Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Native BNB Token Staking

Veridise Inc.

February 17, 2023

► **Prepared For:**

ANKR
https://www.ankr.com/

► **Prepared By:**

Jon Stephens
Jacob Van Geffen
Yanju Chen

► **Contact Us:** contact@veridise.com

► **Version History:**

Feb 17, 2023 V1

© 2022 Veridise Inc. All Rights Reserved.

Contents

 © 2022 Veridise Inc.

1 Executive Summary

From Jan. 19 to Feb. 3, ANKR engaged Veridise to review the security of their Native BNB Token Staking protocol. The review covered the on-chain contracts that implement the protocol logic. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks on commit `41c5531` for the `stakefi-smart-contract` repository and `c004d02` for the `ankr-contracts` repository. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The ANKR Native BNB Token Staking protocol allows users to stake funds in return ANKR's liquid staking tokens `aBNBb` and `aBNBc`. These tokens earn yields for users over time as the token's ratio decreases. The staking pool allows validators to stake the pool funds in Binance's native staking pool to earn rewards on the staked funds. Since user funds may be staked at the time of withdraw, the staking pool also provides a waiting queue so that users can be paid in the order that withdraws were requested.

ANKR provided the source code for the Native BNB Token Staking protocol for review. In addition, they provided some documentation about the intended behavior of the protocol. Finally, they provided a set of tests based on the Truffle testing framework.

Summary of issues detected. The audit uncovered 17 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-ANS-VUL-001 charges users twice for staking their funds and V-ANS-VUL-002 would cause funds to be locked in a waiting queue. In addition, the auditors identified two moderate-severity issues, including the potential to underpay users when unstaking their funds (V-ANS-VUL-004). Finally, the auditors identified several other security concerns, including logic errors (V-ANS-VUL-006), a possible theft from the waiting queue (V-ANS-VUL-010) and a potential overflow on downcast (V-ANS-VUL-016).

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

© 2022 Veridise Inc.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Native BNB Token Staking	`41c5531, c004d02`	Solidity	BNB Smart Chain

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 19 - Feb. 3, 2022	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	1	1
High-Severity Issues	1	1
Medium-Severity Issues	2	1
Low-Severity Issues	7	6
Warning-Severity Issues	6	5
Informational-Severity Issues	0	0
TOTAL	17	14

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Locked Funds	3
Reentrancy	2
Theft	2
Dead Code	2
Denial of Service	1
Overflow	1
Data Validation	1
Gas Optimization	1

© 2022 Veridise Inc.

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the on-chain portion of the Native BNB Token Staking protocol defined in the following scope. In our audit, we sought to answer the following questions:

- Can users steal funds from the pool?
- Are users fairly rewarded with ANKR's liquid staking tokens upon staking?
- Will users be paid upon unstaking?
- If users are placed in the waiting queue, will they eventually be paid?
- Can one user deny payment to others in the waiting queue?
- Can users steal funds from the waiting queue?
- Will users be processed by the waiting queue only once?
- Are all external functions guarded by a reentrancy guard?
- Are all bearing tokens backed by certificate tokens?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The audit reviewed the on-chain behaviors of the Native BNB Token Staking protocol, including user staking, pool funds staking, user withdrawals and liquid staking tokens. The Veridise engineers first inspected the provided documentation to understand the high-level design of the protocol. They then inspected the provided test-cases to better understand the specific contract behavior. Finally, the auditors performed a multi-week audit of the code assisted both by static analyzers and automated testing. In terms of the audit, the following files were in-scope:

Repository: ankr-contracts

- contracts/earn/BearingToken.sol

© 2022 Veridise Inc.

- contracts/earn/CertificateToken.sol
- contracts/earn/EarnConfig.sol
- contracts/earn/LiquidTokenStakingPool.sol
- contracts/earn/extension/ERC20LiquidTokenStakingPool.sol
- contracts/earn/extension/ImmediateLiquidTokenStakingPool.sol
- contracts/earn/extension/ManualClaimLiquidTokenStakingPool.sol
- contracts/earn/extension/MixedLiquidTokenStakingPool.sol
- contracts/earn/extension/QueueLiquidTokenStakingPool.sol
- contracts/earn/extension/ReferralLiquidTokenStakingPool.sol

Repository: stakefi-smart-contract

- bnb-v2/contracts/native-staking/BNBStakingPool.sol
- bnb-v2/contracts/tokens/aBNBb.sol
- bnb-v2/contracts/tokens/aBNBc.sol
- bnb-v2/contracts/tokens/upgrades/aBNBb_R1.sol
- bnb-v2/contracts/tokens/upgrades/aBNBc_R1.sol

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ANS-VUL-001	Users Charged Twice by ERC20 Staking Pool	Critical	Fixed
V-ANS-VUL-002	Locked Funds in Staking Pool	High	Fixed
V-ANS-VUL-003	Potential Reentrancy	Medium	Fixed
V-ANS-VUL-004	Missing Requirements on Unstaking Amount	Medium	Open
V-ANS-VUL-005	Unguarded External Function	Low	Fixed
V-ANS-VUL-006	Incorrect Gap Tracking	Low	Fixed
V-ANS-VUL-007	Min Delegation Requirement Bypassed	Low	Intended Behavior
V-ANS-VUL-008	Inconsistent Values Provided to Function	Low	Fixed
V-ANS-VUL-009	Potential Unlock DoS	Low	Fixed
V-ANS-VUL-010	Potential Theft from Queue	Low	Fixed
V-ANS-VUL-011	Potential Underpay for Shares	Low	Open
V-ANS-VUL-012	Wasting Gas via External Call	Warning	Fixed
V-ANS-VUL-013	Double Converting between Shares and Bonds	Warning	Intended Behavior
V-ANS-VUL-014	Unused State	Warning	Intended Behavior
V-ANS-VUL-015	Duplicated Code	Warning	Invalid
V-ANS-VUL-016	Potential Overflow on Downcast	Warning	Fixed
V-ANS-VUL-017	Potentially Unsafe Conditional	Warning	Open

© 2022 Veridise Inc.

4.1 Detailed Description of Bugs

4.1.1 V-ANS-VUL-001: Users Charged Twice by ERC20 Staking Pool

Severity	Critical	Commit	c298180
Type	Locked Funds	Status	Fixed
Files	ERC20LiquidTokenStakingPool		
Functions	_beforeStake		

The ERC20LiquidTokenStakingPool contract extends the LiquidTokenStakingPool so that users can pay with ERC20 tokens rather than native tokens. To do so, the developers override the `_beforeStake` utility function to accept ERC20 payments from users as shown below.

```
    function _beforeStake(
        address account,
        uint256 amount,
        uint256 /* shares */
    ) internal virtual override {
        require(
            _stakeableToken.transferFrom(account, address(this), amount),
            "failed to receive stakeableToken"
        );
    }
```

Snippet 4.1: The override of `_beforeStake` to accept ERC20 payments

The contract, however, does not override the staking functionality that it inherits from the LiquidTokenStakingPool contract. As shown below, however, these functions require payment in native tokens.

```
    function stakeBonds()
        external
        payable
        override
        nonReentrant
        bondStakingUnpaused
    {
        _stakeBonds(msg.sender, msg.value);
    }

    function stakeCerts() external payable override nonReentrant {
        _stakeCerts(msg.sender, msg.value);
    }
```

Snippet 4.2: The functions used to stake funds in the Liquid Staking Pool

Impact Given that the staking functions require payment in native token, and that this contract transfers ERC20 tokens from the user without crediting them additional funds, the contract

essentially charges users twice. In addition, if `_unsafeTransfer` is used to pay the user back on an unstake (as done in other extensions), these users will only be refunded the ERC20 tokens.

Recommendation Add a separate API specifically for ERC20 tokens so that users will not need to pay ERC20 tokens and native tokens. In addition, if the intention is to only accept ERC20 tokens move the staking functionality from LiquidTokenStakingPool to one of the extensions.

© 2022 Veridise Inc.

4.1.2 V-ANS-VUL-002: Locked Funds in Staking Pool

Severity	High	**Commit**	c298180
Type	Locked Funds	**Status**	Fixed
Files	QueueLiquidTokenStakingPool.sol		
Functions	_distributePendingRewards		

If the LiquidStakingPool does not have sufficient funds to pay all unstaking users, those users will be placed in a queue to wait for more funds to become available. Refunds are then distributed by popping users off the queue and sending the appropriate refund. If the funds are not successfully sent, the user is moved out of the queue and their funds must be claimed manually by the user. During the process in which funds are marked as manually claimed, however, a user's `_pendingClaimerUnstakes` state is not updated to reflect that they no longer have funds in the queue to claim as shown below.

```
1   function _distributePendingRewards() internal {
2       ...
3
4       while (
5           i < _pendingClaimers.length &&
6           poolBalance > 0 &&
7           gasleft() > _DISTRIBUTE_GAS_LIMIT
8       ) {
9           address claimer = _pendingClaimers[i];
10      ...
11      uint256 toDistribute = _pendingClaimerUnstakes[claimer];
12          ...
13          bool success = _unsafeTransfer(claimer, toDistribute, true);
14          /* when we delete items from array we generate new gap, lets remember how
     many gaps we did to skip them in next claim */
15          if (!success) {
16              toDistribute = _pendingClaimerUnstakes[claimer];
17              // already accounted in var _stashedForManualClaims
18              _pendingTotalUnstakes -= toDistribute;
19              _setForManualClaim(claimer, toDistribute);
20              gaps++;
21              i++;
22              continue;
23          }
24          ...
25      }
26     ...
27      emit RewardsDistributed(claimers, amounts);
28  }
```

Snippet 4.3: Snippet of the refund distribution function that handles situations where funds are not successfully sent

Since the `_pendingClaimerUnstakes` state is non-zero if a user must ever perform a manual claim, future attempts to add a user will update the user's refund state but will not add the user to the queue as shown below.

```
    function _addIntoQueue(address claimer, uint256 amount) internal {
        require(
            claimer != address(0),
            "LiquidTokenStakingPool: claimer is zero address"
        );
        if (_pendingClaimerUnstakes[claimer] == 0) {
            _pendingClaimers.push(claimer);
        }
        _pendingTotalUnstakes += amount;
        _pendingClaimerUnstakes[claimer] += amount;
        emit PendingUnstake(claimer, claimer, amount);
    }
```

Snippet 4.4: The function that adds users to the refund queue.

Impact Since users who have required manual claims in the past will not be added to the refund queue array, their funds will be locked within the contract.

Recommendation When a user must manually claim funds, ensure `_pendingClaimerUnstakes` is properly updated.

© 2022 Veridise Inc.

4.1.3 V-ANS-VUL-003: Potential Reentrancy

Severity	Medium	**Commit**	c298180
Type	Reentrancy	**Status**	Fixed
Files	ManualClaimLiquidTokenStakingPool.sol		
Functions	claimManually		

It is common practice to guard against reentrancy vulnerabilities by modifying state before making potentially dangerous calls to untrusted sources. The `claimManually` function, however, makes an external call to some receiver and then adjust the contract state as shown below.

```
function claimManually(uint256 id) external nonReentrant {
        ...

        bool result = _unsafeTransfer(receiverAddress, amount, false);
        require(
            result,
            "LiquidTokenStakingPool: failed to send rewards to receiverAddress"
        );
        _stashedForManualClaims -= amount;
        _manualClaims[receiverAddress] = 0;
        _markedForManualClaim[id] = address(0);
        emit RewardsClaimed(receiverAddress, msg.sender, amount);
    }
```

Snippet 4.5: The `claimManually` function makes an unsafe call before modifying state.

Impact While a majority of the external functions are guarded with a reentrancy guard, a few functions are not guarded. In addition, several view functions and modifiers could use stale values, such as: `getFreeBalance`, `getStashedForManualClaims`, `getForManualClaimOf` and `getForManualClaimOf`.

Recommendation Perform the state modifications before the given external call.

4.1.4 V-ANS-VUL-004: Missing Requirements on Unstaking Amount

Severity	Medium	**Commit**	c298180
Type	Locked Funds	**Status**	Open
Files	LiquidTokenStakingPool.sol		
Functions	_unstakeCertsFor		

In `LiquidTokenStakingPool.sol`, the functions `_unstakeBondsFor` and `_unstakeCertsFor` lack lower bounds on the amount to be unstaked. This is a problem because it allows users to first stake some amount of shares, then unstake an amount just under the value of a share without losing any staked shares. To understand, consider the following code from `_unstakeBondsFor`:

```
function _unstakeBondsFor(address receiverAddress, uint256 amount)
        internal
    {
        address ownerAddress = msg.sender;
        uint256 shares = _bearingToken.bondsToShares(amount);
        ...
        _certificateToken.burn(address(_bearingToken), shares);
        _bearingToken.burn(ownerAddress, shares);
        _afterUnstake(ownerAddress, receiverAddress, amount);
        ...
    }
```

Since `bondsToShares` rounds, it's possible that the amount returned to the user when unstaking is incorrect.

Impact Without the restriction on `amount`, it's possible that a user may unstake almost an entire share's worth of ether for free, or accidentally lose a share with almost no ether returned to them.

Recommendation In both the `_unstakeBondsFor` and `_unstakeCertsFor` functions, include the following requirement on the amount to unstake:

```
require(amount >= getMinStake() && amount % getMinStake() == 0)
```

Alternatively, convert the number of shares to burn back to bonds to determine the actual value of the burnt shares.

© 2022 Veridise Inc.

4.1.5 V-ANS-VUL-005: Unguarded External Function

Severity	Low	**Commit**	c298180
Type	Reentrancy	**Status**	Fixed
Files	ReferralLiquidTokenStakingPool.sol		
Functions	stakeBondsWithCode, stakeCertsWithCode		

A majority of the liquid staking pools in the earn repository use a ReentrancyGuard to prevent potential reentrancy vulnerabilities. The ReferralLiquidTokenStakingPool contract, however, does not use such a reentrancy guard and is inherited by MixedLiquidTokenStakingPool which does have suitable reentrancy protections. Since there are several locations where control is transferred to the user, we would recommend protecting these functions with a ReentrancyGuard.

```
    function stakeBondsWithCode(bytes32 code) external payable override {
        _stakeBonds(msg.sender, msg.value);
        emit ReferralCode(code);
    }

    function stakeCertsWithCode(bytes32 code) external payable override {
        _stakeCerts(msg.sender, msg.value);
        emit ReferralCode(code);
    }
```

Snippet 4.6: The functions within ReferralLiquidTokenStakingPool that are not protected by a reentrancy guard.

Impact Such unprotected external functions could leave the protocol open to reentrancy attacks.

Recommendation Guard all external functions with a ReentrancyGuard.

4.1.6 V-ANS-VUL-006: Incorrect Gap Tracking

Severity	Low	**Commit**	c298180
Type	Logic Error	**Status**	Fixed
Files	QueueLiquidTokenStakingPool.sol		
Functions	_distributePendingRewards		

The `gap` variable is not correctly tracked in all potential branches in the following code snippet of `_distributePendingRewards` function:

```
function _distributePendingRewards() internal {
    ...
    uint256 gaps = 0;
    uint256 i = _pendingGap;

    while (
        i < _pendingClaimers.length &&
        poolBalance > 0 &&
        gasleft() > _DISTRIBUTE_GAS_LIMIT
    ) {
        address claimer = _pendingClaimers[i];
        // if claimer for manual distribute skip him and increase 'i'
        if (this.isMarkedForManualClaim(claimer)) {
            i++;
            continue;
        }
        uint256 toDistribute = _pendingClaimerUnstakes[claimer];
        if (claimer == address(0) || toDistribute == 0) {
            i++;
            gaps++;
            continue;
        }

        ...
        if (!success) {
            toDistribute = _pendingClaimerUnstakes[claimer];
            // already accounted in var _stashedForManualClaims
            _pendingTotalUnstakes -= toDistribute;
            _setForManualClaim(claimer, toDistribute);
            gaps++;
            i++;
            continue;
        }
        ...
        i++;
        gaps++;
    }
    _pendingGap += gaps;
    ...
}
```

Snippet 4.7: Snippet of the `_distributePendingRewards` function where `gap` is updated

© 2022 Veridise Inc.

In particular, when a claimer is skipped due to being marked for manual claim, the corresponding `gap` is not increased which causes miscalculation of the gaps in the array `_pendingClaimers`.

Impact For an array of `_pendingClaimers` that has at least one claimer that is marked for manual claim and skipped in the middle of the while iteration, the next time the function `_distributePendingRewards` is invoked, the very last (few) claimer processed in the previous call to this function will be processed again.

Recommendation The `gap` variable is not needed since it synchronizes with `i`; use `i` instead to track the gaps on the array.

4.1.7 V-ANS-VUL-007: Min Delegation Requirement Bypassed

Severity	Low	**Commit**	c298180
Type	Logic Error	**Status**	Intended Behavior
Files	BNBStakingPool.sol		
Functions	undelegate		

When using the `BNBStakingPool`, there is a requirement on the minimum amount allowed to be delegated. However, this requirement can be circumvented by performing the following operations:

1. `delegate` the amount `minDelegation + e`
2. `undelegate` the amount `minDelegation`

Since `undelegate` does not check that the resulting amount delegated is at least `minDelegation` (only that the amount undelegated is at least `minDelegation`), this may result in a state where the amount delegated is `e`, which may be less than `minDelegation`

```
function undelegate(address validator, uint256 amount) {
    ...
    uint256 minDelegate = _stakingContract.getMinDelegation();
    require(
        amount >= minDelegate,
        "BNBStakingPool: amount less than minDelegate amount"
    );
    ...
}
```

Snippet 4.8: The undelegate function that does not ensure the minimum amount is delegated.

Impact Allowing stakers to delegate less than the `minDelegation` amount may violate properties of the consensus protocol.

Recommendation Add additional constraints to `undelegate` that ensures either zero or at least `minDelegation` funds remain.

Developer Response The `minDelegate` variable is not used to indicate the minimum value that should be delegated but rather the smallest amount that can be delegated or undelegated at once.

© 2022 Veridise Inc.

4.1.8 V-ANS-VUL-008: Inconsistent Values Passed to Token Transfer Utility Functions

Severity	Low	Commit	c298180
Type	Logic Error	Status	Fixed
Files	BearingToken.sol		
Functions	_transfer, _mint, _burn		

Similar to OpenZeppelin's ERC20 contract, the BearingToken contract allows inherited contracts to add behaviors to a token transfer using two functions: `_beforeTokenTransfer` and `_afterTokenTransfer`. Unlike an ERC20 contract, however, the `BearingToken` tracks two values (bonds and shares) using a single token. When calling `_beforeTokenTransfer` and `_afterTokenTransfer` in some cases `bonds` are passed to the function while in others `shares` are passed to the function as shown below.

```
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual override {
        uint256 shares = bondsToShares(amount);
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        _beforeTokenTransfer(from, to, amount);
        ...
        _afterTokenTransfer(from, to, amount);
    }
```

Snippet 4.9: Definition of the `_transfer` function that calls the utility functions with bonds.

```
    function _mint(address account, uint256 shares) internal virtual override {
        uint256 amount = sharesToBonds(shares);
        require(account != address(0), "ERC20: mint to the zero address");
        _beforeTokenTransfer(address(0), account, shares);
        ...
        _afterTokenTransfer(address(0), account, shares);
    }

    function _burn(address account, uint256 shares) internal virtual override {
        uint256 amount = sharesToBonds(shares);
        require(account != address(0), "ERC20: burn from the zero address");
        _beforeTokenTransfer(account, address(0), shares);
        ...
        _afterTokenTransfer(account, address(0), shares);
    }
```

Snippet 4.10: Definition of the `_mint` and `_burn` functions that calls the utility functions with shares.

Impact Any functionality that is added to transfer using these functions in the future is extremely likely to introduce errors as the function has no way of determining if shares or bonds are provided.

Recommendation Either only pass bonds to these functions or only pass shares.

© 2022 Veridise Inc.

4.1.9 V-ANS-VUL-009: Potential Unlock DoS

Severity	Low	**Commit**	c298180
Type	Denial of Service	**Status**	Fixed
Files	BearingToken.sol		
Functions	_mint		

The BearingToken allows users to `lock` and `unlock` CertificateTokens within the contract. When these tokens are `locked`, they are transferred to the BearingToken contract and new BearingTokens are minted for the user. They can then `unlocked` which burns the BearingTokens and transfers CertificateTokens back to the user. Since the CertificateTokens are owned by the BearingToken and transferred to the user on an `unlock`, it is possible that if BearingTokens are unbacked by CertificateTokens, every holder of BearingTokens could not `unlock`. Since there is a separate mint function that does not check if new BearingTokens are backed by CertificateTokens it is possible that this can happen in the future.

```
1   function _mint(address account, uint256 shares) internal virtual override {
2        uint256 amount = sharesToBonds(shares);
3        require(account != address(0), "ERC20: mint to the zero address");
4        _beforeTokenTransfer(address(0), account, shares);
5        _totalSupply += shares;
6        _shares[account] += shares;
7        emit Transfer(address(0), account, amount);
8        _afterTokenTransfer(address(0), account, shares);
9    }
```

Snippet 4.11: The `_mint` function that does not ensure minted tokens are backed.

Impact If a user is able to receive unbacked `BearingTokens`, they could intentionally `unlock` them as a Denial of Service attack against the unlocking functionality of the `BearingToken`

Recommendation Currently the `mint` API is used correctly in that Certificate Tokens are minted to the BearingToken when BearingTokens are minted for a user. However, we would recommend that the developers add a check in the `mint` function to ensure that the `BearingToken` is backed by a sufficient number of `CertificateTokens` to prevent any future mistakes.

4.1.10 V-ANS-VUL-010: Potential Theft from Queue

Severity	Low	**Commit**	c298180
Type	Theft	**Status**	Fixed
Files	QueueLiquidTokenStakingPool		
Functions	_addIntoQueue		

If the LiquidStakingPool does not have sufficient funds to pay all unstaking users, those users will be placed in a queue to wait for more funds to become available. Currently the queue maintains an array of users to be repaid and mapping of amounts to repay individual users. When added to the queue, a check is performed to see if a user has a pending refund. If they don't, they will be added to the end of the array. In addition their pending refund amount will be increased. This implementation has an implicit assumption that any user in the queue also has a non-zero refund amount. As shown below though, there is no check to prevent a user from being added to the queue with a refund of 0.

```
1    function _addIntoQueue(address claimer, uint256 amount) internal {
2        require(
3            claimer != address(0),
4            "LiquidTokenStakingPool: claimer is zero address"
5        );
6        if (_pendingClaimerUnstakes[claimer] == 0) {
7            _pendingClaimers.push(claimer);
8        }
9        _pendingTotalUnstakes += amount;
10       _pendingClaimerUnstakes[claimer] += amount;
11       emit PendingUnstake(claimer, claimer, amount);
12   }
```

Snippet 4.12: The function that adds users to the refund queue.

Impact Since `addToQueue` will allow a user to be added with a refund amount of 0, an attacker could be added multiple times with a zero refund amount so that they appear in the queue multiple times. They could then be added one more time with a non-zero refund to potentially receive multiple non-zero refunds from the pool.

Recommendation While the current implementation won't allow a user to be added to the queue with a zero refund due to a conditional in another function,

 © 2022 Veridise Inc.

4.1.11 V-ANS-VUL-011: Potential Underpay for Shares

Severity	Low	**Commit**	c298180
Type	Theft	**Status**	Open
Files	LiquidTokenStakingPool.sol		
Functions	_stakeBonds		

When staking some amount, it's possible for users to underpay for shares due to the nature of the `bondsToShares` conversion. The following code from `LiquidTokenStakingPool` calculates the number of shares to stake for a given amount of bonds:

```
function _stakeBonds(address staker, uint256 amount) internal {
    uint256 shares = _bearingToken.bondsToShares(amount);
    _stake(staker, amount, shares, true);
    _afterStake(staker, amount, shares);
}
```

However, since `bondsToShares` rounds up, it is possible to stake a small amount of bonds and still acquire a share. This is possible whenever `getMinStake() % sharesToBonds(1) != 0`

Impact Users may acquire shares for cheaper than expected, resulting in a loss of funds.

Recommendation Add a requirement that the amount staked should be a positive multiple of `sharesToBonds(1)`. This can be implemented in a variety of ways, but we recommend doing so by maintaining the invariant `getMinStake() % sharesToBonds(1) == 0 && getMinStake() > 0`. This could be done by:

1. Adding the following requirement to the `setMinimumStake` function: `require(newValue % sharesToBonds(1) == 0 && newValue > 0)`
2. Adding the following call to the `setInternetBondRatioFeed` function: `_liquidStakingPool.setMinimumStake(sharesToBonds(1))`

Alternatively, `getMinStake` could return `sharesToBonds(1) * M` where `M` is number set by `setMinStake` to indicate how many multiples someone must stake.

Since `_stake` already has the requirement `amount%getMinStake()==0 && amount>=getMinStake()`, this will ensure the invariant that any number of shares `N` staked has been paid for by the appropriate number of bonds from the user.

4.1.12 V-ANS-VUL-012: Wasting Gas via External Call

Severity	Warning	**Commit**	c298180
Type	Gas Optimization	**Status**	Fixed
Files	Multiple		
Functions	Multiple		

In several contracts, external functions are being invoked using the form `this.fn(...)` as shown below. Such invocations cause the function to be invoked via an external call rather than a function call.

```
    function getFreeBalance() external view virtual override returns (uint256) {
        return
            address(this).balance < this.getStashedForManualClaims()
                ? 0
                : address(this).balance - this.getStashedForManualClaims();
    }
```

Snippet 4.13: Example code that invokes a function using `this`

Impact Invoking external calls costs more gas than invoking a call to a function inside the contract causing gas to be wasted.

Recommendation Make the required view functions `public` rather than external and then call them without `this`.

© 2022 Veridise Inc.

4.1.13 V-ANS-VUL-013: Double Converting between Shares and Bonds

Severity	Warning	**Commit**	c298180
Type	Logic Error	**Status**	Intended Behavior
Files	aBNBb.sol, BearingToken.sol		
Functions	burnAndSetPending		

Before calling `_burn` from `BearingToken.sol`, functions `burnAndSetPending` and `burnAndSetPendingFor` in `aBNBb.sol` convert from bonds to shares in order to compute the number of shares to burn. However, `_burn` internally converts from shares to bonds. This double conversion can lead to rounding errors, and thus the incorrect number of shares/bonds may be burned.

```
    function burnAndSetPending(address account, uint256 amount)
        external
        override
        whenNotPaused
        onlyLiquidStakingPool
    {
        _pendingBurn[account] += amount;
        pendingBurnsTotal += amount;
        uint256 sharesToBurn = bondsToShares(amount);
        _burn(account, sharesToBurn);
        emit Transfer(account, address(0), amount);
    }
```

Snippet 4.14: `burnAndSetPending` converts from bonds to shares

```
function _burn(address account, uint256 shares) internal virtual override {
    uint256 amount = sharesToBonds(shares);
    ...
    emit Transfer(account, address(0), amount);
    ...
}
```

Snippet 4.15: `_burn` converts from shares to bonds

Impact Double converting in this case can result in unwanted truncation, thus burning fewer tokens than intended.

Recommendation Within `BearingToken`, include an additional `_burnBonds` function that takes bonds rather than shares. This way, bonds can be burned directly without the need to convert from shares to bonds.

4.1.14 V-ANS-VUL-014: Unused State

Severity	Warning	**Commit**	c298180
Type	Dead Code	**Status**	Intended Behavior
Files	BearingToken.sol		
Functions	N/A		

The bearing token declares several variables that are never used, including:

- `_lockedShares`
- `_balances` via ERC20Upgradable
- `_totalSupply` via ERC20Upgradable
- `_owner` via OwnableUpgradable

Impact Unused variables such as `_lockedShares` can lead to bugs in the future as one may expect they hold specific values when they do not. Other unused variables, such as `_balances` can cause issues if an inherited function that uses these variables is called.

Recommendation Remove unused variables in the contract such as `_lockedShares`. In addition, since very few behaviors are inherited from `ERC20Upgradable` (we believe it is just allowance tracking) we would recommend copying the remaining behaviors from the contract and then extend `IERC20Upgradable`.

Developer Response Since these tokens are already deployed to the blockchain as upgradable contracts, these variables cannot be removed.

© 2022 Veridise Inc.

4.1.15 V-ANS-VUL-015: Duplicated Code

Severity	Warning	**Commit**	c298180
Type	Dead Code	**Status**	Invalid
Files	MixedLiquidTokenStakingPool.sol		
Functions	_beforeUnstake		

The MixedLiquidTokenStakingPool overrides the `_beforeUnstake` function as follows:

```
    function _beforeUnstake(
        address, /* ownerAddress */
        address receiverAddress,
        uint256 /* amount */
    )
        internal
        virtual
        override(LiquidTokenStakingPool, ManualClaimLiquidTokenStakingPool)
    {
        require(
            !this.isMarkedForManualClaim(receiverAddress),
            "LiquidTokenStakingPool: receiver is marked for manual claim"
        );
    }
```

Snippet 4.16: The `_beforeUnstake` function declared in the MixedLiquidTokenStakingPool contract

The function it overrides, however is the following:

```
    function _beforeUnstake(
        address, /* ownerAddress */
        address receiverAddress,
        uint256 /* amount */
    ) internal virtual override(LiquidTokenStakingPool) {
        require(
            !this.isMarkedForManualClaim(receiverAddress),
            "LiquidTokenStakingPool: receiver is marked for manual claim"
        );
    }
```

Snippet 4.17: The `_beforeUnstake` function declared in the ManualClaimLiquidTokenStakingPool contract

Note that these functions are the same so the override in MixedLiquidTokenStakingPool is unnecessary.

Developer Response This was done to prevent a compiler warning.

4.1.16 V-ANS-VUL-016: Potential Overflow on Downcast

Severity	Warning	**Commit**	c298180
Type	Overflow	**Status**	Fixed
Files	LiquidTokenStakingPool.sol		
Functions	setMinimumStake		

The `setMinimumStake` function allows the pool governance to change the minimum value that a user is allowed to stake at once. To save space, the stake is expressed as a multiple of `gwei`, allowing the value to be stored using a smaller integer. If the new value is greater than or equal to `18446744073709551616000000000000` the downcast to a `uint64` could overflow.

```
1    function setMinimumStake(uint256 newValue)
2        external
3        override
4        onlyGovernance
5    {
6        require(
7            newValue % 1 gwei == 0,
8            "LiquidTokenStakingPool: value should be multiplied of gwei"
9        );
10       uint256 prevValue = getMinStake();
11       minimumStake = uint64(newValue / 1 gwei);
12       emit MinimumStakeChanged(prevValue, newValue);
13   }
```

Snippet 4.18: Location in setMinimumStake where a downcast is performed

Impact Such an admin error could cause the minimum stake to be lower than intended.

Recommendation Check that `minimumStake * 1 gwei == newValue`

© 2022 Veridise Inc.

4.1.17 V-ANS-VUL-017: Potentially Unsafe Conditional

Severity	Warning	**Commit**	c298180
Type	Data Validation	**Status**	Open
Files	ERC20LiquidTokenStakingPool.sol		
Functions	_unsafeTransfer		

The `_unsafeTransfer` function in the ERC20LiquidTokenStakingPool contract emulates the behavior of the function of the same name in the LiquidTokenStakingPool contract. To do so, the function transfers ERC20 tokens to the indicated user and returns whether or not the transfer was successful by checking both the return value and that the transfer didn't revert. To support tokens that don't properly implement the ERC20 specification, the contract allows no value to be returned.

```
1    function _unsafeTransfer(
2        address receiverAddress,
3        uint256 amount,
4        bool /* limit  */
5    ) internal virtual override returns (bool) {
6        require(
7            address(_stakeableToken) != address(0),
8            "LiquidTokenStakingPool: token is not set"
9        );
10       // bytes4(keccak256(bytes('transfer(address,uint256)')));
11       (bool success, bytes memory data) = address(_stakeableToken).call(
12           abi.encodeWithSelector(0xa9059cbb, receiverAddress, amount)
13       );
14       return success && (data.length == 0 || abi.decode(data, (bool)));
15   }
```

Snippet 4.19: Definition of the `_unsafeTransfer` function

Impact When performing an external call, the fallback function will be executed if a function could not be found with a matching selector. In such a case, the function could execute successfully and likely wouldn't return a value.

Recommendation We would recommend excluding the `data.length == 0` condition for tokens that implement the ERC20 specification correctly.