verichains

SECURITY AUDIT OF

STAKING V2 CONTRACTS AND MIGRATION SCRIPTS

Public Report

Dec 13, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Dec 13, 2023. We would like to thank the BreederDAO for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Staking V2 Contracts and Migration Scripts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY 5

1.1. About Staking V2 Contracts and Migration Scripts 5

1.2. Audit scope 5

1.3. Audit methodology 7

1.4. Disclaimer 8

1.5. Acceptance Minute 8

2. AUDIT RESULT 9

2.1. Overview 9

2.1.1. BreederTimeLockPool 9

2.1.2. BreederTimeLockNonTransferablePool 9

2.1.3. BreederMigrationPool 9

2.1.4. BreederWhitelistPool 9

2.1.5. LiquidityMiningManager 9

2.2. Findings 9

2.2.1. Bonus multiplier abuse HIGH 10

2.2.2. `breedWhitelistTransfer` does not burn `lock` and `bonus` shares HIGH 12

2.2.3. Unable to withdraw tokens if users transfer their share INFORMATIVE 14

3. VERSION HISTORY 16

1. MANAGEMENT SUMMARY

1.1. About Staking V2 Contracts and Migration Scripts

The Staking V2 contracts facilitates a staking mechanism, allowing users to lock up their tokens for a specified period and earn rewards.

The migration scripts helps to migrate all stakes and rewards from Staking V1 pools to the V2 pools.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of Staking V2 Contracts and Migration Scripts. It was conducted on commit `0d192d3d310f42c2cbc3846d97f675424fd2d795` from git repository link: *https://github.com/breederdao/staking-v2*.

The Staking V2 Contracts and Migration Scripts use `LayerZero` to send and receive cross chain messages which is out of scope in this audit.

`IStakingBonusHandler` contract is also out of scope in this audit.

The latest version of the following files were made available in the course of the review:

Staking V2 Contracts:

SHA256 Sum	File
97cfcdfcd7998e470174e54bb87668c48bc1a5ddb36ad25a4c142525b3907918	stakingv2/BreederWhitelistPool.sol
bf8f92450f07964f85f9a17aa754941deca9721e82ec32e9e70ffde95cc8a09c	stakingv2/BreederMigrationPool.sol
cd26c6f2beb357f87406c70faa2c813311c299707d7bba60279423ab4558b50c	stakingv2/BreederLiquidityMiningManager.sol
3ae285e663d347cf8373f6fa2b00e11a40ceb54a2262dc55138ab6195dea9338	stakingv2/BreederTimeLockNonTransferablePool.sol
8ff57819f21364d1ca2fed92162aae4104ba14e29104be5f64094e36b90261fe	stakingv2/BreederStakingView.sol
04181a6e5212e5e01789170121b84b2dafd01ea1f62519a01c085a2e5f4648bd	stakingv2/base/BreederTimeLockPool.sol
15a64262b2aefc27093926b4c6522287cd0b9a99524dafff01790e23f2235b41	stakingv2/base/AbstractRewards.sol

02a52bdbcf33e8b998b7be27cfce0c1a8419b4e8880510b495dd28563855c84c	stakingv2/base/TokenSaver.sol
b63e3e0f3711161f26198221150ccaf2da038237c261d957169bad269a083e94	stakingv2/base/BasePool.sol

Migration Scripts:

SHA256 Sum	File
18adef06658faa49ae31bb31e3a25f4c20b851b7d94fff1b581669a0f0a4d8fb	scripts/batchDepositVerification.js
a228b3435ba8d042256339c657302bbdf7e29a858551ed60b00fac4b432373de	scripts/batchRewardVerification.js
517e83045631333abb8b03a3fb67ab5c107a2b6740876b224424ac36b7ed63d4	scripts/deployment/0_deployMockToken.js
131c759dc060994899a38091c7294b6422e559a0da653d283f9b203b5ecdd7ea	scripts/deployment/1_deployWhitelistPool.js
ad28a4854f615d1abae8a99bcbd402c3ff6737859ae988cfec9a673f42d0c4e7	scripts/deployment/2_deployStaking.js
cc14ddbae302c5775bcb42681b2d96e7b3a0eb842d01bc88f69a5cda019e19e5	scripts/deployment/3_deployLMM.js
903403b63f6970f1164aa8a7a96a550e6db370e3af25c98ca65fc0c25f219d41	scripts/deployment/4_deployStakingView.js
bbad1ae01183612956687a7b202bf9094be3dc9d611cc07e89a8882d1b93e997	scripts/migrate_all.js
fbe79066557cf464486020fe183476e91a3168788415fdd5f531508742871d95	scripts/migration/batchDepositMigration.js
3b5a77010b1acae785fa190f2e1746e5507d30845a088e5fb0f6b5df1869cee4	scripts/migration/batchRewardMigration.js
bbd72aaf3a0147ed073e2f4b680efe09c56d7fe4253e93c4ac3298be1c7eb1fb	scripts/runBatchDepositMigration.js
5ee4e050e3d53604265ff6415ed970717a78db016f5b59f9675ffacc6bb9ebcc	scripts/runBatchDepositSnapshot.js
6ed880008966bb5a6a4993ce99ec2b5260cc74670a8a5d099b1b6cd6b4bfc8a5	scripts/runBatchRewardMigration.js

13cc8617d2a737e2f3561be6d1d8636430c281160040fa32852d4c9dcbd5a1c9	scripts/runBatchRewardSnapshot.js
53b51ef903c80411756ceb419c3e9c475652e2e148e6054a702dd42970c5782c	scripts/snapshot/batchDepositSnapshot.js
9ea269f78e269da0f406cd53351be49f87f0941e9b8c7214d79a4fb34443a859	scripts/snapshot/batchRewardSnapshot.js
e3923e6067510f4c784da625d9006f5221fda1f19508495331a2af74dbac7868	scripts/test.js
209642e23ad15f933d3f4cf17ce3333ce2ee56f14048f0619e930ae5dcee7eee	scripts/utils/date.js
ed0665157e542f22ce91d26189215e8b22142d16bafaabe6aa5f4e514cfbb252	scripts/utils/sleep.js

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

BreederDAO acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. BreederDAO understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, BreederDAO agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the BreederDAO will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the BreederDAO, the final report will be considered fully accepted by the BreederDAO without the signature.

2. AUDIT RESULT

2.1. Overview

The Staking V2 Contracts were written in `Solidity` language, with the required version to be `0.8.8`. The migration scripts were written in `Javascript`, with the help of `Hardhat`.

2.1.1. BreederTimeLockPool

This contract is a base to facilitate a time-locked functionality for staking tokens and rewards distribution. It also provides cross-chain reward claim using `LayerZero` cross-chain message. The contract extends `TokenSaver` for securely saving tokens with `AccessControlEnumerable` features. By default, the contract deployer has `DEFAULT_ADMIN_ROLE`. The admin can set `TOKEN_SAVER_ROLE` to any one and they can withdraw any tokens stored in this pool including stake tokens.

2.1.2. BreederTimeLockNonTransferablePool

This is an implementation of the `BreederTimeLockPool` with transfer disabled.

2.1.3. BreederMigrationPool

This is an implementation of the `BreederTimeLockPool` with transfer disabled that allows for migration deposit from Staking v1 pools to Staking v2 pools.

2.1.4. BreederWhitelistPool

The `BreederWhitelistPool`/`Escrow` pool is maintained to manage the flow of assets and rewards in the staking ecosystem. It holds reward until certain a lock-up period/escrow duration are met. The escrow pool is created to encourage participants to stay committed to the staking program for a more extended period, fostering stability and security within the ecosystem.

It allows for whitelisted addresses to transfer `Breed` to themselves on behalf of a user. The `batchDeposit` function is created for migrating rewards from Staking v1 contract to Escrow v2 contract. It deposits user's unclaimed reward directly to the v2 escrow pool

2.1.5. LiquidityMiningManager

The `LiquidityMiningManager` contract facilitates distribution of funds to the pools. It rewards users for staking tokens and providing liquidity to designated pools.

2.2. Findings

During the audit process, the audit team found some vulnerability issues in the given version of the Staking V2 Contracts and Migration Scripts.

BreederDAO fixed all the issues according to Verichains's draft report in commit `3bdc9f41d151b06f14fdd1978db56dbf1c10cc4c`.

#	Issue	Severity	Status
1	Bonus multiplier abuse	HIGH	Fixed
2	`breedWhitelistTransfer` does not burn `lock` and `bonus` shares	HIGH	Fixed
3	Unable to withdraw tokens if users transfer their share	INFORMATIVE	Acknowledged

Table 2. Findings

2.2.1. Bonus multiplier abuse HIGH

Affected files:

- BreederTimeLockPool.sol

When users deposit tokens, the pool mints an amount of shares plus a bonus from locking duration and `stakingBonusHandler` but when they withdraw their tokens, only the base and locking bonus shares are burnt, the bonus from `stakingBonusHandler` are left over. Users can abuse this bonus by repeating the process of "lock deposit for 10 minutes (`MIN_LOCK_DURATION`) then withdraw" to mint unlimited amount of bonus shares.

```
function deposit(
    uint256 _amount,
    uint256 _duration,
    address _receiver
) public virtual override {
    _makeDeposit(_amount, _duration, _receiver);
}

function _makeDeposit(
    uint256 _amount,
    uint256 _duration,
    address _receiver
) internal {
    require(_amount > 0, "BreederTimeLockPool.deposit: cannot deposit 0");
    // Don't allow locking > maxLockDuration
    uint256 duration = _duration.min(maxLockDuration);
    // Enforce min lockup duration to prevent flash loan or MEV transaction ordering
    duration = duration.max(MIN_LOCK_DURATION);

    // Transfer token from depositor to this contract
```

```
    depositToken.safeTransferFrom(_msgSender(), address(this), _amount);

    depositsOf[_receiver].push(
        Deposit({
            amount: _amount,
            start: uint64(block.timestamp),
            end: uint64(block.timestamp) + uint64(duration)
        })
    );

    uint256 bonusMultiplier;
    if (address(stakingBonusHandler) != address(0)) {
        bonusMultiplier = stakingBonusHandler.getMultiplier(_receiver);
    }

    // Calculate mint amount from amount staked, multiplier for staked duration and bonus
multiplier.
    uint256 mintAmount = (_amount *
        (getMultiplier(duration) + bonusMultiplier)) / 1e18;

    // Mint new pool tokens and assign them to the receiver address
    _mint(_receiver, mintAmount);
    emit Deposited(_amount, duration, _receiver, _msgSender());
    emit DepositedBonus(
        _amount,
        duration,
        _receiver,
        _msgSender(),
        bonusMultiplier
    );
}

function withdraw(uint256 _depositId, address _receiver) public virtual {
    require(_receiver != address(0), "Receiver cannot be zero");
    require(
        _depositId < depositsOf[_msgSender()].length,
        "BreederTimeLockPool.withdraw: Deposit does not exist"
    );
    Deposit memory userDeposit = depositsOf[_msgSender()][_depositId];
    require(
        block.timestamp >= userDeposit.end,
        "BreederTimeLockPool.withdraw: too soon"
    );

    //                     No risk of wrapping around on casting to uint256 since deposit
end always > deposit start and types are 64 bits
    uint256 shareAmount = (userDeposit.amount *
        getMultiplier(uint256(userDeposit.end - userDeposit.start))) / 1e18;

    _removeDepositAmount(_depositId, userDeposit.amount, _msgSender());
```

```
    // burn pool shares
    _burn(_msgSender(), shareAmount);

    // return tokens
    depositToken.safeTransfer(_receiver, userDeposit.amount);
    emit Withdrawn(_depositId, _receiver, _msgSender(), userDeposit.amount);
}
```

RECOMMENDATION

Burn the `stakingBonusHandler` shares along with the base and locking shares when users withdraw their tokens.

UPDATES

- *Dec 13, 2023*: This issue has been acknowledged and fixed.

2.2.2. `breedWhitelistTransfer` does not burn `lock` and `bonus` shares HIGH

Affected files:

- BreederWhitelistPool.sol

`breedWhitelistTransfer` function allows whitelisted addresses to transfer `Breed` to themselves on behalf of a user. It burns the minted shares but forget to burn the `lock` and `bonus` shares minted when depositing (`mintAmount = (_amount * (getMultiplier(duration) + bonusMultiplier)) / 1e18;`).

```
function _makeDeposit(
    uint256 _amount,
    uint256 _duration,
    address _receiver
) internal {
    require(_amount > 0, "BreederTimeLockPool.deposit: cannot deposit 0");
    // Don't allow locking > maxLockDuration
    uint256 duration = _duration.min(maxLockDuration);
    // Enforce min lockup duration to prevent flash loan or MEV transaction ordering
    duration = duration.max(MIN_LOCK_DURATION);

    // Transfer token from depositor to this contract
    depositToken.safeTransferFrom(_msgSender(), address(this), _amount);

    depositsOf[_receiver].push(
        Deposit({
            amount: _amount,
            start: uint64(block.timestamp),
            end: uint64(block.timestamp) + uint64(duration)
        })
```

```
    );

    uint256 bonusMultiplier;
    if (address(stakingBonusHandler) != address(0)) {
        bonusMultiplier = stakingBonusHandler.getMultiplier(_receiver);
    }

    // Calculate mint amount from amount staked, multiplier for staked duration and bonus
multiplier.
    uint256 mintAmount = (_amount *
        (getMultiplier(duration) + bonusMultiplier)) / 1e18;

    // Mint new pool tokens and assign them to the receiver address
    _mint(_receiver, mintAmount);
    emit Deposited(_amount, duration, _receiver, _msgSender());
    emit DepositedBonus(
        _amount,
        duration,
        _receiver,
        _msgSender(),
        bonusMultiplier
    );
}

function breedWhitelistTransfer(
    address userToWithdrawFrom,
    uint256[] calldata _depositIds,
    uint256 transferAndBurnAmount
) external onlyRole(WHITELIST_TRANSFER_ROLE) {
    // Used to track amount to burn and transfer.  Takes care of possibility of an attacker
submitting a
    // `transferAndBurnAmount` that is greater than the amount available within
`_depositIds` submitted.
    uint256 toBurnAndTransfer;

    // Adjusting amounts in user Deposit structs as needed.
    for (uint256 i = 0; i < _depositIds.length; ++i) {
        uint256 currentDepositId = _depositIds[i];
        require(
            currentDepositId < depositsOf[userToWithdrawFrom].length,
            "BreederWhitelistPool.breedWhitelistTransfer: Deposit does not exist"
        );
        uint256 currentDepositAmount = depositsOf[userToWithdrawFrom][
            currentDepositId
        ].amount;

        if (transferAndBurnAmount >= currentDepositAmount) {
            _removeDepositAmount(
                currentDepositId,
                currentDepositAmount,
```

```
                userToWithdrawFrom
            );

            // Adjust
            transferAndBurnAmount -= currentDepositAmount;
            toBurnAndTransfer += currentDepositAmount;
        } else {
            // Remove remaining.
            _removeDepositAmount(
                currentDepositId,
                transferAndBurnAmount,
                userToWithdrawFrom
            );

            // Adjust
            toBurnAndTransfer += transferAndBurnAmount;

            // In case there are extraneous depositIds that can not be withdrawn from
            break;
        }
    }

    // Burn EBreed
    _burn(userToWithdrawFrom, toBurnAndTransfer);
    // Transfer Breed
    depositToken.transfer(_msgSender(), toBurnAndTransfer);

    emit WhitelistBreedTransferred(
        userToWithdrawFrom,
        _msgSender(),
        toBurnAndTransfer,
        _depositIds
    );
}
```

RECOMMENDATION

Burn the shares with bonus like in the withdrawal process.

UPDATES

- *Dec 13, 2023*: This issue has been acknowledged and fixed.

2.2.3. Unable to withdraw tokens if users transfer their share INFORMATIVE

Affected files:

- BreederTimeLockPool.sol

While other pools disable transfer by default, `BreederTimeLockPool` does not disable transfer so if it is deployed as a standalone pool instead of extend by others, the shares in this pool can be transferred. The problem is that if user transfer their share to another, neither the recipient nor the sender can withdraw the staked tokens which leaves the staked tokens stuck in the pool. Because this contract is listed as base contract which is extended by other pools, we don't raise an issue here but only a notice that the transfer logic need to be adjusted if it is deployed as a standalone pool.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 13, 2023*	Public Report	Verichains Lab

Table 3. Report versions history