BIFROST

Staking Contract Security Audit

revision 1.0

Prepared for
Bifrost

Prepared by
Andrew Wesie / Theori
Brian Pak / Theori
Juno Im / Theori

December 4th, 2020

Theori

Table of Contents

Executive Summary

Starting on November 30, 2020, Theori assessed the Bifrost staking contract for one week. We focused on identifying issues that put deposits at risk, allow unauthorized modification of parameters, or diverge from the intended behavior. We found two issues that may result in the contract code behavior diverging from the model design. We identified two additional issues where an owner could mistakenly break the functionality of the contract. We did not find any critical flaws that would put user's funds at risk.

We started with a review of the design document to ensure that its formulas are sound. We then cross referenced the model and test cases with the contract code to verify that the behavior matches. Next, we reviewed the contract code for common programming errors and validated the access controls. Lastly, we enumerated a set of recommendations for how to improve the quality of the code and its deployment. For some of the issues and recommendations we provide exemplar code, but this is only for demonstration purposes.

Scope

A smart contract that implements a token staking service provided by Bifrost is audited. The following is the list of the files reviewed by Theori for this assessment.

- Distribution_Model_Contract_Model_Design.pdf
- DistributionModelV3.sol
- ERC20.sol
- module/eventModule.sol
- module/externalModule.sol
- module/internalModule.sol
- module/safeMath.sol
- module/storageModule.sol
- module/viewModule.sol

The code was received on November 30th, 2020. Commit hash was not given.

Correctness Verification

Design document

The design document provides the intended behavior of the distribution model and serves as the reference to determine if the contract is behaving as intended. We specifically focused on section 3.3("Strategy") which contains the equations that should match with the solidity code.

Using the example from section 3.3, we verify the derivation of $RewardLane$, where the parameters (d, $R_{b,1}$) and total deposits (D_t) are fixed for a duration of n blocks:

d :	The decrement amount of R_b per block
n :	The number of blocks
D_t :	The total deposits
D_{user} :	The deposits for a single user
R_b :	The total rewards for all users for a single block
$R_{b,1}$:	The total rewards for the first block
R_u :	The reward unit for a single block
R_{user} :	The reward for a user for a single block

$$R_{user} = \frac{D_{user}}{D_t} R_b = D_{user} R_u$$

$$R_u = \frac{R_b}{D_t}$$

$$R_{b,i} = R_{b,i-1} - d$$

$$\sum_{i=1}^{n} R_{user,i} = D_{user} \sum_{i=1}^{n} R_{u,i} = D_{user} RewardLane$$

$$RewardLane = \sum_{i=1}^{n} R_{u,i}$$

$$RewardLane = \frac{\sum_{i=1}^{n} R_{b,i}}{D_t} = \frac{nR_{b,1} - (d + 2d + \cdots + (n-1)d)}{D_t}$$

$$RewardLane = \frac{nR_{b,1} - d(n-1)(1+n-1)/2}{D_t} = \frac{2nR_{b,1} - n(n-1)d}{2D_t}$$

This equation can be used to calculate the rewards for a user over the span of n blocks if the parameters, total deposits, and the user's deposit do not change. If the parameters or total deposits change, then $RewardLane$ must first be calculated. If D_{user} changes, then R_{user} must also be calculated before the update.

While reviewing the design document, we noted typos in section 3.2:

- The diagram calculates the wrong value of R for user 3
- The ratios for each user at block height 2 in the text is incorrect
- The R_{user} for each user at block height 2 in the text is incorrect

Smart contract

We checked the smart contract code against the model in the design document:

- *_redeemAll* calculates R_{user} before *_deposit* or *_withdraw* can modify D_{user}
- *_redeemAll* calls *_updateRewardLane* to calculate the latest $RewardLane$
- Since *_redeemAll* is always called before *_deposit* and *_withdraw*, it is also true that *_updateRewardLane* is always called before D_t is changed
- *setRewardVelocity* calls *_updateRewardLane* before it updates parameters

One complication in the smart contract is the addition of a queue of parameter updates, stored in *registeredPoints*, where each update has a starting block number, new block reward, and a new block reward decrement. These parameter updates are applied within *_updateRewardLane*. If a parameter update needs to be applied, the $RewardLane$ is calculated using the current parameters up until the update's starting block. Then, the parameter update is applied, preserving the assumptions enumerated above.

The calculation of $RewardLane$ is within *_calcNewRewardLane*, based on *_totalDeposit* (D_t), *_rewardPerBlock* ($R_{b,1}$), *_decrementUnitPerBlock* (d), and *delta* (n). Unfortunately, the solidity code does not take the same form as the design document, but the behavior is the same:

= _meanOfInactiveLane(_rewardPerBlock, delta, _decrementUnitPerBlock) * (1 / totalDeposit) * delta

= (2 * _rewardPerBlock * delta – delta * (delta - 1) * _decrementUnitPerBlock) / (2 * delta) * (1 / totalDeposit) * delta

= (2 * _rewardPerBlock * delta – delta * (delta - 1) * _decrementUnitPerBlock) / (2 * totalDeposit)

$$= \frac{2nR_{b,1} - n(n-1)d}{2D_t}$$

The gas efficiency of the solidity code could be improved by inlining *_meanOfInactiveLane* and simplifying the terms. This would also eliminate the need for checking if delta is zero.

The calculation of the new block reward is also within *_calcNewRewardLane*. It calls *_getNewRewardPerBlock* which multiplies the decrement unit by the delta and subtracts the result from the current block reward, while ensuring the block reward does not become negative. This behavior differs from the design document:

$$\text{Design document: } R_{b,i} = R_{b,i-1} - d$$
$$\text{Solidity code: } R_{b,i} = \max\,(R_{b,i-1} - d, 0)$$

While the difference appears minor, it invalidates the derivation of $RewardLane$ when $nd > R_{b,1}$. This also implies that the behavior of *_calcNewRewardLane* is incorrect for this case since it relies on the correctness of the derivation. Additional logic should be added to *_calcNewRewardLane* to handle this corner case, since currently it may cause a reward that is too small or *safeSub* to fail due to a subtraction overflow.

Another difference between the calculation of the new block reward in `_calcNewRewardLane` and the design document is the case when `_totalDeposit` is zero. In section 4 of the design document, the `rewardPerBlock` is updated even when `totalDeposit` is zero. But in the solidity code, `rewardPerBlock` is updated only if `totalDeposit` is non-zero. This is likely a bug in the solidity code that should be corrected.

Findings

These are the potential issues that may have correctness and/or security impacts. We advise Bifrost team to remediate found issues quickly to ensure the safety of the contract.

Summary

#	ID	Title	Severity
1	THE-BIFROST-001	Potentially incorrect reward calculation	High
2	THE-BIFROST-002	Incorrect behavior when *`totalDeposit`* is zero	Medium
3	THE-BIFROST-003	Failure to ensure *`registeredPoints`* is sorted	Low
4	THE-BIFROST-004	Failure to ensure only future points may be deleted	Low

Issue #1: Potentially incorrect reward calculation

ID	Summary	Severity
THE-BIFROST-001	Reward calculation is incorrect if `delta` times `_decrementUnitPerBlock` is greater than the starting block reward	High

As described in the "Correctness Verification" section, `_calcNewRewardLane` relies on the definition: $R_{b,i} = R_{b,i-1} - d$. If `delta` times `_decrementUnitPerBlock` is greater than `_rewardPerBlock`, then that definition is incorrect and the `_meanOfInactiveLane` formula will return a wrong value.

This corner case should be handled with some additional logic. For example, in`_calcNewRewardLane`:

```
function _calcNewRewardLane(
    uint256 _rewardLane,
    uint256 _totalDeposit,
    uint256 _rewardPerBlock,
    uint256 _decrementUnitPerBlock,
    uint256 delta) internal pure returns (uint256, uint256) {
        uint256 newRewardPerBlock = _getNewRewardPerBlock(_rewardPerBlock, _decrementUnitPerBl
ock, delta);
        if(_totalDeposit != 0) {
            uint256 distance;
            if (newRewardPerBlock == 0) {
                uint256 tmpDelta = _rewardPerBlock / _decrementUnitPerBlock;
                distance = expMul( _meanOfInactiveLane(_rewardPerBlock, tmpDelta, _decrementUn
itPerBlock), safeMul( expDiv(one, _totalDeposit), tmpDelta) );
                distance = safeAdd( distance, _getNewRewardPerBlock(_rewardPerBlock, _decremen
tUnitPerBlock, tmpDelta) );
            } else {
                distance = expMul( _meanOfInactiveLane(_rewardPerBlock, delta, _decrementUnitP
erBlock), safeMul( expDiv(one, _totalDeposit), delta) );
            }
            uint256 newRewardLane = safeAdd(_rewardLane, distance);
            return (newRewardLane, newRewardPerBlock);
        }
        return (_rewardLane, newRewardPerBlock);
}
```

Issue #2: Incorrect behavior when totalDeposit is zero

ID	Summary	Severity
THE-BIFROST-002	Block reward is not updated in `_calcNewRewardLane` if `_totalDeposit` is zero	Medium

In `_calcNewRewardLane`, the block reward should always be updated even if `_totalDeposit` is zero. Section 4 of the design document demonstrates the correct behavior. The solidity code should be corrected, for example:

```
function _calcNewRewardLane(
    uint256 _rewardLane,
    uint256 _totalDeposit,
    uint256 _rewardPerBlock,
    uint256 _decrementUnitPerBlock,
    uint256 delta) internal pure returns (uint256, uint256) {
        uint256 newRewardPerBlock = _getNewRewardPerBlock(_rewardPerBlock, _decrementUnitPerBl
ock, delta);

        if(_totalDeposit != 0) {
            uint256 distance = expMul( _meanOfInactiveLane(_rewardPerBlock, delta, _decrementU
nitPerBlock), safeMul( expDiv(one, _totalDeposit), delta) );
            uint256 newRewardLane = safeAdd(_rewardLane, distance);
            return (newRewardLane, newRewardPerBlock);
        }
        return (_rewardLane, newRewardPerBlock);
}
```

Issue #3: Failure to ensure registeredPoints is sorted

ID	Summary	Severity
THE-BIFROST-003	*registerRewardVelocity* fails to ensure that the *registeredPoints* array is sorted	Low

The loop over *registeredPoints* in *_updateRewardLane* assumes that the array is sorted. *registerRewardVelocity* should ensure that this property is always true.

If a point with an out-of-order block number is added to *registeredPoints*, then later points in *registeredPoints* may never be processed. A simple fix is to require the array remains sorted, for example:

```
function registerRewardVelocity(uint256 _blockNumber, uint256 _rewardPerBlock, uint256 _decrem
entUnitPerBlock) onlyOwner external {
    require(_blockNumber > block.number, "new Reward params should register earlier");
    require(registeredPoints.length == 0 || _blockNumber > registeredPoints[registeredPoints.l
ength-1].blockNumber, "Earlier velocity points are already set.");
    _registerRewardVelocity(_blockNumber, _rewardPerBlock, _decrementUnitPerBlock);
}
```

Issue #4: Failure to ensure only future points may be deleted

ID	Summary	Severity
THE-BIFROST-004	`deleteRegisteredRewardVelocity` fails to ensure that only future points may be deleted	Low

`deleteRegisteredRewardVelocity` should verify that the index to be deleted is greater than or equal to `passedPoint`.

If an index less than `passedPoint` is removed using `deleteRegisteredRewardVelocity`, then `passedPoint` needs to be updated as well otherwise some point in `registeredPoints` will not be processed. It is better to just not allow these points to be removed because they have already been processed and removing them will not undo their effect. For example:

```
function deleteRegisteredRewardVelocity(uint256 _index) onlyOwner external {
    require(_index >= passedPoint, "Reward velocity point already passed.");
    _deleteRegisteredRewardVelocity(_index);
}
```

Code Quality Recommendations

These are the recommendations to improve the code quality for better readability and optimization. They do not impose any immediate security impacts.

Summary

#	Title	Type	Importance
1	Fix typos and misspellings in design document and solidity code	Readability	Minor
2	Use a proxy contract so you can upgrade the contract	Extensibility	Major
3	Allow the ownership to be transferred	Extensibility	Major
4	Follow best practices for owner wallet	Safety	Major
5	Reduce gas usage by simplifying the math	Optimization	Minor
6	Reduce gas usage by clearing storage	Optimization	Minor
7	Reduce gas usage when deleting a point	Optimization	Minor
8	Consistent usage of variables	Readability	Minor
9	Consider using OpenZeppelin's ERC20 implementation	Safety	Minor

Recommendation #1: Fix typos and misspellings in design document and solidity code

In the design document, the diagram in section 3.2 is inconsistent with itself and the accompanying text.

- In the comment for `_meanOfInactiveLane` (internalModule.sol): "the **avaerage** of the **RewardLance** of the inactive" (average, RewardLane).
- In the comment for `_setParams` (internalModule.sol): "The **decerement** amount of the reward token per a block" (decrement).
- In the comment for `_calcNewRewardLane` (internalModule.sol): "**Thte** total deposit" (The).
- In the comment for `_meanOfInactiveLane` (internalModule.sol), the equations should be:
  return Sn / n, where Sn = ((n{2*a - (n-1)d}) / 2)
  == ((2na - n(n-1)d) / 2) / n

Recommendation #2: Use a proxy contract so you can upgrade the contract

Ethereum contracts are immutable. If any security vulnerabilities or correctness bugs are found, a new contract is created with a new address. If the contract address is used publicly, there is a risk that users will continue to use the old contract instead of the new contract, though some users may consider this to be a benefit.

There are two options: 1) make the contract upgradeable by using a proxy contract that points to an implementation contract address that can be modified by the owner, or 2) add a feature for the owner to completely disable or kill the old contract. The first option, an upgradeable contract, is usually preferred for usability. The second option is more secure because users can opt into the new contract while not being able to accidentally continue to use the old contract.

Recommendation #3: Allow the ownership to be transferred

Currently, the address that creates the contract will forever be the owner of the contract. This is risky because you may need to rotate keys in the future and you would want to be able to transfer the ownership to the new address. Otherwise, you will need to create a new contract which has the downsides described previously.

Recommendation #4: Follow best practices for owner wallet

Use a multi-sig wallet for the owner account to lower the risk of a malicious insider manipulating the contract parameters. Consider using an HSM to prevent the theft of the private keys.

Recommendation #5: Reduce gas usage by simplifying the math

It is possible to reduce gas usage in `_meanOfInactiveLane` by simplifying the math.

`_calcNewRewardLane` calls `_meanOfInactiveLane` which could be simplified to:

```
if (n > 0)
    return safeDiv(safeSub(safeMul(2,a), safeMul(safeSub(n,1), d)), 2);
else
    return 0;
```

By inlining `_meanOfInactiveLane` into `_calcNewRewardLane` further optimizations are possible. For instance, the check that `delta` is not zero can be safely eliminated if `safeSub(n,1)` is changed to `(n - 1)`, because the result is multiplied by `delta`, which if `delta` is zero the multiplication will always return zero.

Recommendation #6: Reduce gas usage by clearing storage

Reduce gas usage by clearing the point in *registeredPoints* after it is applied in *_updateRewardLane*.

A call that applies a point from *registeredPoints* will use more gas than a call that does not need to do this additional work. If the storage for the just applied point is zeroed, some gas will be refunded resulting in less overhead when applying a reward point.

```
delete registeredPoints[i];
```

Recommendation #7: Reduce gas usage when deleting a point

Currently, *deleteRegisteredRewardVelocity* uses a linear algorithm to remove a point from *registeredPoints*. This is expensive because each iteration requires both a storage load and a storage store operation. If it is expected that the number of iterations may be large, then it would be beneficial to modify the data structure to allow for delete in O(1) time.

For example, if we treat a registered point as deleted if its *blockNumber* is zero, then:

```
function _deleteRegisteredRewardVelocity(uint256 _index) internal {
    uint256 len = registeredPoints.length;
    require(len != 0 && _index < len, "error: no elements in registeredPoints");
    RewardVelocityPoint memory point = registeredPoints[_index];
    emit DeleteRegisterRewardParams(_index, point.blockNumber, point.rewardPerBlock, point.dec
rementUnitPerBlock);
    delete registeredPoints[_index];
}

function _updateRewardLane() internal returns (uint256) {
    ...

    for(uint256 i=vars.memPassedPoint; i<vars.len; i++) {
        RewardVelocityPoint memory point = registeredPoints[i];
        if (point.blockNumber == 0) {
            vars.tmpPassedPoint = i+1;
            continue;
        }

    ...
}
```

Recommendation #8: Consistent usage of variables

In `_deposit`, the storage variable `totalDeposited` is cached in the stack variable `totalDeposit`. Then, the storage variable `totalDeposited` is accessed again later:

```
    totalDeposit = safeAdd(totalDeposited, amount);
```

Use `totalDeposit` instead of `totalDeposited`.

In `_withdraw`, the storage variable `totalDeposited` is cached in the stack variable `totalDeposit`. Then, the storage variable `totalDeposited` is accessed again later:

```
    totalDeposit = safeSub(totalDeposited, amount);
```

Use `totalDeposit` instead of `totalDeposited`.

In `_updateRewardLane`:

```
    if(vars.memLastBlockNum != vars.tmpLastBlockNum) lastBlockNum = vars.memThisBlockNum;
```

While the behavior is correct, it does not match the other code patterns. Use `vars.tmpLastBlockNum` instead of `vars.memThisBlockNum`.

Recommendation #9: Consider using OpenZeppelin's ERC20 implementation

One improvement of the OpenZeppelin implementation are the non-standard `increaseAllowance` and `decreaseAllowance` functions. These functions mitigate a possible race condition where a user wants to reduce the allowance for a spender, but the spender first uses their allowance and then gets the new allowance additionally.

Further information about this theoretical attack can be found at:

- https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#IERC20-approve-address-uint256-
- https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

Conclusion

We reviewed the design documentation and implementation of Bifrost's staking contract. We verified the contract code implementation against the model described in the documentation. While there are a few minor issues, we have not found any critical security vulnerabilities that would risk the safety of the staked tokens. We also suggest readability improvements and code optimization.

Note that the audited contract is a small portion of the company's multichain DeFi platform, called BiFi[1]. We recommend that the rest of the system is audited as well to ensure the security of the platform as a whole.

[1] https://bifi.finance/