking rewards earned from the Beacon Chain "real-time"**, nevertheless rewards for users are estimated by the second,

the contract has a “rewardsPerSecond” variable that takes care of this. Every 4 hours the rewards are **confirmed or adjusted**. The report provided by the bot **can be verified by everyone**, meaning the rewards are publicly informed by the **Beacon Chain**. The protocol provides the addresses required for public verification.

2.4.4 Withdrawals might be Delayed if the Ethereum Network is Congested

Status Confirmed

Introduced by `version 1`

Description Users are allowed to withdraw their staked `Ethers` in the contract `Withdrawal`. In order to successfully execute the withdrawal, two conditions must be met: 1) Sufficient time has elapsed, and 2) There exists an adequate amount of `Ethers` within the contract.

`Ethers` are acquired through the disassemble of validators, and the disassembling delay of validators is not predetermined, but rather contingent upon network demand. Consequently, withdrawls might encounter delays during periods of `Ethereum` network congestion.