RECALL

Staking Contract

Security Assessment Report

Version: 2.0

July, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Recall components in scope. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Recall components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Recall components in scope.

Overview

This security engagement focuses on the recall-staking protocol's smart contract system, which facilitates token staking and reward claiming.

The architecture consists of three contracts: The `Staking` contract implements core staking and unstaking logic, while the `RewardAllocation` contract manages all reward distribution and claim mechanisms. Additionally, the `NftReceipt` contract handles NFT functionality related to staking actions.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the recall-staking-contracts repository.

The scope of this time-boxed review was strictly limited to the following files at commit ae3fbed: The fixes of the identified issues were assessed at commit b0ab097.

- `NftReceipt.sol`
- `RewardAllocation.sol`
- `Staking.sol`

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 6 issues during this assessment. Categorised by their severity:

- High: 2 issues.
- Low: 1 issue.
- Informational: 3 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Recall components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
RSC-01	Bypassing Stake Ownership Checks In `unlockedAll` Mode Allows Lock Of Funds	High	Resolved
RSC-02	Withdrawal Bypasses Unstaking And Cooldown	High	Resolved
RSC-03	Old Stake And NFT Not Cleared On Full-Amount Partial Relock	Low	Resolved
RSC-04	Withdrawals Are Allowed During Protocol Pause When `unlockedAll` Is False	Informational	Resolved
RSC-05	Merkle Root Duplicates Only Allow Single Claim	Informational	Resolved
RSC-06	Emergency Withdrawal Is Not Possible To Be Called Again	Informational	Resolved

RSC-01	Bypassing Stake Ownership Checks In `unlockedAll` Mode Allows Lock Of Funds		
Asset	`Staking.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When `unlockedAll` is set to `true`, the following checks are bypassed in the `withdraw()` function:

```
if (!unlockedAll) {
    if (block.timestamp < userStake.withdrawAllowedTime) revert NotUnstakedYet();
    if (!_tokenIds[msg.sender].remove(tokenId)) revert NotStakeOwner(tokenId);
}
```

The original intent was to allow all stakers to withdraw their tokens regardless of the `withdrawCooldown`. However, bypassing the ownership check (`_tokenIds[msg.sender].remove(tokenId)`) introduces a critical issue - a malicious staker can withdraw (and burn) another user's stake while claiming their own locked tokens, effectively locking the victim's funds in the protocol.

Assume three users have staked tokens as follows:

- **Alice**: Staked 40 tokens (owns NFT ID 1)
- **Bob**: Staked 60 tokens (owns NFT ID 2)
- **Charlie (attacker)**: Staked 100 tokens (owns NFT ID 3)

If the protocol is paused and `unlockedAll` is set to `true`, Charlie can exploit the vulnerability as follows:

1. Charlie calls `withdraw(1)` (NFT ID 1 belongs to Alice):
    - The contract deletes Alice's stake data.
    - Transfers 40 tokens to Charlie (instead of Alice).
    - Burns Alice's NFT (ID 1).
2. Charlie then calls `withdraw(2)` (NFT ID 2 belongs to Bob):
    - Bob's stake data is deleted.
    - 60 tokens are transferred to Charlie.
    - Bob's NFT (ID 2) is burned.
3. Result:
    - Charlie now has 100 tokens (his original stake).
    - Alice and Bob can no longer withdraw their stakes:
        - Calling `withdraw(1)` or `withdraw(2)` reverts (NFTs already burned).
        - Calling `withdraw(3)` reverts due to underflow (their staked amounts are less than Charlie's 100-token stake).

Recommendations

To prevent this, ensure the ownership check (`_tokenIds[msg.sender].remove(tokenId)`) is always be enforced, even when `unlockedAll` is `true` , e.g.:

```
function withdraw(uint256 tokenId) external nonReentrant {
    StakeInfo memory userStake = _stakeInfo[tokenId];

    if (!unlockedAll) {
        if (block.timestamp < userStake.withdrawAllowedTime) revert NotUnstakedYet();
    }

    if (!_tokenIds[msg.sender].remove(tokenId)) revert NotStakeOwner(tokenId);

    delete _stakeInfo[tokenId];

    totalUserStaked[msg.sender] -= userStake.amount;
    totalStaked -= userStake.amount;

    nftReceipt.burn(tokenId);
    stakeToken.safeTransfer(msg.sender, userStake.amount);

    emit Withdraw(msg.sender, tokenId, userStake.amount);
}
```

Resolution

The recommendation has been implemented in commit aa88c3c.

RSC-02	Withdrawal Bypasses Unstaking And Cooldown		
Asset	`Staking.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

In normal operation (when `unlockedAll` is `false`), users should first call `unstake()` and wait for the `withdrawCooldown` before withdrawing. However, the current implementation allows immediate withdrawal because `withdrawAllowedTime` is initialized to zero during staking, and zero `withdrawAllowedTime` is not checked during withdrawal:

```
_stakeInfo[newTokenId] = StakeInfo(
    uint256(amount),
    uint64(block.timestamp),
    uint64(lockupEndTime),
    0  // ← `withdrawAllowedTime` set to 0
);
```

As a result, if a user skips `unstake()` and calls `withdraw()` directly, the following check passes since `0 < block.timestamp`, allowing premature withdrawal:

```
if (!unlockedAll) {
    if (block.timestamp < userStake.withdrawAllowedTime) revert NotUnstakedYet();
    if (!_tokenIds[msg.sender].remove(tokenId)) revert NotStakeOwner(tokenId);
}
```

Recommendations

Enforce that `withdrawAllowedTime` must be non-zero (indicating `unstake()` was called) before allowing withdrawal:

```
if (!unlockedAll) {
    if (block.timestamp < userStake.withdrawAllowedTime || userStake.withdrawAllowedTime == 0)
        revert NotUnstakedYet();
    if (!_tokenIds[msg.sender].remove(tokenId))
        revert NotStakeOwner(tokenId);
}
```

Resolution

The recommendation has been implemented in commit 69f34b5.

RSC-03	Old Stake And NFT Not Cleared On Full-Amount Partial Relock		
Asset	`Staking.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A zero amount stake is left behind without cleanup, creating stale state and orphaned NFTs.

The protocol includes two functions for relocking staked tokens:

1. **Full Relock**:
   `function relock(uint256 tokenId, uint256 newLockDuration)`
   - Fully relocks the staked amount.
   - Burns the old stake info and its corresponding NFT.
2. **Partial Relock**:
   `function relock(uint256 tokenId, uint256 newLockDuration, uint256 newLockAmount)`
   - Relocks a portion of the staked amount.
   - Updates the old stake info, creates a new stake info and mints a new NFT for the new stake info.

During partial relocking, if `newLockAmount` equals `userOldStake.amount`, the old stake's amount is reduced to zero (`userOldStake.amount -= newLockAmount`). However, unlike full relocking, the old stake info and its NFT are not removed, leading to an inconsistency as a zero staked amount should result in the removal of the stake info and burning of the corresponding NFT.

Recommendations

Enforce that `newLockAmount` must be strictly less than `userOldStake.amount`. If full relocking is intended, users should call the dedicated `relock()` function without `newLockAmount`.

Resolution

The recommendation has been implemented in commit 431e5d2.

RSC-04	Withdrawals Are Allowed During Protocol Pause When `unlockedAll` Is False
Asset	`Staking.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When the `Staking` contract is paused (but `unlockedAll` is `false`), withdrawals should be temporarily disabled until the protocol resumes normal operation. However, the current implementation still permits withdrawals if the cooldown period has passed regardless of pausing status, bypassing the expected security measure.

```
if (!unlockedAll) {
    if (block.timestamp < userStake.withdrawAllowedTime) revert NotUnstakedYet();
    if (!_tokenIds[msg.sender].remove(tokenId)) revert NotStakeOwner(tokenId);
}
```

The issue is raised as informational as this is a design decision. There are trade-offs to both allowing or rejecting withdrawals during a pause, with the current implementation allowing users to withdraw their funds even when the protocol is paused.

Allowing withdrawals during a pause can be beneficial in preventing contract users from being locked out of their funds. However, it also poses risks if the pause is related to a security vulnerability or critical issue, such as a bug in the withdrawal logic allowing an attacker to drain the contract.

Recommendations

Consider whether withdrawals should be blocked or allowed while the contract is paused.

Resolution

This issue has been addressed in the commit c960b5c.

RSC-05	Merkle Root Duplicates Only Allow Single Claim
Asset	`RewardAllocation.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

If a Merkle root contains duplicate entries for the same address with different claim amounts, only one claim can be processed due to the claim tracking mechanism. This results in legitimate reward allocations becoming permanently inaccessible to users.

The `claim()` function tracks claims per user per root using the mapping `hasClaimed[root][msg.sender]` at line [**117**]:

```
if (hasClaimed[root][msg.sender]) {
    revert RewardAllocation__RewardAlreadyClaimedForThisAllocation({
        root: root,
        user: msg.sender
    });
}
```

Once a user claims any amount from a root, the flag is set to `true` at line [**139**], preventing any subsequent claims from the same root regardless of whether valid duplicate entries exist with different amounts.

Recommendations

Consider implementing leaf-specific claim tracking instead of user-per-root tracking. This would allow multiple valid claims from the same user within a single root, provided each corresponds to a unique leaf in the Merkle tree.

Resolution

This issue has been addressed in the commits bb6738b and 753c018.

RSC-06	Emergency Withdrawal Is Not Possible To Be Called Again
Asset	`RewardAllocation.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When a token is emergency withdrawn using `emergencyWithdraw()`, the contract marks `isTokenEmergencyWithdrawn[token]` as `true`. If some amounts of the token are transferred to the contract, they become permanently stuck due to the following check:

```
if (isTokenEmergencyWithdrawn[token]) {
    revert RewardAllocation__ThisTokenIsEmergencyWithdrawn(token);
}
```

This blocks any further withdrawals, even for newly deposited tokens.

Recommendations

Emergency withdrawals should remain available for any tokens held by the contract, regardless of prior withdrawals.

Resolution

This issue has been addressed in the commit ab4e36c.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

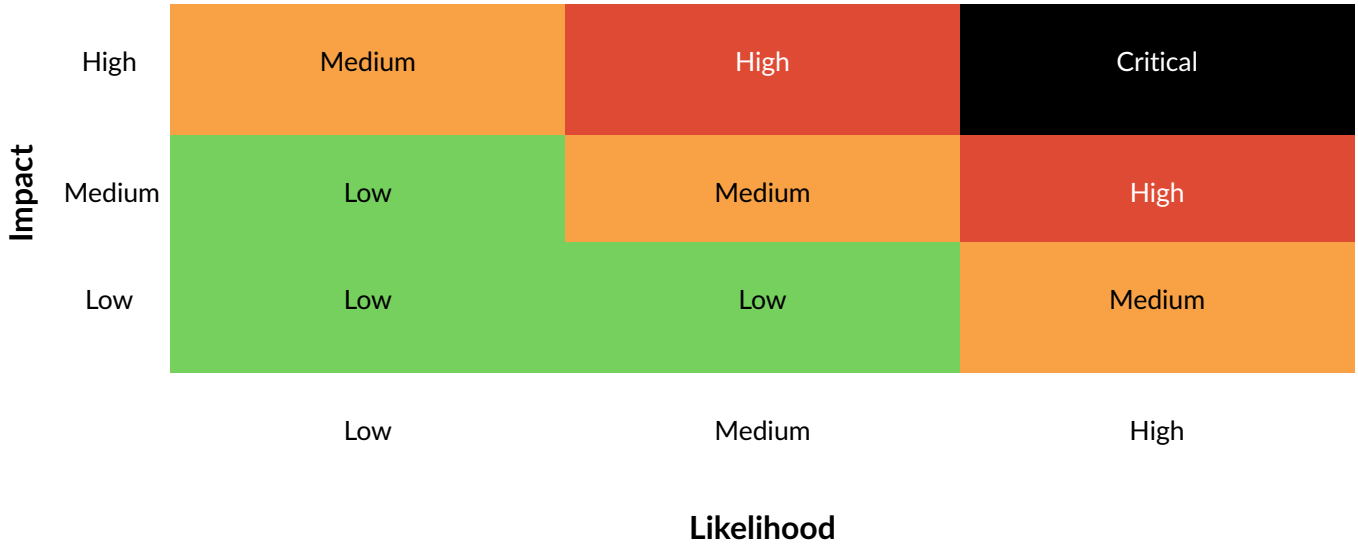

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].