SALUS SECURITY

JAN 2024

# CODE SECURITY ASSESSMENT

TONKA FINANCE

# Overview

## Project Summary

- Name: Tonka Finance - staking
- Platform: Ethereum
- Language: Solidity
- Repository: https://github.com/Tonka-Finance/Tonka-Contracts
- Audit Scope: See Appendix - 1

# Project Dashboard

## Application Summary

| | |
|---|---|
| **Name** | Tonka Finance - staking |
| **Version** | v1 |
| **Type** | Solidity |
| **Dates** | Jan 05 2024 |
| **Logs** | Jan 05 2024 |

## Vulnerability Summary

| | |
|---|---|
| **Total High-Severity issues** | 4 |
| **Total Medium-Severity issues** | 2 |
| **Total Low-Severity issues** | 4 |
| **Total informational issues** | 2 |
| **Total** | 12 |

## Contact

E-mail: support@salusec.io

## Risk Level Description

| | |
|---|---|
| **High Risk** | The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users. |
| **Medium Risk** | The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact. |
| **Low Risk** | The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances. |
| **Informational** | The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth. |

# Content

# Introduction

## 1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

## 1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

## 1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

# Findings

## 2.1 Summary of Findings

| ID | Title | Severity | Category | Status |
|---|---|---|---|---|
| 1 | Precision loss in updateReward() can be exploited to prevent the contract from generating rewards | High | Business Logic | Pending |
| 2 | Uninitialised variables | High | Business Logic | Pending |
| 3 | CreateStakingPool() records incorrect poolId | High | Business Logic | Pending |
| 4 | Missing update of pointDebt in stake() and withdraw() | High | Business Logic | Pending |
| 5 | CurrentPoolId is not updated in createStakingPool() | Medium | Business Logic | Pending |
| 6 | Users receive less reward when there is insufficient balance in the contract | Medium | Business Logic | Pending |
| 7 | Lack of upper limit for fees in setFees() | Low | Centralization | Pending |
| 8 | Centralization risks | Low | Centralization | Pending |
| 9 | Loss of precision can lead to unexpected situations | Low | Business Logic | Pending |
| 10 | Missing event | Low | Logging | Pending |
| 11 | Non-stakers can call harvest() | Informational | Data Validation | Pending |
| 12 | Users can create a staking pool using malicious tokens | Informational | Access Control | Pending |

## 2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

| **1. Precision loss in updateReward() can be exploited to prevent the contract from generating rewards** | |
|---|---|
| Severity: High | Category: Business Logic |
| Target:<br>- src/staking/DoubleStaking.sol | |

### Description

src/staking/DoubleStaking.sol:L197-L219

```
function updateReward() public {
    // One block only update once
    if (block.timestamp <= lastRewardTime) return;

    // If use balance here, the reward will be inaccurate
    // uint256 balance = IERC20(tokenA).balanceOf(address(this));

    // If no one staked, not calculate reward
    if (totalStakedA == 0) {
        lastRewardTime = block.timestamp;
        return;
    }

    uint256 timePassed = block.timestamp - lastRewardTime;
    uint256 newReward = rewardSpeed * timePassed;
    uint256 newPoints = pointSpeed * timePassed;
    accRewardPerToken += newReward / totalStakedA;
    accRewardPerPoint += newPoints / totalStakedA;

    lastRewardTime = block.timestamp;

    emit RewardUpdated(newReward, newPoints);
}
```

In the highlighted code above, when “newReward” is less than “totalStakedA”, downward rounding of the division occurs. As a result, "accRewardPerToken" increases by zero, indicating that the reward generated from "lastRewardTime" to the current time is zero. The same precision loss issue also applies to the Point rewards.

Since the updateReward() is a public function, a malicious user can exploit the precision loss issue to prevent the contract from generating new rewards.

**Example Attach Scenario**

To prevent users from receiving new rewards, a malicious attacker can exploit the following scenario when rewardSpeed is set to 1:

At timestamp 1, Alice deposits 10 (amountA).

At timestamp 9, the attacker calls updateReward(). At this point, newReward is 9, while totalStakedA is 10. Due to rounding down, accRewardPerToken increases by zero.

The attacker can repeatedly call updateReward() following the above pattern. This prevents the contract from generating new rewards.

## Recommendation

It is recommended that lastRewardTime, accRewardPerToken, and accRewardPerPoint not be updated when accRewardPerToken or accRewardPerPoint increases by zero.

To fix the code, the following is an example:

```
function updateReward() public {
    // One block only update once
    if (block.timestamp <= lastRewardTime) return;

    // If use balance here, the reward will be inaccurate
    // uint256 balance = IERC20(tokenA).balanceOf(address(this));

    // If no one staked, not calculate reward
    if (totalStakedA == 0) {
        lastRewardTime = block.timestamp;
        return;
    }

    uint256 timePassed = block.timestamp - lastRewardTime;
    uint256 newReward = rewardSpeed * timePassed;
    uint256 newPoints = pointSpeed * timePassed;

    uint256 newRewardPerToken = newReward / totalStakedA;
    uint256 newPointsPerToken = newPoints / totalStakedA;
    if (newRewardPerToken == 0 || newPointsPerToken == 0) {
        return;
    }
    accRewardPerToken += newRewardPerToken;
    accRewardPerPoint += newPointsPerToken;


    lastRewardTime = block.timestamp;

    emit RewardUpdated(newReward, newPoints);
}
```

|  |  |
|---|---|
| **2. Uninitialised variables** |  |
| Severity:  High | Category: Business Logic |
| Target:<br>- src/staking/DoubleStaking.sol |  |

## Description

a) The immutable variable pointToken is not assigned a value in the constructor. This causes pointToken related calls to revert.

src/staking/DoubleStaking.sol:L17

```
address public immutable pointToken;
```

b) The pointSpeed variable is not assigned or updated in the code, which will cause the user to never receive the point reward.

src/staking/DoubleStaking.sol:L24

```
uint256 public pointSpeed;
```

## Recommendation

It is recommended to assign the correct value to the variables mentioned.

| 3. CreateStakingPool() records incorrect poolId | |
|---|---|
| Severity: High | Category: Business Logic |
| Target:<br>- contracts/staking/DoubleStakingFactory.sol | |

## Description

src/staking/DoubleStakingFactory.sol:L44-L55

```
function createStakingPool(
    address _tokenA,
    address _tokenB,
    uint256 _stakeRatio,
    address _rewardToken
) external returns (address newStakingPool) {
    require(stakingPools[currentPoolId] == address(0), "Staking pool already exists");

    newStakingPool = address(new DoubleStaking(_tokenA, _tokenB, _stakeRatio,
_rewardToken));
    stakingPools[_stakeRatio] = newStakingPool;
    emit NewPoolCreated(currentPoolId, newStakingPool);
}
```

The createStakingPool() function incorrectly uses _stakeRatio as the id of the newly created pool.

## Recommendation

It is recommended to use the correct poolId - currentPoolId to create the pool correctly.

## 4. Missing update of pointDebt in stake() and withdraw()

| Severity: High | Category: Business Logic |
|---|---|
| Target:<br>- src/staking/DoubleStaking.sol | |

## Description

User.pointDebt is used as the checkpoint variable for received points, and it should be updated with each point reward claim. However, it is no updated in the withdraw() and stake() functions

src/staking/DoubleStaking.sol:L139-L173

```
function withdraw(uint256 _amountA) external {
    if (user.amountA > 0) {
        uint256 pendingReward = (user.amountA * accRewardPerToken) / 1e18 -
user.rewardDebt;
        uint256 actualReward = _safeRewardTransfer(msg.sender, pendingReward);

        uint256 pendingPoints = (user.amountA * accRewardPerPoint) / 1e18 -
user.pointDebt;
        IMintableToken(pointToken).mint(msg.sender, pendingPoints);

        emit Harvest(msg.sender, actualReward, pendingPoints);
    }
    ...
    user.rewardDebt = (user.amountA * accRewardPerToken) / 1e18;
}
```

Since user.pointDebt is not being updated, this results in the user being able to collect more point reward with each call.

## Recommendation

It is recommended that user.pointDebt be updated in the withdraw() and stake() functions.

| 5. CurrentPoolId is not updated in createStakingPool() | |
|---|---|
| Severity: Medium | Category: Business Logic |
| Target:<br>- contracts/staking/DoubleStakingFactory.sol | |

## Description

src/staking/DoubleStaking.sol:L44-L55

```
function createStakingPool(
    address _tokenA,
    address _tokenB,
    uint256 _stakeRatio,
    address _rewardToken
) external returns (address newStakingPool) {
    require(stakingPools[currentPoolId] == address(0), "Staking pool already exists");

    newStakingPool = address(new DoubleStaking(_tokenA, _tokenB, _stakeRatio,
_rewardToken));
    stakingPools[_stakeRatio] = newStakingPool;
    emit NewPoolCreated(currentPoolId, newStakingPool);
}
```

The currentPoolId is not updated in createStakingPool(), which would cause the currentPoolId to always be 1.

Once the pool for PoolId 1 is created, the highlighted check above would prevent any new pool to be created.

## Recommendation

Consider increasing the currentPoolId by one at the end of the function, for example:

```
function createStakingPool(
    address _tokenA,
    address _tokenB,
    uint256 _stakeRatio,
    address _rewardToken
) external returns (address newStakingPool) {
    require(stakingPools[currentPoolId] == address(0), "Staking pool already exists");

    newStakingPool = address(new DoubleStaking(_tokenA, _tokenB, _stakeRatio,
_rewardToken));
    stakingPools[currentPoolId] = newStakingPool;
    emit NewPoolCreated(currentPoolId, newStakingPool);
    currentPoolId += 1;
}
```

| 6. Users receive less reward when there is insufficient balance in the contract | |
|---|---|
| Severity:  Medium | Category: Business Logic |
| Target:<br>- src/staking/DoubleStaking.sol | |

## Description

src/staking/DoubleStaking.sol:L221-L231

```
function _safeRewardTransfer(address _to, uint256 _amount) internal returns (uint256) {
    uint256 balance = IERC20(rewardToken).balanceOf(address(this));

    if (_amount > balance) {
        IERC20(rewardToken).safeTransfer(_to, balance);
        return balance;
    } else {
        IERC20(rewardToken).safeTransfer(_to, _amount);
        return _amount;
    }
}
```

When distributing rewards, if the contract balance is insufficient, rewards will be distributed based on the remaining balance.

When users claim rewards in this situation, they would receive a lower reward than they should. This could surprise the user and potentially harm the project's reputation.

## Recommendation

It is recommended to record undistributed rewards in a queue named pendingReward when the balance is insufficient, and create a new function to handle this queue when the balance is topped up.

| 7. Lack of upper limit for fees in setFees() | |
|---|---|
| Severity: Low | Category: Centralization |
| Target:<br>- src/staking/DoubleStaking.sol | |

## Description

The setFees() function is used to modify the values of stakeFee and withdrawFee.

src/staking/DoubleStaking.sol:L85-L88

```
function setFees(uint256 _stakeFee, uint256 _withdrawFee) external onlyFactory {
    stakeFee = _stakeFee;
    withdrawFee = _withdrawFee;
}
```

However, the function does not have an upper limit for fees. If the fee setter's private key is compromised, the attacker can set stakeFee and withdrawFee to 1000 (i.e., 100%), preventing new users from staking into the system and existing users from withdrawing from the system.

## Recommendation

It is recommended to include a reasonable upper limit for stakeFee and withdrawFee in setFees().

| 8. Centralization risks | |
|---|---|
| Severity: Low | Category: Centralization |
| Target:<br>- contracts/staking/TokaPoints.sol<br>- contracts/staking/DoubleStakingFactory.sol | |

## Description

a) In the TokaPoints contract, there is a privileged owner role.
The owner role has the ability to:

- add/remove minter
- add/remove burner

If the owner's private key is compromised, the attacker can exploit the owner's role to add their address to the minter and burner roles, allowing them to arbitrarily mint tokens and burn tokens owned by anyone else.

b) In the DoubleStakingFactorycontract, there is a privileged owner role.

The owner role has the ability to:

- set rewardSpeed for user created pools
- set fee for user created pools

If the owner's private key is compromised, the attacker can exploit the owner's role to set the rewardSpeed and fee for any pool.

## Recommendation

We recommend transferring the owner roles to multisig accounts with a timelock feature for enhanced security. This ensures that no single person has full control over the account and that any changes must be authorized by multiple parties.

| 9. Loss of precision can lead to unexpected situations | |
|---|---|
| Severity: Low | Category: Business Logic |
| Target:<br>- src/staking/DoubleStaking.sol | |

## Description

src/staking/DoubleStaking.sol:L97-L137

```
function stake(uint256 _amountA) external {
    ...
    // Fixed ratio of tokenA to tokenB
    uint256 amountB = (_amountA * stakeRatio) / 1000;

    IERC20(tokenB).safeTransferFrom(msg.sender, address(this), amountB);
    ...
}
```

a) In the above highlighted code, when _amountA * stakeRatio is less than 1000, amountB = 0, which means that the user will be able to complete the staking without paying tokenB. This may not be intended with the design of doubleStaking.

src/staking/DoubleStaking.sol:L97-L137

```
function stake(uint256 _amountA) external {
    ...
    uint256 amountAWithoutFee = (_amountA * (1000 - stakeFee)) / 1000;
    uint256 amountBWithoutFee = (amountB * (1000 - stakeFee)) / 1000;

    pendingFeeA += (_amountA - amountAWithoutFee);
    pendingFeeB += (amountB - amountBWithoutFee);

    // Record the assets received
    totalStakedA += amountAWithoutFee;
    totalStakedB += amountBWithoutFee;

    user.amountA += amountAWithoutFee;
    user.amountB += amountBWithoutFee;
}
```

b）In the highlighted code, if _amountA * (1000 - stakeFee) is less than 1000, amountAWithoutFee is set to 0. This means that user.amountA will not increase even if the user transfers a non-zero amount of tokenA into the contract. The same issue also applies to the amountBWithoutFee variable and the variables in withdraw().

## Recommendation

It is recommended to revert the call if the mentioned variable (amountB, amountAWithoutFee, amountBWithoutFee) is set to zero due to precision loss.

## 10. Missing event

| Severity: Low | Category: Logging |
|---|---|
| Target:<br>- contracts/staking/TokaPoints.sol | |

### Description

Important parameter or configuration changes should trigger an event to enable off-chain tracking, but functions mentioned below change important parameters without emitting events.

src/staking/TokaPoints.sol:L14-L28

```
function addMinter(address _minter) external onlyOwner {
    isMinter[_minter] = true;
}

function removeMinter(address _minter) external onlyOwner {
    isMinter[_minter] = false;
}

function addBurner(address _burner) external onlyOwner {
    isBurner[_burner] = true;
}

function removeBurner(address _burner) external onlyOwner {
    isBurner[_burner] = false;
}
```

### Recommendation

It is recommended to design corresponding events for these functions to enable off-chain tracking of configuration changes.

## 2.3 Informational Findings

| **11. Non-stakers can call harvest()** | |
|---|---|
| Severity: Informational | Category: Data Validation |
| Target:<br>- src/staking/DoubleStaking.sol | |

### Description

src/staking/DoubleStaking.sol:L97-L137

```
function harvest() external {
    updateReward();

    UserInfo storage user = users[msg.sender];

    uint256 pendingReward = (user.amountA * accRewardPerToken) / 1e18 - user.rewardDebt;
    uint256 actualReward = _safeRewardTransfer(msg.sender, pendingReward);

    uint256 pendingPoints = (user.amountA * accRewardPerPoint) / 1e18 - user.pointDebt;
    IMintableToken(pointToken).mint(msg.sender, pendingPoints);

    user.rewardDebt = (user.amountA * accRewardPerToken) / 1e18;
    user.pointDebt = (user.amountA * accRewardPerPoint) / 1e18;

    emit Harvest(msg.sender, actualReward, pendingPoints);
}
```

The harvest() function allows users to collect their rewards. However, it emits the Harvest event even when user.amountA is zero (i.e., the caller is a non-staker). Emitting a Harvest event for non-stakers may confuse the off-chain indexers.

### Recommendation

It is recommended to revert the call when user.amountA is zero to prevent redundant event emission.

| 12. Users can create a staking pool using malicious tokens | |
|---|---|
| Severity: Informational | Category: Access Control |
| Target:<br>- contracts/staking/DoubleStakingFactory.sol | |

## Description

src/staking/DoubleStakingFactory.sol:L44-L55

```
function createStakingPool(
    address _tokenA,
    address _tokenB,
    uint256 _stakeRatio,
    address _rewardToken
) external returns (address newStakingPool) {
    require(stakingPools[currentPoolId] == address(0), "Staking pool already exists");

    newStakingPool = address(new DoubleStaking(_tokenA, _tokenB, _stakeRatio,
_rewardToken));
    stakingPools[_stakeRatio] = newStakingPool;
    emit NewPoolCreated(currentPoolId, newStakingPool);
}
```

The createStakingPool() function is an external function that can be called by any user. As a result, the malicious users can create pools with malicious tokens.

## Recommendation

It is recommended to configure the createStakingPool() function to an onlyOwner function.

# Appendix

## Appendix 1 - Files in Scope

This audit covered the following files in commit dbe909d:

| File | SHA-1 hash |
|---|---|
| DoubleStaking.sol | a5333fe45d5d5be7d774299c5c2fb512c6deb208 |
| DoubleStakingFactory.sol | ef658fd920a185d7c539e5ad0f26e7f6a769030e |
| TokaPoints.sol | 3501e15ab4d3ee9751817af1d649d74e1eca463e |