our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Beraji-KO

SECURITY REVIEW

Date: 27 February 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Beraji-KO

Beraji-KO is a token and NFT liquid staking PvP card game within the Beraji ecosystem, allowing only whitelisted tokens and NFTs on Berachain

Players enhance their cards, representing blue-chip Berachain projects, using real-time data such as token values and social metrics. In addition to competing in 24-hour tournaments, players can also create bets on the matches they play.

By strategically staking tokens and NFTs, players unlock special abilities, ascend leaderboards, and secure rewards, creating a dynamic and competitive environment that mirrors the dynamics of the Berachain market.

Learn more about Beraji-KO's concept and the technicalities behind it here

4. Risk Classification

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

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review was conducted over the span of 5 days by the core Shieldify team. The code is written professionally with all best practices considered. However, the security review identified several High and Medium-severity issues, concerning a possible loss of funds for the stakers and owners, along with some other misconfigurations, and two issues of low severity regarding overpayment resulting in stuck funds and potential denial of service of the reward updating functionality.

Shieldify extends their gratitude to Beraji for cooperating with all the questions our team had and strictly following the provided recommendations.

5.1 Protocol Summary

Project Name	**Beraji-KO**
Repository	beraji-ko
Type of Project	DeFi, Staking
Audit Timeline	5 days
Review Commit Hash	290103f27365cc419fe0a934feaf5f18e6e12d3a
Fixes Review Commit Hash	de6902450de75b82e27729b5acf4ba6d349206d4

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/Token.sol	19
contracts/StakingKo.sol	469
contracts/NativeFee.sol	30
contracts/interface/IStakingKo.sol	70
Total	**588**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical/High** issues: **4**
- **Medium** issues: **3**

- **Low** issues: **2**
- **Info** issues: **1**

ID	Title	Severity	Status
[H-01]	Attacker Can Redirect Funds To Staking Via Allowances	High	Fixed
[H-02]	Users Might Not Be Able To Fully Claim	High	Fixed
[H-03]	Stakers Can Lose Their Earned `aSugar`	High	Fixed
[H-04]	Owner Cannot Claim Token Fees	High	Fixed
[M-01]	Fee On Transfer Tokens Are Not Supported	Medium	Fixed
[M-02]	Users Can Use Outdated Prices As Long As The Signature Expiry Has Not Passed	Medium	Acknowledged
[M-03]	Stakers Accumulate `aSugar` Based On The Spot Price During The Claim	Medium	Acknowledged
[L-01]	The `updateReward()` Function Could DOS Or Become Very Expensive	Low	Acknowledged
[L-02]	Overpayment of Native Fee Results in Stuck Native Funds	Low	Fixed
[I-01]	Usage of Transfer Instead of Call For Native Funds	Info	Fixed

7. Findings

[H-01] Attacker Can Redirect Funds To Staking Via Allowances

Severity

High Risk

Description

In `deligateStake()` the `transferFrom()` function uses an arbitrary value for the `from` address. Before users or owner call `stake()`, `deligateStake()`, `createReward()`, `repayment()` or `depositRewardAmount()` they first approve the StakingKo contract. Once this is done the attacker can frontrun their next call and call `deligateStake()` on their behalf.

This way funds that were intended to be added by the owner as rewards will be staked on behalf of the owner. The owner will then need to wait for the lock to expire and try again.

In the case of users that try to stake in `lockId` 0, this lock can be changed to any other lock when the attacker stakes on behalf of the user.

Location of Affected Code

File: contracts/StakingKo.sol

```
function deligateStake(
    address _account,
    uint256 _poolId,
    uint256 _lockId,
    uint256 _amount,
    // aSugar printer
    uint256 _stPrice,
    uint256 _expriedTime,
    bytes memory _sig
)
    public
    payable
    validPool(_poolId)
    poolIsActive(_poolId)
    validPoolLock(_poolId, _lockId)
    updatePoolPrice(_poolId, _stPrice, _expriedTime, _sig)
    updateReward(_poolId, _account)
    payNativeFee
    whenNotPaused
{
    require(_amount != 0, "amount = 0");
    Pool storage pool = pools[_poolId];

@>  pool.stakingToken.safeTransferFrom(_account, address(this), _amount);

    // code

    emit Staked(_poolId, _account, stakeAmount, _stPrice);
}
```

Impact

The attacker can redirect funds intended for rewards and stake them instead + he can change the settings of the stake for the users who try to stake.

Recommendation

When staking on behalf of somebody use the funds of the caller instead. Use TransferFrom with `msg.value` instead of with arbitrary address.

Team Response

Fixed.

[H-02] Users Might Not Be Able To Fully Claim

Severity

High Risk

Description

The `fullyClaimReward()` function loops through all locks and all rewards for a pool. Let's say we have 10 reward tokens and 5 locks.

This is 10 reward tokens * 5 locks = 50 loops.

In the `updateReward()` modifier we do the same 50 more loops.

It looks like `_claimReward()` incorrectly has the `updateReward` modifier which makes the looping exponential:

1. 50 loops in `updateReward()` before the execution of `fullyClaimReward()`.
2. 50 loops of `_claimReward()` _ 50 loops for each `_claimReward()` due to `updateReward()` modifier: 50 _ 50 = 2500 loops
3. 50 + 2500 = 2550 loops for calling `fullyClaimReward()` once.

This at some point could reach the block gas limit and prevent the user from calling `fullyClaimReward()` successfully.

Location of Affected Code

File: contracts/StakingKo.sol

```
function _claimReward(
    uint256 _poolId,
    uint256 _lockId,
    uint256 _rewardId
)
    internal
    validPool(_poolId)
    validPoolLock(_poolId, _lockId)
    validReward(_poolId, _rewardId)
@>  updateReward(_poolId, msg.sender)
{
    uint256 rwAmount = userRewards[_poolId][_lockId][_rewardId][msg.
       sender];
    PoolReward storage rw = mapPoolRewards[_poolId][_rewardId];

    if (rwAmount != 0) {
        userRewards[_poolId][_lockId][_rewardId][msg.sender] = 0;
        rw.rewardToken.safeTransfer(msg.sender, rwAmount);
        rw.balance -= rwAmount;
        emit RewardClaimed(msg.sender, _poolId, _lockId, _rewardId,
           rwAmount);
    }
}
```

Impact

The `fullyClaimReward()` function can get DOSed or very expensive gas-wise.

Recommendation

Remove `updateReward()` modifier from `_claimReward()`.

Team Response

Fixed.

[H-03] Stakers Can Lose Their Earned `aSugar`

Severity

High Risk

Description

Users who stake a second time without calling `fullyClaimReward()` will lose their accumulated unclaimed `aSugar`. Furthermore, the second stake could be executed by somebody else via `deligateStake()` if there is an active allowance.

Example:

The user stakes 100e18 of a token. He gets minted X amount of aSugar based on lock leverage and stakeToken price.

`cheque.lastPrintedAt` gets updated to `block.timestamp`. This is very important since the accumulation of aSugar is based on the staked amount and the difference between the `block.timestamp` when calling `fullyClaimReward()` and `cheque.lastPrintedAt`.

Since `cheque.lastPrintedAt` gets updated in `deligateStake()` and the accumulated amount from previous stakes is not minted then the user loses this accumulation.

Flow:

1. User stakes
2. Time passes
3. User stakes again
4. Calls `fullyClaimReward()` but the aSugar generated between both stakes is lost.

Location of Affected Code

File: contracts/StakingKo.sol

```
function deligateStake(
    address _account,
    uint256 _poolId,
    uint256 _lockId,
    uint256 _amount,
    uint256 _stPrice,
    uint256 _expriedTime,
    bytes memory _sig
)
    public
    payable
    validPool(_poolId)
    poolIsActive(_poolId)
    validPoolLock(_poolId, _lockId)
    updatePoolPrice(_poolId, _stPrice, _expriedTime, _sig)
    updateReward(_poolId, _account)
    payNativeFee
    whenNotPaused
{
    require(_amount != 0, "amount = 0");
    Pool storage pool = pools[_poolId];
    pool.stakingToken.safeTransferFrom(_account, address(this), _amount);

    PoolLock memory lock = mapPoolLocks[_poolId][_lockId];

    uint256 stakeAmount = _payTokenFee(_poolId, _lockId, _amount);
    uint256 shares = _buffAmount(_poolId, _lockId, stakeAmount);

    TokenCheque storage cheque = userTokenCheques[_poolId][_lockId][
        _account];
    cheque.poolId = _poolId;
    cheque.lockId = _lockId;
    cheque.shares += shares;
    cheque.stakedBalance += stakeAmount;
    cheque.lastPrintedAt = block.timestamp;

    pool.totalShares += shares;
    pool.totalStaked += stakeAmount;
    if (lock.duration == 0) pool.totalStakedLimited += stakeAmount;

    uint256 stakingTokenDecimals = ERC20(pool.stakingToken).decimals();
    uint256 tvl = (stakeAmount * pool.stPrice) / PRECISION;
    uint256 asgAmount = (tvl * 10 ** 18) / 10 ** stakingTokenDecimals;
@>  uint256 totalASugar = _buffAmount(_poolId, _lockId, asgAmount);

    _mintASugar(_poolId, _lockId, _account, totalASugar);

    emit Staked(_poolId, _account, stakeAmount, _stPrice);
}
```

Impact

Staker can lose aSugar unclaimed rewards if he stakes again before calling `fullyClaimReward()`
.

Recommendation

Calculate the accumulated aSugar of the staker from any previous stakes in `deligateStake()` and mint this amount together with the amount of aSugar tokens that will be minted for the current call of stake.

Team Response

Fixed.

[H-04] Owner Cannot Claim Token Fees

Severity

High Risk

Description

Each lock can have a different `tokenFeeRate` that takes from the staked amount. The owner can claim these amounts via `claimTokenFee()`.

The issue is that lock.tokenFeeBalance gets reset to 0 and then the transfer is made:

```
lock.tokenFeeBalance = 0;
pool.stakingToken.safeTransfer(msg.sender, lock.tokenFeeBalance);
```

As a result, the owner cannot claim the fees.

Location of Affected Code

File: contracts/StakingKo.sol

```
function claimTokenFee(uint256 _poolId) external payable onlyOwner {
    Pool storage pool = pools[_poolId];
    PoolLock[] storage locks = mapPoolLocks[_poolId];

    for (uint256 lId = 0; lId < locks.length; lId++) {
        PoolLock storage lock = locks[lId];
        if (lock.tokenFeeBalance != 0) {
            lock.tokenFeeClaimed += lock.tokenFeeBalance;
@>          lock.tokenFeeBalance = 0;
@>          pool.stakingToken.safeTransfer(msg.sender, lock.
   tokenFeeBalance);
        }
    }
}
```

Impact

The owner cannot claim the staking fees.

Recommendation

Cache tokenFeeBalance and use the cached value for the transfer.

Team Response

Fixed.

[M-01] Fee On Transfer Tokens Are Not Supported

Severity

Medium Risk

Description

When funds are transferred in the contract via some of the functions the amounts that are recorded in storage are according to the specified arguments instead of the actual received amounts. In the case of fee on transfer tokens, these recorded values will not be the correct ones.

Location of Affected Code

File: contracts/StakingKo.sol

```
function createReward( uint256 _poolId, address _rewardToken, uint256
    _rewardAmount, uint256 _duration ) external onlyOwner validPool(
    _poolId) poolIsActive(_poolId) {

function deligateStake( address _account, uint256 _poolId, uint256
    _lockId, uint256 _amount, // aSugar printer uint256 _stPrice, uint256
    _expriedTime, bytes memory _sig ) public payable validPool(_poolId)
    poolIsActive(_poolId) validPoolLock(_poolId, _lockId) updatePoolPrice(
    _poolId, _stPrice, _expriedTime, _sig) updateReward(_poolId, _account)
     payNativeFee whenNotPaused {

function repayment(uint256 _poolId, uint256 _amount) external onlyOwner
    validPool(_poolId) {

function depositRewardAmount( uint256 _poolId, uint256 _rewardId, uint256
     _amount ) external onlyOwner validPool(_poolId) validReward(_poolId,
    _rewardId) {
```

Impact

Last users that try to claim their receipts or rewards might not be able to due to not having enough funds in the contract.

Recommendation

Consider implementing a private function for `transferFrom()` that records the balance of the contract before and after the transfer and returns the actual received amount.

Team Response

Fixed.

[M-02] Users Can Use Outdated Prices As Long As The Signature Expiry Has Not Passed

Severity

Medium Risk

Description

Pool prices influence how much `aSugar` gets minted to stakers when they stake. For this to happen `aSugarSigner` will produce signatures for the staking token prices during each stake and full claim reward.

The message hashes of these signatures are composed of: - stakeToken address - price - expired time

One signature can be used as long as it is not expired which means there can be overlapping between price updates. In cases of high volatility, this could be crucial.

Let's say the expiry time of the signature is 15 minutes, right after this signature is created there could be a drop in price and the following produced signatures are at a price of -50%. Now the user can decide in this 15-minute window which signature to use because both the 1 USD price and the 0.5 USD price will be accepted by the contract as valid.

This way user can manipulate their `aSugar` minting amounts.

Location of Affected Code

File: contracts/StakingKo.sol

```
modifier updatePoolPrice(uint256 _poolId, uint256 _stPrice, uint256
   _expriedTime, bytes memory _sig) {
    require(block.timestamp < _expriedTime, "Expired");
    Pool memory pool = pools[_poolId];
@>  bytes32 message = keccak256(abi.encode(pool.stakingToken, _stPrice,
   _expriedTime));
    address signer = _recoverSigner(message, _sig);
    require(signer == aSugarSigner, "Invalid signer");
    pools[_poolId].stPrice = _stPrice;
    _;
}
```

Impact

Users can use outdated prices as long as the signature expiry has not passed.

Recommendation

Add an ID to the message hash that will be incremented on each price change. On each signature usage, the contract will check if the signature counter is less than the one recorded in contract storage. If it is then revert. This way only the newest price is going to be accepted.

Team Response

Acknowledged.

[M-03] Stakers Accumulate `aSugar` Based On The Spot Price During The Claim

Severity

Medium Risk

Description

Let's say Alice stakes 100e18 of a token for 10 days. During these 10 days, the price of the token was 10 USD. At the end of the 10th day the price doubles and becomes 20 USD per token.

Alice harvests the accumulated `aSugar`. She will get: - tvl = 20 USD * 100 tokens = 2000 USD - `pool.aSugarDailyRoi` = 1% - `aSugarPerDay` = 20 `aSugar` - `aSugarToMint` = 10 days * 20 `aSugar` = 200 `aSugar`

Now let's suppose the opposite case where the price was 10 USD for the whole 10 days but at the end, it dropped to 5 USD. The user will have staked more value over time but he will claim less based on the spot price.

Location of Affected Code

File: contracts/StakingKo.sol

```
function _earnedASugar(uint256 _poolId, uint256 _lockId, address _account
   ) internal view returns (uint256) {
    TokenCheque storage cheque = userTokenCheques[_poolId][_lockId][
       _account];
    Pool memory pool = pools[_poolId];
    if (cheque.stakedBalance == 0 || cheque.lastPrintedAt == 0) return 0;

    uint256 duration = block.timestamp - cheque.lastPrintedAt;
@>  uint256 tvl = (pool.stPrice * cheque.stakedBalance) / PRECISION;

    uint256 earnedASugarPerDay = _buffAmount(_poolId, _lockId, (tvl *
       pool.aSugarDailyRoi) / PRECISION);
    return (earnedASugarPerDay * duration) / ONE_DAY;
}
```

Impact

Users will be minted an unfair amount of `aSugar` tokens despite them locking more/less value over time.

Recommendation

Consider rewarding the users based on the average price of the stake token over their staking period. This information could be collected and generated off-chain and provided to the `fullyClaimReward()` via a signature (if we follow the current pattern of price updates this contract uses).

Team Response

Acknowledged.

[L-01] The `updateReward()` Function Could DOS Or Become Very Expensive

Severity

Low Risk

Description

The `updateReward()` modifier is called in `deligateStake()`, `unstake()`, `fullyClaimReward()`, `_claimReward()`, `notifyRewardAmount()`.
At the moment this modifier loops through all rewards * all locks, reads from storage, performs calculations, and writes to storage. These are gas-intensive operations that scale with the supported rewards and locks. If we have 10 rewards and 5 locks we get 50 loops in total just before the main function is about to get executed.

Location of Affected Code

File: contracts/StakingKo.sol

Impact

As there is no limit to how many reward tokens and locks will be added, this can result in DOS for the functions that use the `updateReward()` modifier due to gas reaching the block gas limit or becoming more expensive.

Recommendation

Consider limiting the amount of locks and rewards in the contract.

Team Response

Acknowledged.

[L-02] Overpayment of Native Fee Results in Stuck Native Funds

Severity

Low Risk

Description

The `payNativeFee()` modifier forces the caller to pay a certain amount of native funds. When an excess amount is sent the execution continues and the excess is no longer reachable for either the owner or the user.

Location of Affected Code

File: contracts/NativeFee.sol

```
modifier payNativeFee() {
    if (nativeFee != 0) {
@>      require(msg.value >= nativeFee, "Invalid native fee");
        nativeFeeBalance += nativeFee;
    }
    _;
}
```

Impact

Stuck excess native funds

Recommendation

Either revert if `msg.value` is not the exact needed quantity or return the excess amount.

Team Response

Fixed.

[I-01] Usage of Transfer Instead of Call For Native Funds

Severity

Informational Risk

Description

The usage of `transfer()` for the transfers of native funds is not recommended since it forwards only 2300 gas to the recipient. If the recipient is a smart contract with a receive function that executes logic there is a high chance to revert and DOS the claim of native fees.

Location of Affected Code

File: contracts/NativeFee.sol

```
function _claimNativeFee() internal {
@>  payable(msg.sender).transfer(nativeFeeBalance);
    nativeFeeClaimed += nativeFeeBalance;
    nativeFeeBalance = 0;
}
```

Impact

Forwarding too little gas.

Recommendation

Consider using `call` instead of `transfer`.

Team Response

Fixed.

, our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!