TRIBYTE

# AMR PROTOCOL

## SECURITY AUDIT REPORT

JUN 2025

**Every Byte Builds Immutable Trust**

No. 202506120002

# Contents

# Summary

## Executive Summary

Tribyte performed a comprehensive security audit of the AMR Protocol contracts throughout June 2025, with the objective of identifying vulnerabilities and ensuring the robustness of the codebase. The audit encompassed both core contracts and their dependencies, delivering a thorough evaluation of the project's security posture.

The assessment leveraged a dual methodology, combining Manual Review and Static Analysis to meticulously examine the contracts. Our team adopted a multifaceted testing strategy, integrating black-box, gray-box, and white-box techniques to simulate real-world attack scenarios and detect potential weaknesses. Black-box testing evaluated the contracts from an external attacker's perspective, gray-box testing probed internal behaviors using specialized scripting tools, and white-box testing featured an in-depth, line-by-line code review by our experts.

## Project Summary

| | |
|---|---|
| **Project Name** | AMR Protocol |
| **Language** | Solidity |
| **Github Link** | https://github.com/linus994433/amr-contracts |
| **Commit Hash** | d4d7cc3035642c6ac354476dde06167aac64e908 |
| **Deployment Address (BSC)** | AMR Token: 0x232a7a48d1dd946617d82fab36b46a30f69df4a3<br>AMRStaking: 0x5fa78db915aecbdfd502affda277cb9632c8d4c2<br>LiquidityStaking: 0x76812D914B346506f14f7F51068eAA7F2f7c6D49 |

Note: The implementation contracts for AMRStaking and LiquidityStaking are not verified on BscScan. It remains the responsibility of the AMR Protocol team to ensure that the deployed contracts align with the audited versions.

## Vulnerability Summary

| Severity Level | Summary |
|---|---|
| Critical | |
| High | 2 |
| Medium | 3 |

| | |
|---|---|
| Low | 6 |
| Informational | 3 |

# Findings

## [CC-01] Non-Compliant Code Comments

| | |
|---|---|
| Category | Coding Convention |
| Severity | Low |
| Location | src/interfaces/IPancakeRouter.sol<br>src/libraries/InterestLib.sol<br>src/libraries/PriceLib.sol<br>src/libraries/StakingConstants.sol<br>src/libraries/StakingConstants.testnet.sol<br>src/upgrades/AMRStaking.5-8.sol<br>src/AMRStaking.sol<br>src/AMRStaking.testnet.sol<br>src/LiquidityStaking.sol<br>src/LiquidityStaking.testnet.sol<br>src/TestZYRAToken.sol |

### Description

The code contains comments written in Chinese and does not strictly adhere to the NatSpec (Natural Specification) standard. Given that contract source code, including comments, is publicly exposed upon verification on platforms like Etherscan or BscScan, non-English comments may confuse international developers and auditors, potentially undermining the project's professionalism.

### Recommendation

We recommend adopting the NatSpec convention for all code comments to enhance clarity and consistency. Additionally, comments should be written in English to ensure accessibility and professionalism, especially since they will be publicly visible upon contract verification. Standardized and well-documented code improves maintainability, facilitates audits, and strengthens the project's brand reputation within the global blockchain community.

## [CD-01] Suboptimal Contract Code Structure

| | |
|---|---|
| Category | Contract Design |
| Severity | Low |
| Location | src/libraries/StakingConstants.sol<br>src/libraries/StakingConstants.testnet.sol<br>src/upgrades/AMRStaking.5-8.sol<br>src/AMRStaking.sol<br>src/AMRStaking.testnet.sol<br>src/LiquidityStaking.sol<br>src/LiquidityStaking.testnet.sol |

### Description

The codebase exhibits significant code duplication across multiple contracts, which negatively impacts maintainability and increases the overall contract size during deployment. This redundancy is particularly evident in the overlapping logic between contracts, and the presence of duplicated code across testnet and mainnet versions further exacerbates these issues, introducing inconsistencies and inefficiencies.

### Recommendation

We recommend refactoring the codebase to reduce duplication and enhance efficiency. Specifically, the substantial overlap between AMRStaking.sol and LiquidityStaking.sol should be consolidated into a shared library to minimize redundancy and improve maintainability. For AMRStaking.testnet.sol and its counterparts (AMRStaking.sol and AMRStaking.5-8.sol), as well as LiquidityStaking.testnet.sol and LiquidityStaking.sol, the differences appear to stem primarily from parameter variations. We suggest using a single contract with configurable parameters set during deployment or initialization. This approach ensures consistent logic across testnet and mainnet environments, reduces deployment size, and simplifies future upgrades.

## [CD-02] Uninitialized Implementation Contracts in UUPS Proxy Pattern

| | |
|---|---|
| Category | Contract Design |
| Severity | High |
| Location | src/AMRStaking.sol<br>src/AMRStaking.testnet.sol<br>src/LiquidityStaking.sol<br>src/LiquidityStaking.testnet.sol |

### Description

The AMRStaking and LiquidityStaking contracts utilize OpenZeppelin's UUPS (Universal Upgradeable Proxy Standard) proxy pattern to support contract upgrades. However, they fail to adhere to OpenZeppelin's best practices, which mandate calling _disableInitializers() within the constructor of the implementation contracts to prevent direct initialization. Without this safeguard, an attacker could directly invoke the initialize() method on the AMRStaking or LiquidityStaking implementation contracts. This could lead to unauthorized state modifications in the implementation, potentially impacting the storage state of the proxy contract and introducing unpredictable risks, such as storage collisions or privilege escalation.

### Recommendation

To mitigate this vulnerability, we recommend adding the following constructor to both AMRStaking and LiquidityStaking contracts, ensuring compliance with UUPS best practices:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Additionally, utilize the OpenZeppelin openzeppelin-foundry-upgrades plugin for contract deployment and upgrades. This plugin facilitates secure proxy deployment and initialization, ensuring that the initialize() method is called only through the proxy, thereby protecting the contract's state integrity and upgradeability.

## [GV-01] Unbounded Array Traversal Leading to Potential Out-of-Gas Errors

| | |
|---|---|
| Category | General Vulnerability |
| Severity | High |
| Location | src/LiquidityStaking.sol: #L255, #L394, #L527, #L622, #L713, #L795, #L845, #L1120, #L1133, #L1149, #L1166, #L1185, #L1206, #L1343, #L1396, #L1423, #L1441<br>src/LiquidityStaking.testnet.sol: #L252, #L390, #L520, #L615, #L701, #L783, #L833, #L1108, #L1121, #L1137, #L1154, #L1173, #L1194, #L1331, #L1384, #L1411 |

### Description

The codebase contains multiple instances of array traversal where array lengths are unrestricted, posing a significant risk of infinite growth. This issue is exacerbated by nested loop structures within certain code branches, amplifying gas consumption. Both Ethereum Mainnet and BSC Mainnet enforce a block gas limit of approximately 30,000,000 gas, and individual transaction gas limits can easily reach this ceiling, leading to transaction reversion.

A representative example is the liquidWithdraw() function in LiquidityStaking.sol and its called functions:

```
function liquidWithdraw() external whenNotPaused nonReentrant {
    // ...
    for (uint256 i = 0; i < liquidStakes[msg.sender].length; i++) {
        LiquidStakeInfo storage stakeInfo = liquidStakes[msg.sender][i];
        _compoundLiquidRewards(msg.sender, i);
        // ...
    }
    // ...
}

function _compoundLiquidRewards(address _user, uint256 _stakeIndex) internal
{
    LiquidStakeInfo storage stakeInfo = liquidStakes[_user][_stakeIndex];
    if (stakeInfo.stakedAmount == 0) return;
    uint256 newRewards = _calculateLiquidRewards(_user, _stakeIndex) -
stakeInfo.pendingRewards;
    // ...
    for (uint256 i = 0; i < completePeriods; i++) {
        uint256 periodStartTime = stakeInfo.lastUpdateTime + (i *
COMPOUND_PERIOD);
        uint256 applicableRate = _getLiquidRewardRateAt(periodStartTime);
        // ...
    }
}

function _calculateLiquidRewards(address _user, uint256 _stakeIndex)
internal view returns (uint256) {
    // ...
    for (uint256 i = 0; i < completePeriods; i++) {
        uint256 periodStartTime = stakeInfo.lastUpdateTime + (i *
COMPOUND_PERIOD);
        uint256 periodEndTime = periodStartTime + COMPOUND_PERIOD;
        uint256 applicableRate = _getLiquidRewardRateAt(periodStartTime);
        // ...
     }
}

function _getLiquidRewardRateAt(uint256 _timestamp) internal view returns
(uint256) {
    // ...
    for (uint256 i = liquidRewardRateHistory.length - 1; i > 0; i--) {
        if (_timestamp >= liquidRewardRateHistory[i].timestamp) {
            return liquidRewardRateHistory[i].rewardRate;
        }
    }
    // ...
}
```

The liquidWithdraw() function iterates over every liquidStakes[msg.sender] entry, calling _compoundLiquidRewards() to compute rewards for each stake. _compoundLiquidRewards() further loops through completePeriods, invoking _getLiquidRewardRateAt() to determine the reward rate, which in turn iterates over liquidRewardRateHistory. This three-level nested loop structure, where liquidStakes[msg.sender].length, completePeriods, and liquidRewardRateHistory.length can theoretically grow indefinitely — especially liquidStakes[msg.sender] — increases the likelihood of out-of-gas errors over time. As the contract operates, the gas required for a single liquidWithdraw() transaction may exceed limits, ultimately rendering the function unusable.

## Recommendation

To mitigate this vulnerability, we advise avoiding array traversals, particularly for arrays with potentially unbounded growth, due to their uncontrollable gas consumption. If traversal is unavoidable, the following measures should be implemented:

- **Impose Length Restrictions:** Limit the maximum number of stakes per user in liquidStakes (e.g., to 10) to cap gas usage.
- **Provide Targeted Access Methods:** Introduce functions to process specific or a limited number of array elements, such as liquidWithdraw(uint256 stakeIndex) for a single stake or liquidWithdraw(uint256 startIndex, uint256 count) for a defined range of stakes.
- **Offchain Logic:** Migrate portions of the computation (e.g., reward calculations) to off-chain services like a subgraph, reducing on-chain gas demands and enhancing scalability.

## [GV-02] Oracle Price Manipulation Risk

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Medium |
| Location | src/libraries/PriceLib.sol: #L41 |

### Description

The getUnitTokenPrice function in PriceLib.sol retrieves the instantaneous price of $AMR relative to $USDT directly from the $AMR-$USDT trading pair via the PancakeSwap router. The implementation is as follows:

```
function getUnitTokenPrice(
    IPancakeRouter router,
    address tokenIn,
    address tokenOut
) internal view returns (uint256) {
    uint256 oneToken = 10**18;

    address[] memory path = new address[](2);
    path[0] = tokenIn;
    path[1] = tokenOut;

    uint256[] memory amounts = router.getAmountsOut(oneToken, path);
    return amounts[1];
}
```

This function relies on the current block's reserves, which are subject to real-time changes due to trades, liquidity provision (mint), or removal (burn). Attackers can exploit this vulnerability using flash loans to manipulate reserves within the same block. For instance, an attacker could borrow a large amount of $USDT, purchase $AMR to inflate its price, and then sell back the $AMR to restore the initial state, all within one transaction. During this manipulation, the attacker could invoke AMRStaking.stake() to stake $AMR at a significantly reduced cost in $USDT terms.

Although AMRStaking.stake() incorporates a default 10% price tolerance range, this safeguard is insufficient to prevent manipulation. An attacker could still use a flash loan to drastically increase the $AMR price, enabling the router to return a $USDT amount that falls within the ±10% tolerance of the target stake value, thereby successfully

executing the stake with minimal $AMR expenditure.

### Recommendation

We recommend integrating a trusted oracle solution, such as Chainlink or Redstone, to obtain reliable $AMR price feeds and mitigate manipulation risks. If no suitable oracle price feed is available and reliance on the PancakeSwap trading pool is unavoidable, we suggest developing a custom Oracle contract based on historical price data. For guidance, refer to the Uniswap V2 documentation on building an oracle: https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle. This approach should implement a time-weighted average price (TWAP) to smooth out instantaneous price fluctuations and enhance resistance to flash loan attacks.

## [GV-03] Unclaimed Airdrop Loss During Stake Operations

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Medium |
| Location | src/AMRStaking.sol: #L354<br>src/AMRStaking.testnet.sol: #L348<br>src/upgrades/AMRStaking.5-8.sol: #L373 |

### Description

The contract contains a vulnerability related to the handling of unclaimed airdrops during staking operations. The relevant functions are as follows:

```
function stake(
        NodeType nodeType,
        uint256 amrAmount,
        address inviter
) external whenNotPaused nonReentrant notLocked {
        //...
        require(userStakes[msg.sender].isActive == false, "Already staked");
        //...
        uint256 airdropAmount = (amrAmount *
        nodeConfigs[nodeType].airdropPercentage) / 10000;
        userAirdrops[msg.sender] = AirdropInfo({
            airdropAmount: airdropAmount,
            startTime: currentTime,
            claimed: 0
        });
}

function withdrawPrincipal(uint256 amount) external nonReentrant notLocked {
    // ...
    if (userStake.amrAmount == 0) {
        userStake.isActive = false;
    }
    // ...
}

function claimAirdrop(uint256 amount) external nonReentrant notLocked {
    // ...
}
```

If a user has previously staked and both the principal and airdrop are fully unlocked, they can call withdrawPrincipal() to fully withdraw their principal, setting userStake.isActive to false. If the user then initiates a new stake via stake() without first claiming their unclaimed airdrop, the existing userAirdrops[msg.sender] is overwritten with the new airdrop details. This overwrite results in the loss of the unclaimed airdrop, depriving the user of their entitled rewards.

### Recommendation

To address this vulnerability, we recommend implementing one of the following solutions:

- **Automatic Airdrop Claim on Withdrawal:** Modify withdrawPrincipal() to automatically claim any unclaimed airdrop for the user before updating userStake.isActive, ensuring no rewards are lost.
- **Airdrop Preservation on New Stake:** Update stake() to check for any unclaimed airdrop associated with the user and merge it into the new airdrop allocation, preserving the user's previous rewards.

These adjustments will protect user assets and enhance the contract's fairness and reliability.

### [GV-04] Excessive Permissions in emergencyWithdraw Function

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Low |
| Location | src/AMRStaking.sol: #L1038<br>src/AMRStaking.testnet.sol: #L1034<br>src/upgrades/AMRStaking.5-8.sol: #L1227 |

#### Description

The emergencyWithdraw function, defined as follows, grants excessive privileges to holders of the SUPER_ADMIN_ROLE:

```
function emergencyWithdraw(
    address token,
    address receiver,
    uint256 amount
) external onlyRole(SUPER_ADMIN_ROLE) {
    require(receiver != address(0), "Invalid receiver");
    IERC20(token).safeTransfer(receiver, amount);
    emit EmergencyWithdraw(token, receiver, amount);
}
```

This function allows the admin to transfer any ERC-20 token, including user-staked $AMR, to any address at any time, provided the receiver is not the zero address. The lack of additional safeguards or restrictions means that user assets, such as those staked via the contract, are not adequately protected. This overreach poses a risk of unauthorized asset withdrawal, undermining the security and trust in the protocol.

#### Recommendation

To mitigate this vulnerability, we recommend implementing additional checks and restrictions on the emergencyWithdraw function. Specifically, if the token is $AMR, withdrawals should be limited to amounts exceeding the total staked amount (trackable via a variable such as totalStakedAmount). This ensures that only surplus or unallocated $AMR can be withdrawn, preserving user-staked assets and enhancing the contract's security posture.

## [GV-05] Insufficient Validation of recoverSigner Output

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Low |
| Location | src/AMRStaking.sol: #L680<br>src/AMRStaking.testnet.sol: #L688<br>src/upgrades/AMRStaking.5-8.sol: #L868 |

### Description

The claimRewardsWithSignature function contains a potential vulnerability related to signature validation, as shown below:

```
function claimRewardsWithSignature(
    uint256 amount,
    LockPeriod lockPeriod,
    bytes memory signature,
    bytes32 orderId,
    uint256 deadline
) external whenNotPaused nonReentrant notLocked {
    // ...
    address recoveredSigner = recoverSigner(ethSignedMessageHash,
signature);
    require(recoveredSigner == signerAddress, "Invalid signature");
    // ...
}
```

The recoverSigner function, which utilizes ecrecover to derive the signer's address from a signature, may return address(0) if the signature is invalid (e.g., malformed or tampered). If signerAddress is not explicitly set or initialized, this could allow the signature validation check to be bypassed, enabling unauthorized claims of rewards. This oversight undermines the intended security of the signature-based access control.

### Recommendation

To address this vulnerability, we recommend adding a check to ensure recoveredSigner is not address(0) before proceeding with validation. Alternatively, consider leveraging OpenZeppelin's ECDSA library for robust signature

verification, which includes built-in safeguards against such edge cases. This will enhance the reliability of the signature validation process and prevent potential exploitation.

## [GV-06] Missing Chain ID and Nonce in Signature Validation

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Low |
| Location | src/AMRStaking.sol: #L609<br>src/AMRStaking.testnet.sol: #L609<br>src/upgrades/AMRStaking.5-8.sol: #L789<br>src/LiquidityStaking.sol: #L354, #L436, #L1298<br>src/LiquidityStaking.testnet.sol: #L350, #L429, #L1286 |

### Description

For example, the claimRewardsWithSignature function in AMRStaking exhibits a vulnerability due to inadequate signature construction, as illustrated below:

```
function claimRewardsWithSignature(
    uint256 amount,
    LockPeriod lockPeriod,
    bytes memory signature,
    bytes32 orderId,
    uint256 deadline
) external whenNotPaused nonReentrant notLocked {
    require(block.timestamp <= deadline, "Signature expired");

    bytes32 messageHash = keccak256(
        abi.encodePacked(
            msg.sender,
            amount,
            lockPeriod,
            orderId,
            deadline
        )
    );
    bytes32 ethSignedMessageHash = keccak256(
        abi.encodePacked("\x19Ethereum Signed Message:\n32", messageHash)
    );

    require(!usedSignatures[ethSignedMessageHash], "Signature already
used");

    // ...
 }
```

The signature data lacks inclusion of the chain ID, which poses a risk if the contract is deployed across multiple chains. In such cases, the same signature could be replayed on different chains, enabling unauthorized reward claims. Additionally, while usedSignatures prevents signature reuse within the same chain, it does not enforce the sequential order of signature usage, leaving the system vulnerable to unintended reprocessing.

### Recommendation

To mitigate these risks, we recommend enhancing the signature scheme as follows:

- **Include Chain ID:** Incorporate the chain ID in the signed message to prevent cross-chain replay attacks.
- **Implement Nonce:** Add a nonce value to the signature data to ensure sequential usage and prevent single-chain replay attacks, tracking it per user (e.g., via a nonces mapping) and incrementing it with each valid claim.

These changes will strengthen the security of the signature validation process across multi-chain deployments.

## [GV-07] Potential Access to Invalid Data in withdrawReward

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Medium |
| Location | src/AMRStaking.sol: #L735<br>src/AMRStaking.testnet.sol: #L731<br>src/upgrades/AMRStaking.5-8.sol: #L911 |

### Description

The withdrawReward function in the contract poses a vulnerability due to inadequate validation of the claimIndex parameter. The relevant data structures and function implementation are as follows:

```
mapping(address => mapping(uint256 => RewardClaim)) public userRewardClaims;
mapping(address => uint256) public userRewardClaimCount;

function withdrawReward(
    uint256 claimIndex
) external nonReentrant notLocked {
    RewardClaim storage claim = userRewardClaims[msg.sender][claimIndex];
    require(!claim.claimed, "Already claimed");
    require(block.timestamp >= claim.releaseTime, "Not released yet");

    claim.claimed = true;
    //...
}
```

The function does not verify whether claimIndex falls within a valid range. Since userRewardClaims[msg.sender] is a mapping, accessing an out-of-range claimIndex does not trigger an error but instead returns an empty RewardClaim struct. By default, this struct has claimed set to false and releaseTime set to 0. While the amount field being 0 prevents asset loss, executing the function with an invalid claimIndex sets claimed to true, resulting in wasted gas and pollution of the storage slot. This behavior could lead to unintended state changes and operational inefficiencies.

### Recommendation

To mitigate this vulnerability, we recommend adding a validation check to ensure claimIndex is within a valid range, such as requiring claimIndex < userRewardClaimCount[msg.sender]. This will prevent access to uninitialized or invalid data. Alternatively, if the withdrawReward function is not actively utilized, we suggest removing it entirely to eliminate the associated risks and reduce contract complexity.

## [GV-08] Inefficient Linear Search in _getPeriodRewardRateAt

| | |
|---|---|
| Category | General Vulnerability |
| Severity | Low |
| Location | src/LiquidityStaking.sol: #L1412<br>src/LiquidityStaking.testnet.sol: #L1400 |

### Description

The _getPeriodRewardRateAt function employs a linear search to determine the reward rate, as shown below:

```
function _getPeriodRewardRateAt(uint256 _period, uint256 _timestamp)
internal view returns (uint256) {
    // ...
    for (uint256 i = periodRewardRateHistory[_period].length - 1; i > 0; i-
-) {
        if (_timestamp >= periodRewardRateHistory[_period][i].timestamp) {
            return periodRewardRateHistory[_period][i].rewardRate;
        }
    }
    // ...
}
```

This function iterates through the entire periodRewardRateHistory[_period] array, resulting in a time complexity of O(n). Such a linear search is inefficient, particularly as the history grows, leading to increased gas costs and potential performance bottlenecks.

### Recommendation

To enhance efficiency, we recommend replacing the linear search with a binary search algorithm. This would reduce the time complexity to O(log n), significantly improving gas efficiency and scalability.

## [CC-02] Unnecessary Getter Functions

| Category | Coding Convention |
| --- | --- |
| Severity | Informational |
| Location | src/AMRStaking.sol: #L914, #L922, #L930<br>src/AMRStaking.testnet.sol: #L910, #L918, #L926<br>src/upgrades/AMRStaking.5-8.sol: #L1090, #L1098, #L1106 |

### Description

The contract defines the following state variables with public visibility, which automatically generates corresponding getter functions:

```
uint256 public totalStakedAmount;
uint256 public totalPlatinumStakedAmount;
uint256 public totalPlatinumPlusStakedAmount;
```

Additionally, explicit getter functions have been implemented as follows:

```
function getTotalStakedAmount() public view returns (uint256) {
    return totalStakedAmount;
}

function getTotalPlatinumStakedAmount() public view returns (uint256) {
    return totalPlatinumStakedAmount;
}

function getTotalPlatinumPlusStakedAmount() public view returns (uint256) {
    return totalPlatinumPlusStakedAmount;
}
```

Since declaring these variables as public automatically generates getter functions with the same names, the explicit getter implementations are redundant. This duplication increases code complexity and maintenance overhead

without providing additional functionality.

### Recommendation

We recommend removing the redundant getter functions (getTotalStakedAmount, getTotalPlatinumStakedAmount, and getTotalPlatinumPlusStakedAmount) to streamline the codebase. Relying on the automatically generated getters will maintain the same accessibility while reducing unnecessary code and improving overall clarity.

## [CC-03] Missing Essential Event Emissions

| Category | Coding Convention |
| --- | --- |
| Severity | Informational |
| Location | src/AMRStaking.sol: #L711, #L723, #L997, #L1153<br>src/AMRStaking.testnet.sol: #L719, #L993, #L1146<br>src/upgrades/AMRStaking.5-8.sol: #L899, #L1186, #L1339<br>src/LiquidityStaking.sol: #L1001, #L1106<br>src/LiquidityStaking.testnet.sol: #L989, #L1094 |

### Description

The contract lacks event emissions for several state-changing functions, reducing traceability and transparency. For example, consider the updateSignerAddress function in AMRStaking.sol:

```
function updateSignerAddress(address _newSignerAddress)
    external
    onlyRole(SUPER_ADMIN_ROLE)
{
    require(_newSignerAddress != address(0), "Invalid signer address");
    signerAddress = _newSignerAddress;
}
```

This function updates a critical parameter (signerAddress) without emitting an event, hindering the ability to track changes on-chain. Similar methods that would benefit from event emissions include:

- AMRStaking.sol: updateSignerAddress, updateRewardReceiver, updateRedemptionWindow, updateMaxPeriodsToProcess
- AMRStaking.testnet.sol: updateSignerAddress, updateRedemptionWindow, updateMaxPeriodsToProcess
- AMRStaking.5-8.sol: updateSignerAddress, updateRedemptionWindow, updateMaxPeriodsToProcess
- LiquidityStaking.sol: setPeriodDiscount, updateSignerAddress
- LiquidityStaking.testnet.sol: setPeriodDiscount, updateSignerAddress

Additionally, the initialize() method, if it performs initial state setup, should also emit corresponding events to log these configurations.

### Recommendation

We recommend emitting appropriate events for all state-changing functions and the initialize() method to enhance contract transparency and traceability. For instance, adding an event like event SignerAddressUpdated(address newSignerAddress) for updateSignerAddress would allow off-chain monitoring and improve the protocol's security posture. This practice ensures that critical updates are logged on-chain, facilitating audits and user oversight.

## [CC-04] Missing Check for Duplicate Values in Setter/Updater Functions

| | |
|---|---|
| Category | Coding Convention |
| Severity | Informational |
| Location | src/LiquidityStaking.sol: #L208, #L1106<br>src/LiquidityStaking.testnet.sol: #L205, #L1094 |

### Description

The contract contains setter and updater functions that lack validation to prevent redundant updates. For example:

```
function setRewardReceiver(address _newReceiver) external onlyRole(ADMIN_ROLE) {
    require(_newReceiver != address(0), "Invalid receiver address");
    address oldReceiver = rewardReceiver;
    rewardReceiver = _newReceiver;
    emit RewardReceiverUpdated(oldReceiver, _newReceiver);
}


function updateSignerAddress(address _newSignerAddress)
    external
    onlyRole(SUPER_ADMIN_ROLE)
{
    require(_newSignerAddress != address(0), "Invalid signer address");
    signerAddress = _newSignerAddress;
}
```

The setRewardReceiver function checks that _newReceiver is not the zero address but does not verify whether _newReceiver differs from the existing rewardReceiver. Similarly, updateSignerAddress ensures _newSignerAddress is not the zero address but omits a check against the current signerAddress. This omission allows unnecessary updates to proceed, resulting in wasted gas and potentially misleading event emissions (e.g., RewardReceiverUpdated for identical values).

### Recommendation

We recommend adding a check to ensure the new value differs from the existing value before proceeding with the update. For instance, include a `require(_newReceiver != rewardReceiver, "No change in receiver address")` in `setRewardReceiver` and a `require(_newSignerAddress != signerAddress, "No change in signer address")` in `updateSignerAddress`. This will prevent redundant state changes, optimize gas usage, and maintain the integrity of event logs.

# Appendix

## Vulnerability Fix Status

| Status | Description |
| --- | --- |
| Resolved | The project team has successfully implemented a complete fix to address the vulnerability, eliminating the associated risks. All identified issues have been rectified, and the codebase has been updated to ensure the vulnerability no longer poses a threat. |
| Mitigated | The project team has taken steps to reduce the impact or likelihood of the vulnerability being exploited, but the issue has not been completely resolved. While the risk has been partially addressed, some potential exposure may still remain, and further action is recommended. |
| Acknowledged | The project team has reviewed and confirmed the existence of the vulnerability but has chosen not to address or mitigate it at this time. This status indicates awareness of the issue, and the team may accept the associated risks or plan to address it in the future. |
| Declined | The project team has reviewed the reported vulnerability but determined it does not require action, either because they believe it poses no significant risk to the project or because it falls outside the project's scope or priorities. The issue remains unaddressed, and the associated risks are accepted by the team. |

## Vulnerability Severity Level

| Level | Description |
| --- | --- |
| Critical | Critical severity vulnerabilities pose an immediate and severe threat to the project's security, potentially leading to significant loss of funds, unauthorized access, or complete system compromise. These issues must be addressed urgently before deployment or continued operation to ensure the safety of users and the integrity of the project. |
| High | High severity vulnerabilities can substantially impact the project's functionality, potentially enabling exploitation that results in loss of assets, data breaches, or disruption of critical operations. It is strongly recommended to prioritize and resolve these issues promptly to mitigate risks. |
| Medium | Medium severity vulnerabilities may affect the project's performance or security under specific conditions, potentially leading to inefficiencies, minor exploits, or degraded user experience. It is advisable to address these issues to enhance the overall robustness of the system. |

| | |
|---|---|
| Low | Low severity vulnerabilities have a minimal impact on the project's security or functionality and are unlikely to be exploited in typical scenarios. However, they may still pose theoretical risks. The project team should evaluate these issues and consider fixing them to improve long-term stability. |
| Informational | Informational findings do not directly impact the security or functionality of the project but highlight areas for improvement, such as adherence to best practices, code optimization, or architectural enhancements. Addressing these suggestions can lead to better maintainability and alignment with industry standards. |

## Audit Items

| Categories | Audit Items |
|---|---|
| Coding Convention | Obsolete Code |
| | Debug Code |
| | Comments / Dev Notes |
| | Compiler Versions |
| | License Identifier |
| | Require / Revert / Assert Usage |
| | Contract Size |
| | Gas Consumption |
| | Event Emission |
| | Parameter Check |
| General Vulnerability | Centralization |
| | Denial of Service |
| | Reply Attack |
| | Reentrancy Attack |
| | Race Conditions |
| | Integer Overflow / Underflow |
| | Arithmetic Accuracy Deviation |
| | Array Index Out of Bounds |

| | |
|---|---|
| | Receive / Fallback Function |
| | Payable and msg.value Usage |
| | tx.origin Authentication |
| | ERC20 Token Decimals |
| | ERC20 Safe Transfer |
| | ERC721 Safe Transfer |
| | Rebasable Token Support |
| | Native Token Support |
| | Storage / Memory Usage |
| | Function Permissions |
| External Dependency | Oracle Usage |
| | External Protocol Interaction |
| Protocol Design | Economics Design |
| | Formula Derivation |
| Contract Design | Factory Contract |
| | Proxy Usage |
| | EIP2535 Diamond Pattern Usage |
| | Upgradability / Pluggability |

## Disclaimer

Tribyte issues this audit report based solely on the code and materials provided by the client up to the report's issuance date. We assume the provided information is complete, accurate, and untampered. Tribyte is not liable for any losses or issues arising from incomplete, altered, or concealed information, or from changes made after the audit.

This report evaluates only the specified smart contracts or systems within the agreed scope, using Tribyte's tools and methodologies. It does not endorse the project's business model, team, or legal status, nor does it guarantee the absence of vulnerabilities due to technical limitations. The report is for the client's use only and may not be shared, quoted, or relied upon by third parties without Tribyte's written consent.