Security Audit

Date : 2024-05-28

Staking Upgradable Audit

Auditor : Anjanay Raina

Executive Summary

Type	Smart Contract Audit
Audit Timeline	5 days
Runtime Environment	EVM
Languages	Solidity

Scope

/contracts/MainStaking.sol

/contracts/NativeStakingProxy.sol

/contracts/Staking.sol

/contracts/UpgradeStaking.sol

Summary of Findings

ID	Name	Description	Severity
H-01	Staking::distributeReward() may be made unusable by a malicious staker or end up exceeding the block gas limit.	Loop used for distributing rewards can be DOSed	High
M-01	Send ether with call instead of transfer	Transferring native tokens are recommended to be done using call instead of transfer	Medium
M-02	`OwnableUpgradeable` uses single-step ownership transfer	Single step ownership transfer done	Medium
M-03	Centralization Risk: Funds can be frozen when critical key holders lose access to their keys	Lost or stolen keys can pose a serious risk to the protocol	Medium
NC-01	Missing Event Logging for State-Changing Functions	Several state-changing functions in the contract do not log events for the changes they make	Non-Critical
G-01	Use assembly for 0 address checks	Using assembly for Null Address checks can help save gas on function calls	Gas
G-02	`<=` is cheaper than `<`	Strict inequalities (<) are more expensive than	Gas

		non-strict ones (<=). This is due to some supplementary checks (ISZERO, 3 gas)	
G-03	For Loop can be optimized for reducing gas	For loop can be optimized to reduce the gas cost	Gas
G-04	Using `calldata` instead of `memory` for read-only arguments saves gas		Gas
I-01	Incorrect or Misleading Documentation	Incorrect comments have been added in the code	Informational
I-02	Use of Informative Variable Names	The variable name $ used in the _disableInitializers function is not informative and does not convey its purpose or content.	Informational
I-03	Modifier Side Effects	Modifiers should only implement checks and validations inside of it and should not make state changes and external calls	Informational

Findings

[H-01] Staking::distributeReward() may be made unusable by a malicious staker or end up exceeding the block gas limit.

Severity: High

Location: Staking.sol , UpgradeStaking.sol

Description: The function distributeRewards uses a for loop to distribute the rewards of the stakers. The problem is that even if one of the stakers is malicious and reverts on receiving ether the whole transaction fails preventing valid stakers from receiving their rewards. Even if we assume that there are no malicious stakers in the protocol still if the length of the array of the stakers grows to a substantial amount, the function might exceed the block gas limit essentially keeping the protocol in a state of Denial of Service till some stakers unstake so that the length of the array gets small enough

Impact
The whole distributed reward functionality will be frozen in any case if there is a malicious staker or if the length of the staker array gets large enough. This hampers the core functionality of the protocol and the rewards will be stuck in the contract till the owner decides to call the 'emergencyDrain()' function.

Recommendation
Instead of using push transfers , consider using a pull transfer type of reward distribution structure where individual stakers can call a function to get the rewards that they are allocated. Such batch transfers always pose a risk of a DOS attack or can exceed the block gas limit.

Proof of Concept

```
function distributeReward(uint256 reward) internal {
      uint256 stakedAmountTotal = getTotalStakedAmount();
      for (uint256 i = 0; i < stakers.length; i++) {
        address staker = stakers[i];
        if (stakingAmount[staker] > 0) {
            uint256 userReward = (stakingAmount[staker] * reward) /
stakedAmountTotal;
            addressToRewardAmount[staker] += userReward;
            // Record the claim
            userToClaimedReward[staker].push(Claim({
                amount: userReward,
                timestamp: block.timestamp
            }));
            payable(staker).transfer(userReward);
```

```
            emit RewardDistributed(staker, userReward);
        }
    }
}
```

[M-01] Send ether with call instead of transfer

Severity: Medium

Location : Staking.sol , UpgradeStaking.sol

Description: In both of the withdraw functions, transfer() is used for native ETH withdrawal. The **transfer()** and **send()** functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example, EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction

Impact
The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

1) The claimer smart contract does not implement a payable function.
2) The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
3) The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Recommendation

Use call() instead of transfer()

Proof of Concept :

```
function emergencyDrainAmount(uint256 _amount) external onlyOwner {
    require(address(this).balance >= _amount, "Less Balance");
    payable(owner()).transfer(_amount);
}
```

Similar Findings :
https://solodit.xyz/issues/m-04-send-ether-with-call-instead-of-transfer-code4rena-redacted-cartel-redacted-cartel-contest-git

[M-02] `OwnableUpgradeable` uses single-step ownership transfer

Severity: Medium

Location : Staking.sol , UpgradeStaking.sol

Description: Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. The ownership pattern implementation for the protocol is in **OwnableUpgradeable.sol** where a single-step transfer is implemented.This can be a problem for all methods marked in **onlyOwner** throughout the protocol, some of which are core protocol functionality.

Impact
The emergency withdrawal functionality of the contract will be bricked meaning that no withdrawal can be made in case of a protocol failure or a hack.

Recommendation

It is a best practice to use two-step ownership transfer pattern, meaning ownership transfer gets to a "pending" state and the new owner should claim his new rights, otherwise the old owner still has control of the contract. Consider using OpenZeppelin's **Ownable2Step** contract

Proof of Concept :

```
function renounceOwnership() public virtual onlyOwner {
      _transferOwnership(address(0));
  }
```

[M-03] Centralization Risk: Funds can be frozen when critical key holders lose access to their keys

Severity: Medium

Location : Staking.sol , UpgradeStaking.sol

Description: The Owner can call important functions like 'renounceOwnership' and 'emergencyDrain' that store the owner of the function and withdraw funds from the protocol. This introduces a high centralization risk, which can cause funds to be frozen in case protocol malfunctions or the funds to be drained in case of stolen wallet keys.

Impact
The funds can get locked in the contract in case of lost keys and the key protocol parameters will also be at risk of getting changed as well as the funds being drained.

Recommendation

Ensure that the owner is a MultiSig wallet

Proof of Concept :

```
function renounceOwnership() public virtual onlyOwner {
      _transferOwnership(address(0));
  }
```

[NC-01] Missing Event Logging for State-Changing Functions

Severity: Non-Critical

Location: StakingUpgrade/NativeStakin.sol

Description: Several state-changing functions in the contract do not log events for the changes they make. Event logging is a best practice in Solidity to provide transparency and facilitate easier debugging and auditing. For example, the setStakers function updates the staking amounts but does not emit an event to record this change.

Recommendation

Several state-changing functions in the contract do not log events for the changes they make. Event logging is a best practice in Solidity to provide transparency and facilitate easier debugging and auditing. For example, the setStakers function updates the staking amounts but does not emit an event to record this change.

```
event StakersUpdated(address staker, uint256 amount);

function setStakers() internal {
    uint256 length = stakers.length;
    for (uint256 i = 0; i < length; i++) {
        address staker = stakers[i];
        uint256 stakedAmount = getStakedAmountPerValidator(staker);
        stakingAmount[staker] = stakedAmount;
        emit StakersUpdated(staker, stakedAmount); // Log the update
    }
}
```

POC :

```
function setStakers() internal {
        uint256 length = stakers.length;
        for (uint256 i = 0; i < length; i++) {
            address staker = stakers[i];
            stakingAmount[staker] = getStakedAmountPerValidator(staker);
        }
```

```
    }
```

[G-01] Use assembly for 0 address checks

Severity: Gas

Location: Staking.sol, UpgradeStaking.sol

Description
Using assembly for Null Address checks can help save gas on function calls

POC

```
   function __Ownable_init_unchained(address initialOwner) internal
onlyInitializing {
        if (initialOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(initialOwner);
    }
```

Recommendation

```
function __Ownable_init_unchained(address initialOwner) internal
onlyInitializing {
    assembly {
        if iszero(initialOwner) {
            // Revert with the OwnableInvalidOwner error and address(0)
            mstore(0x00, 0x82b42900) // Function selector for
OwnableInvalidOwner(address)
            mstore(0x04, 0)          // Argument: address(0)
            revert(0x00, 0x24)       // Revert with the function selector
and argument
        }
    }
    _transferOwnership(initialOwner);
}
```

[G-02] `<=` is cheaper than `<`

Severity: Gas

Location : MainStaking.sol, Staking.sol , UpgradeStaking.sol

Description
Strict inequalities (<) are more expensive than non-strict ones (<=). This is due to some supplementary checks (ISZERO, 3 gas)

Recommendation
I suggest using <= instead of <

POC

```
  require(
            _validators.length > _minimumNumValidators,
            "Validators can't be less than the minimum required validator
num"
        );

        require(
            _addressToValidatorIndex[staker] < _validators.length,
            "index out of range"
        );
```

[G-03] For Loop can be optimized for reducing gas

Severity: Gas

Location : MainStaking.sol , Staking.sol , UpragdeStaking.sol

Description
For loop can be optimized to reduce the gas cost

POC :

```
    function validatorBLSPublicKeys() public view returns (bytes[] memory)
{
        bytes[] memory keys = new bytes[](_validators.length);
        for (uint256 i = 0; i < _validators.length; i++) {
            keys[i] = _addressToBLSPublicKey[_validators[i]];
        }

        return keys;
    }
```

Recommendation
Change the loop to something like this

```
function validatorBLSPublicKeys() public view returns (bytes[] memory) {
    uint256 length = _validators.length; // Cache the length of the
_validators array
    bytes[] memory keys = new bytes[](length);

    for (uint256 i = 0; i < length; ) {
        keys[i] = _addressToBLSPublicKey[_validators[i]];
        assembly {
            i := add(i, 1) // Use assembly to increment i
        }
    }

    return keys;
}
```

[G-04] Using `calldata` instead of `memory` for read-only arguments saves gas

Severity: Gas

Location : MainStaking.sol , Staking.sol, UpragdeStaking.sol

Description
Use calldata for arguments that are not changed while the function calls.

Recommendation
Change memory to calldata in the function parameters.

POC :

```
    function functionCall(address target, bytes memory data) internal
returns (bytes memory) {
        return functionCallWithValue(target, data, 0);
    }
```

[G-05] Functions guaranteed to revert when called by normal users can be marked `payable`

Severity: Gas

Location : Staking.sol , UpgradeStaking.sol

Description
If a function modifier such as **onlyOwner** is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Recommendation
Make the functions with onlyOwner modifier as payable to save some gas.

POC :

```
function transferOwnership(address newOwner) public virtual onlyOwner {
        if (newOwner == address(0)) { // @audit GO can use assembly for 0
address checks
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(newOwner);
    }
```

[I-01] Internal Function Naming Conventions

Severity: Informational

Location: MainStaking.sol , Staking.sol and UpgradeStaking.sol

Description
The current code does not follow the best practice for naming internal functions. Best practices recommend naming internal functions in camel case and prefixing them with an underscore (_).

Recommendation
Rename internal functions to follow camel case with an underscore prefix for improved readability and maintainability. For example, change functionName to _functionName.

POC :

```
   function setStakers() internal {
        uint256 length = stakers.length;
        for (uint256 i = 0; i < length; i++) {
            address staker = stakers[i];
            stakingAmount[staker] = getStakedAmountPerValidator(staker);
        }
           }
```

[I-02] Use of Informative Variable Names

Severity: Informational

Location: Staking.sol

Description
The variable name **$** used in the **_disableInitializers** function is not informative and does not convey its purpose or content. Best practices suggest using descriptive variable names to improve code readability and maintainability.

Recommendation
Replace the variable name $ with a more descriptive name that clearly indicates its purpose. For example, change **InitializableStorage storage $ = _getInitializableStorage();** to **InitializableStorage storage initializableStorage = _getInitializableStorage();**

POC :

```
    function _disableInitializers() internal virtual {
        // solhint-disable-next-line var-name-mixedcase
        InitializableStorage storage $ = _getInitializableStorage();

        if ($._initializing) {
            revert InvalidInitialization();
        }
        if ($._initialized != type(uint64).max) {
            $._initialized = type(uint64).max;
            emit Initialized(type(uint64).max);
        }
    }
```

[I-03] Modifier Side Effects

Severity: Informational

Location: UpgradeStaking.sol

Description

The variable name **$** used in the **_disableInitializers** function is not informative and does not convey its purpose or content. Best practices suggest using descriptive variable names to improve code readability and maintainability.

Recommendation
Replace the variable name $ with a more descriptive name that clearly indicates its purpose. For example, change **InitializableStorage storage $ = _getInitializableStorage();** to **InitializableStorage storage initializableStorage = _getInitializableStorage();**