BLOCKSEC

Security Audit Report for Staking Contracts

Date: April 17, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	MegaETH
Target	Staking Contracts

Version History

Version	Date	Description
1.0	April 17, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-20 09:17:12+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 the Audit Target

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

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Staking Contracts of MegaETH.

The Staking Contracts of MegaETH is a KPI-based staking system for the MEGA token on the MegaETH chain. Users can stake MEGA tokens and earn MEGA tokens based on achieved KPI milestones (i.e., tranches). Each tranche is created with a reward token and maturity curves that scale rewards according to the staking duration. This update introduces two key features: (1) a recommit mechanism, enabling users to claim rewards as new stakes; (2) proportional releases of locked staking rewards.

Note this audit only focuses on the smart contracts in the following directories/files. Code prior to and including the baseline version 0, where applicable, is outside the scope of this audit and assumes to be reliable and secure.

- src/abstracts/MegaStakingCore.sol
- src/abstracts/MegaStakingLocked.sol
- src/abstracts/MegaStakingStake.sol
- src/abstracts/MegaStakingView.sol
- src/interfaces/IError.sol
- src/lib/RewardCalculationLib.sol
- src/MegaStaking.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

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. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
Staking Contracts	`Version 0`	`c8ace833af6b246f3cbbf13d1678392b1ff010a4`
	`Version 1`	`75e292cf78c545459001af141add55f8c150dba8`
	`Version 2`	`cefa83744a52f01e21e0f69294414a42dd2e49e8`

[1] https://github.com/megaeth-labs/mega-staking-contract.git

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 Target, 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 the Target.

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 the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target 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 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness

- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 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 five categories:

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

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

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

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **two** notes.

- Low Risk: 1
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Lack of pause state checks in the recommit staking mechanism	Security Issue	Fixed
2	-	Ensure proper setup for the reward token	Note	-
3	-	Ensure proper off-chain analysis before updating locked staking records	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of pause state checks in the recommit staking mechanism

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MegaStakingCore`, the function `_finalizeRecommit()` creates a new stake based on users' unclaimed rewards. Specifically, users are allowed to trigger the recommit mechanism via claim operations (e.g.,`claimRewards()` and `claimAllRewards()`). However, there is a lack of staking pause state check (i.e., the function `_requireNotPausedStaking()`). As a result, users are allowed to create staking records when staking operations are banned.

```
577   function _finalizeRecommit(bool _recommit) internal {
578       if (!_recommit) return;
579
580       bytes32 slot = _RECOMMIT_AMOUNT_SLOT;
581       uint256 totalAmount;
582       assembly {
583           totalAmount := tload(slot)
584       }
585       if (totalAmount == 0) return;
586
587       // Clear transient before state changes (supports sequential external calls in same tx)
588       assembly {
589           tstore(slot, 0)
590       }
591
592       uint128 amount128 = totalAmount.toUint128();
593       MegaStakingStorage storage $ = _getMegaStakingStorage();
594
595       UserStakeRecord storage record = $.userStakeRecords[msg.sender];
596       uint32 stakeId = record.nextStakeId;
```

```
597
598         $.userStakes[msg.sender][stakeId] = UserStakeInfo({
599             stakeId: stakeId,
600             startTime: block.timestamp.toUint32(),
601             amount: amount128,
602             nextUnclaimedDistributedTrancheIndex: $.distributedTrancheIds.length.toUint32(),
603             pendingRewardsStartIndex: $.distributedTrancheIds.length.toUint32(),
604             stakeType: StakeType.RECOMMIT
605         });
606
607         record.activeStakeIds.add(uint256(stakeId));
608         record.nextStakeId++;
609         $.totalStakedAmount += amount128;
610
611         emit Staked(msg.sender, stakeId, amount128, StakeType.RECOMMIT);
612     }
```

Listing 2.1: src/abstracts/MegaStakingCore.sol

```
68     function stake(uint128 _amount) external notZeroAmount(_amount) nonReentrant {
69         _requireNotPausedStaking();
70         _stake(_amount, StakeType.REGULAR);
71     }
```

Listing 2.2: src/abstracts/MegaStakingStake.sol

Impact Users are allowed to create staking records when staking operations are banned

Suggestion Add a `_requireNotPausedStaking()` check in claiming operations when `_recommit` is true.

2.2 Note

2.2.1 Ensure proper setup for the reward token

Introduced by `Version 1`

Description The project states that the only supported reward token is `MEGA`. Consequently, the `require(rewardToken == address(MEGA_TOKEN), RecommitNonMegaReward())` guard in `_transferReward()` and `_transferLockedStakingReward()` will always pass in practice, and users' reward recommit operations cannot be blocked by non-MEGA reward tokens.

```
446         if (_recommit) {
447             // Recommit mode: create a new stake from reward tokens (must be MEGA)
448             require(rewardToken == address(MEGA_TOKEN), RecommitNonMegaReward());
449             tranche.trancheAvailableRewardAmount -= reward128;
450             $.userTotalRewardsClaimed[msg.sender][rewardToken] += reward;
451             _accumulateRecommit(reward128);
452             emit RewardsClaimed(msg.sender, _stakeId, trancheId, reward, rewardToken);
453         } else if (_allowFailure) {
```

Listing 2.3: src/abstracts/MegaStakingStake.sol

```
524         if (_recommit) {
525             require(tranche.rewardToken == address(MEGA_TOKEN), RecommitNonMegaReward());
526             tranche.trancheAvailableRewardAmount -= reward128;
527             $.userTotalRewardsClaimed[msg.sender][tranche.rewardToken] += _reward;
528             _accumulateRecommit(reward128);
529             emit LockedStakingRewardsClaimed(_lockedStakingId, msg.sender, _trancheId, _reward,
                    tranche.rewardToken);
530         } else if (_allowFailure) {
```

Listing 2.4: src/abstracts/MegaStakingLocked.sol

2.2.2 Ensure proper off-chain analysis before updating locked staking records

Introduced by Version 1

Description The functions `updateUnstakingCurve()`, `updateLockedStakingAmount()`, and `updateLockedStakingCurve()` in the contract `MegaStakingLocked` allow the `LOCKED_STAKING_MANAGER` to modify reward-relevant parameters at any time without on-chain guards that check whether any referencing tranches have already reached `DISTRIBUTED` status. In contrast, the function `updateMaturityCurve()` of the contract `MegaStakingKPI` reverts if the target curve is used by any `UNDER_REVIEW` or `DISTRIBUTED` tranches. Since the function `_computeLockedReward()` re-computes the reward amount (i.e., `fullReward`) on every claim using the latest parameters, related updates can retroactively alter reward outcomes. Therefore, the project must ensure proper offchain analysis before invoking aforementioned functions.

```
239     function updateLockedStakingAmount(uint32 _lockedStakingId, uint128 _newAmount)
240         external
241         onlyRole(LOCKED_STAKING_MANAGER)
242     {
243         require(_newAmount >= MIN_LOCKED_STAKING_AMOUNT, InsufficientLockedStakingAmount());
244
245         MegaStakingStorage storage $ = _getMegaStakingStorage();
246
247         // Verify locked staking exists
248         LockedStaking storage lockedStaking = $.lockedStakings[_lockedStakingId];
249         require(lockedStaking.beneficiary != address(0), LockedStakingNotExists());
250
251         uint128 oldAmount = lockedStaking.initialStakedAmount;
252         lockedStaking.initialStakedAmount = _newAmount;
253
254         emit LockedStakingAmountUpdated(_lockedStakingId, oldAmount, _newAmount);
255     }
```

Listing 2.5: src/abstracts/MegaStakingLocked.sol

```
55     function updateUnstakingCurve(uint32 _unstakingCurveId, UnstakingCurveLib.Point[] calldata
           _points)
56         external
57         onlyRole(LOCKED_STAKING_MANAGER)
58     {
59         MegaStakingStorage storage $ = _getMegaStakingStorage();
```

```
60
61      // Verify curve already exists
62      require($.unstakingCurves[_unstakingCurveId].points.length > 0, UnstakingCurveNotSet());
63
64      // Validate the points
65      UnstakingCurveLib.validate(_points);
66
67      // Store the curve points as packed uint64 values
68      delete $.unstakingCurves[_unstakingCurveId].points;
69      for (uint256 i = 0; i < _points.length; i++) {
70          uint64 packed = UnstakingCurveLib.packPoint(_points[i].timestamp, _points[i].ratio);
71          $.unstakingCurves[_unstakingCurveId].points.push(packed);
72      }
73
74      emit UnstakingCurveUpdated(_unstakingCurveId, _points);
75  }
```

Listing 2.6: src/abstracts/MegaStakingLocked.sol

```
184 function updateLockedStakingCurve(uint32 _lockedStakingId, uint32 _newUnstakingCurveId)
185     external
186     onlyRole(LOCKED_STAKING_MANAGER)
187 {
188     MegaStakingStorage storage $ = _getMegaStakingStorage();
189
190     // Verify locked staking exists
191     LockedStaking storage lockedStaking = $.lockedStakings[_lockedStakingId];
192     require(lockedStaking.beneficiary != address(0), LockedStakingNotExists());
193
194     // Verify new unstaking curve exists
195     require($.unstakingCurves[_newUnstakingCurveId].points.length > 0, UnstakingCurveNotSet());
196
197     uint32 oldCurveId = lockedStaking.unstakingCurveId;
198     lockedStaking.unstakingCurveId = _newUnstakingCurveId;
199
200     emit LockedStakingCurveUpdated(_lockedStakingId, oldCurveId, _newUnstakingCurveId);
201 }
```

Listing 2.7: src/abstracts/MegaStakingLocked.sol


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS