Ramen Finance
Audit Report

September 30, 2024

Conducted by:

Bogo Cvetkov (b0g0), Independent Security Researcher

Table of Contents

1. About b0g0 3
2. About Ramen Finance 3
3. Risk Classification 3
   - 3.1. Impact 3
   - 3.2. Likelihood 4
   - 3.3. Handling severity levels 4
4. Executive Summary 4
5. Disclaimer 4
6. Findings 7
   - 6.1. Medium Severity 7
     - 6.1.1. Unstaking could be DOSed if lockPeriod is updated 7
     - 6.1.2. Changing ramen token address can break unstaking for new deposits 8
     - 6.1.3. Deposits are allowed even after contract is revoked 10
   - 6.2. Low Severity 12
     - 6.2.1. Array lookup might start reverting if it gets too big 12
     - 6.2.2. Use safeTransfer instead of transfer 13
   - 6.3. Governance 14
     - 6.3.1. Governance Privileges 14
   - 6.4. Informational 15
     - 6.4.1. Insufficient validation 15
     - 6.4.2. Gas optimizations 16
     - 6.4.3. Emit events on important state changes 18

1. About b0g0

Bogo Cvetkov (b0g0) is a smart contract security researcher with a proven track record of consistently uncovering vulnerabilities in a wide spectrum of DeFi protocols. Constantly pushing the limits of his expertise, he strives to be a superior security partner to any protocol & client he dedicates himself to!

2. About Ramen Finance

Ramen Finance is a Berachain-native token launchpad protocol powering liquidity bootstrapping and price discovery for new assets.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality
- **Low** - funds are not at risk

3.2. Likelihood

- **High** - almost certain to happen, easy to perform, or highly incentivized
- **Medium** - only conditionally possible, but still relatively likely
- **Low** - requires specific state or little-to-no incentive

3.3. Handling severity levels

- **Critical** - Must fix as soon as possible (if already deployed)
- **High** - Must fix (before deployment if not already deployed)
- **Medium** - Should fix
- **Low** - Could fix
- **Governance** - Could fix

4. Executive Summary

For the duration of 5 days **b0g0** has invested his expertise as a security researcher to analyze the smart contracts of **Ramen Finance** protocol and assess the state of its security. For that time a total of 9 issues have been detected, out of which **0** have been assigned a severity level of **High** and **3** a severity level of **Medium**.

5. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This effort is limited by time, resources, and expertise. My evaluation of the codebase aims to uncover as many vulnerabilities as possible, given the above limitations! Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended!

Overview

Project	Ramen Finance
URL	https://ramen.finance/
Platform	Berachain
Language	Solidity
Repo	https://github.com/ramenfinance/ramen-finance/tree/master/contracts-ramenfinance/contracts
Commit Hash	-
Mitigation	468f7f981ea7a136cfc4b45494afe117ee17b8e9
Dates	26 September - 30 September 2024

Scope

Contract	Address
gRamen.sol	-
ClaimTimelock.sol	-
TimelockFactory.sol	-

Issue Statistic

Severity	Count
High	0
Medium	3
Low/Informational	6
Total	**9**

6. Findings

6.1. Medium Severity

6.1.1. Unstaking could be DOSed if lockPeriod is updated

Context: gRamen.sol

Description :

When a stake is created a lock time is calculated for it based on the lockPeriod variable :

```
Stake memory s = Stake({
            amount: _amount,
            stakedAt: uint64(block.number),
            expiredAt: uint64(block.number) + lockPeriod
        });
```

Later during unstaking the calculated expiredAt value is used to determine if any penalty rate should be applied on the amount returned to the owner :

```
function getPenaltyRate(
        address _addr,
        uint256 _stakeIndex
    ) public view returns (Stake memory _s, uint256 _penaltyRate) {
        Stake memory s = stakes[_addr][_stakeIndex];
        . . .
        if ((block.number - s.stakedAt) < lockPeriod) {
            uint256 remainingWeeks = s.expiredAt - block.number;
            . . .;
        }
      . . .
    }
```

In case lockPeriod gets updated between stake() & unstake(), unstaking could be DOSed for some time, due to an underflow error.

Here is a concrete example:

- lockPeriod is set to 16 weeks
- Alice stakes 100 ramen and expiredAt is set to block.number + lockPeriod
- lockPeriod is updated to 20 weeks
- Alice decides to unstake at week 17 and since 17 < 20, the if branch is activated

```
- if ((block.number - s.stakedAt) < lockPeriod)
```

- The calculation reverts, since 16 weeks (s.expiredAt) is deducted from 17 weeks (block.number)

```
-  uint256 remainingWeeks = s.expiredAt - block.number;
```

As result unstaking is blocked for the next 3 weeks, until the new lockPeriod has elapsed.

Recommendation

Consider saving the lockPeriod at the time of the stake in the Stake struct and use it to calculate the penalty tax in order to prevent the above scenario.

Resolution:

Fixed - lockPeriod has been changed into an immutable variable

6.1.2. Changing ramen token address can break unstaking for new deposits

Context: gRamen.sol

Description:

This is how stakes are created in the system:

```
function _stake(address _to, uint128 _amount) internal {
        require(_amount > 0, 'stake: amount must be greater than zero');
        //@audit-info - save ramen token address
        Stake memory s = Stake({
            amount: _amount,
            stakedAt: uint64(block.number),
            expiredAt: uint64(block.number) + lockPeriod
        });
        . . .
        ramen.safeTransferFrom(msg.sender, address(this), _amount);
        _mint(_to, _amount);
        . . .
    }
```

And this is how unstaking works:

```
function _unstake(
        uint256 _stakeIndex
    ) internal returns (uint128 _remainingAmount) {
        . . .
        ramen.safeTransfer(msg.sender, s.amount - uint128(penaltyRate));
        ramen.burn(penaltyRate);
        _burn(msg.sender, s.amount);
    }
```

Upon staking the amount and time of staking is saved inside the Stake struct. The current ramen address in which the deposit is made is not saved in the Stake struct.

During unstaking the tokens that are returned to the owner are again based on the current ramen token.

In case the setToken() function is used to change the ramen token address, the following scenario is possible:

- ramen address is address(1)

- Alice stakes 100 ramen (address(1)) tokens
- ramen address is updated to address(2)
- Bob stakes 100 ramen(address(2)) tokens
- Alice unstakes and receives 100 ramen(address(2)) tokens
- Bob tries to unstake but he can't, since Alice has taken his tokens

The probability of the above scenario is low, since the ramen token is not very likely to change. However the functionality for that exists - setToken() - and in case it happens it would seriously affect the integrity of the system

Recommendation:

If ramen is not expected to change, consider making the variable immutable and remove the setter function.

If the team decides to keep the functionality, then consider saving the current ramen address in the Stake struct, so that unstaking happens in the token that was used at the time of staking.

Resolution:

Fixed - ramen has been changed into an immutable variable

6.1.3. Deposits are allowed even after contract is revoked

Context: ClaimTimelock.sol

Description:

The ClaimTimelock contract accepts deposits of particular token and allows them to be claimed only after timelockStarted has been enabled and unlockTimestamp has elapsed :

```
function claim(address beneficiary) external nonReentrant {
        require(timelockStarted, 'ClaimTimelock: timelock is not started yet');
        require(vestedAmount[beneficiary] > 0, 'nothing to claim'); //checks
        require(
            block.timestamp > unlockTimestamp,
            'ClaimTimelock: not yet unlocked'
```

```
        );
        . . .
    }
```

The contract admin can enable the timelock through the startTimelock() function, which is a one time operation:

```
function startTimelock(
        uint256 _unlockTimestamp
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(
            unlockTimestamp == uint256(0),
            'ClaimTimelock: timelock already started'
        );
        unlockTimestamp = _unlockTimestamp;
        timelockStarted = true;
    }
```

The contract also implements a revoke() function, which sets the timelockStarted variable to false - something that cannot be undone. This is intended to permanently disable any claims. However there is nothing stopping deposit() from functioning. This means that if any deposits are made after revoke() (consciously or accidentally) the amounts will get stuck in the contract and have to be rescued by calling revoke() by DEFAULT_ADMIN_ROLE.

Recommendation:

Consider adding a check in deposit() if the contract has been revoked and revert with an error.

Resolution:

Fixed

6.2. Low Severity

6.2.1. Array lookup might start reverting if it gets too big

Context: gRamen.sol

Description:

The contract uses the stakes[address] array state variable to store each new stake for an address. The following view functions have been defined to get different info about the stakes:

- getStakes()
- getPenaltyRates()

Both of them loop through the whole stakes array, which might be problematic in case the array gets too big. This is especially true for the getPenaltyRates() function, where getPenaltyRate() gets called for each element.

Since the array only grows, there might be a scenario that if it gets too big and gas prices are high the functions will revert. This might create problems for external parties that integrate with the protocol

Recommendation:

A best practise when working with functions that loop through arrays is to provide start and end indexes, so that the caller can control which parts of the array to iterate through. Consider adding startIndex and endIndex parameters to the above functions.

Resolution:

Fixed

6.2.2. Use safeTransfer instead of transfer

Context: ClaimTimelock.sol

Description:

Using safeTransfer is considered best practise when transferring tokens - it makes sure that there won't be any problems if some non-standard ERC20 tokens are used (USDT). Currently transfer instead of safeTransfer is used in the following functions of ClaimTimelock:

- revoke()
- claim()

Recommendation:

Consider using safeTransfer instead of plain transfer, to make sure you have covered potential edge cases.

Resolution:

Fixed

6.3. Governance

6.3.1. Governance Privileges

Context: gRamen.sol

Description:

The contract DEFAULT_ADMIN_ROLE account has control over several variables that can impact the outcome of a transaction:

- setToken
- setLockPeriod
- setStartDecay

Recommendation:

Consider incorporating a Gnosis multi-signature contract as the DEFAULT_ADMIN_ROLE and ensuring that the Gnosis participants are trusted entities

Resolution:

Acknowledged

6.4. Informational

6.4.1. Insufficient validation

Context: gRamen.sol

Description:

All issues related to validation are collected here to keep the report focused and easy to read:

- Inside the constructor of gRamen contract validate that the _ramen parameter is not address(0). It's a simple guard that would prevent potential mistakes during deployment
- Inside gRamen.restake() validate that the remainingAmount parameter is not 0

Recommendation:

Consider implementing the above mentioned recommendations

Resolution:

Fixed

6.4.2. Gas optimizations

Context: gRamen.sol

Description:

All issues related to gas are collected here to keep the report focused and easy to read:

- Inside gRamen.getPenaltyRate() move the decayPeriod calculation inside the if block, to save a bit of gas when lockPeriod has expired
- Inside gRamen._unstake() consider adding an additional boolean parameter (bool _sent) which can be used during restaking to save a substantial amount of gas. Currently gRamen.restake() uses _unstake() which always transfers the ramen to the owner, to only transfer it back in the same transaction for the new stake. Since the ramen tokens are already in the contract, there is no need to transfer them back and forth. Optimizing the flow would cut following gas costs:
  - Transferring to the owner
  - Transferring from the owner back to the contract
  - Approval from the owner so that the contract can transfer back the ramen

  An additional benefit is that the flow would be more intuitive from UI perspective - users won't have to give an additional approval and spend gas, before they restake. A possible approach might look like this:

```
function _unstake(
        uint256 _stakeIndex
        bool _send
    ) internal returns (uint128 _remainingAmount) {
        . . .

      if (_send) {
        ramen.safeTransfer(msg.sender, s.amount - uint128(penaltyRate));
       }
        . . .

        return s.amount - uint128(penaltyRate);
    }
```

```
function restake(uint256 _stakeIndex) external {
        uint128 remainingAmount = _unstake(_stakeIndex, false);

        Stake memory s = Stake({
            amount: remainingAmount,
            stakedAt: uint64(block.number),
            expiredAt: uint64(block.number) + lockPeriod
        });

        stakes[msg.sender][_stakeIndex] = s;

 ->>     //ramen.safeTransferFrom(msg.sender, address(this), remainingAmount);
        _mint(msg.sender, remainingAmount);

        emit AddStake(s, msg.sender, _stakeIndex);
    }
```

- Inside gRamen.unstake() consider checking if the stake was not unstaked already, currently the whole flow with transfer and burning runs every time, even when amounts are 0. Also an event is emitted each time. All of this wastes unnecessary gas and emits empty events
- Remove the _setStake() internal function, since it is not used anywhere - this will reduce bytecode size.

Recommendation:

Consider implementing the above mentioned recommendations

Resolution:

Fixed

6.4.3. Emit events on important state changes

Context: ClaimTimelock.sol

Description:

The following state changing functions do not emit events:

- deposit()
- claim()
- startTimelock()
- revoke()

It is considered a best practise to emit events, that mark changes in the state of the smart contract

Recommendation:

Consider emitting relevant events in the above functions

Resolution:

Acknowledged