BAIL security

BAILSEC.IO

EMAIL : OFFICE@BAILSEC.IO

TWITTER : @BAILSECURITY

TELEGRAM : @HELLOATBAILSEC

FINAL REPORT:

Trustswap StakingPool

January 2024

+StakingPool Extension

February 2024

Disclaimer:

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Trustswap StakingPool
Website	trustswap.com
Type	Staking contract
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/trustswap/teamfinance-contract-stackingpool/blob/e115760ec5bcb40133190047a787c3e598973cc2/src/contracts/StakingPool.sol
Resolution 1	https://github.com/trustswap/teamfinance-contract-stackingpool/blob/4198fece84b9c31f2c4d36a5c6c6fe761873d58a/src/contracts/StakingPool.sol
Resolution 2	https://github.com/trustswap/teamfinance-contract-stackingpool/blob/ca703563dcb2d60f996644de0f94afbd82419db9/src/contracts/StakingPool.sol
Resolution 3	https://github.com/trustswap/teamfinance-contract-stackingpool/blob/61b5907b8298c0c3f5ced71c05b54b2b236f0e8e/src/contracts/StakingPool.sol
Resolution 4	https://github.com/trustswap/teamfinance-contract-stackingpool/blob/bb1a0db8d022432b3ba9a367ba93efb5299f0245/src/contracts/StakingPool.sol
Resolution 5	https://github.com/trustswap/teamfinance-contract-stackingpool/blob/b7c09e9f2eaf2040ed7c4209844cd3b175c0ce48/src/contracts/StakingPool.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	4	4		
Medium	3	2	1	
Low				
Informational	1	1		
Configurational				
Governance	1			1
Quality assurance				
Total	9	7	1	1

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Configurational	Issues which may arise due to different configurational settings
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior
Quality assurance	Aggregated minor issues, ensuring a high quality codebase.

3. Detection

StakingPool

The StakingPool contract is primarily designed to manage the dynamics of token staking and reward distribution.

At its core, the contract enables users to stake tokens in a variety of pools to earn rewards. These rewards are contingent on the amount of tokens staked and the duration of the stake. To facilitate this, the contract encompasses functionalities for depositing and withdrawing tokens, along with the capability for users to claim their accrued rewards.

A key feature of this contract is its democratized approach to pool creation. Any user within the ecosystem can create a new staking pool, specifying critical parameters such as the type of staking and reward tokens, the total reward amount, and the timeframe for the distribution of these rewards, which goes hand in hand with providing the reward tokens.

This flexibility paves the way for a diverse range of staking opportunities, catering to different user preferences and strategies. In terms of pool management, the contract empowers pool owners with the ability to stop their pools using a stopReward function, which ceases the distribution of rewards and permits the owners to reclaim any unallocated rewards, ensuring a fair closure to the pool's lifecycle. Additionally to that, the pool owner can empty any reward tokens when there is no stake in it and the pool has ended.

Moreover, as known from the masterchef, the contract addresses emergency scenarios through an emergencyWithdraw function, allowing users to withdraw their staked tokens promptly, irrespective of the pool's reward status. For the correctness of the reward calculation, the standard masterchef reward logic is incorporated.

Issue	Contract owner can withdraw all tokens
Severity	Governance
Description	The contract owner has a significant governance privilege which allows for withdrawing all tokens in the contract.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, since the contract is behind a proxy, the risk of governance interactions is given anyways.

Issue	Exploit of withdrawEmptyPool allows for stealing all tokens
Severity	High
Description	The withdrawEmptyPool function allows the pool creator to claim any leftover funds after the pool has been disabled, this can be useful in such a scenario where users have withdrawn via emergencyWithdraw, which results in their rewards being stuck in the contract. This allows for stealing all tokens in the contract, let’s simply consider there is an existing USDC pool with a lot of staked USDC, Alice, the hacker wants to drain all USDC in the contract. **PoC:** 1. Alice creates a pool with USDT as staking token and USDC as reward token. 2. Call stopPool, which ends the pool and transfers all provided USDC rewards which are left to the owner. *This step is not mandatory, one could also choose a honeypot

	staking token and a low endTime which ensures that the time passes without anyone actually staking tokens in it.
	
	3. Call withdrawEmptyPool, which then casts the balance of the rewardToken, which is USDC:
	
	uint256 rewardTokenBalance = rewardToken.balanceOf(address(this));
	
	This will be the value in the contract - any staked USDC by any user.
	
	and will transfer it out:
	
	rewardToken.safeTransfer(owner, rewardTokenBalance);
	
	All USDC in the contract has been stolen, this can be executed over and over to drain all tokens in the contract.

Recommendations	Consider removing this function.
Comments / Resolution	Failed resolution, instead of removing the function it has been adjusted. However, the function still allows for stealing all tokens in the contract by bypassing an important check: (endTime > block.timestamp) {revert PoolNotEnded(endTime);} This check reverts if the endTime is larger than block.timestamp, the problem is, it does not revert if endTime == block.timestamp. PoC of Exploit: 1. Alice calls addPool with any token and USDC as reward token 2. Alice calls stopReward which transfers all USDC back to Alice, this function sets endTime = block.timestamp

	3. Alice calls withdrawEmptyPool in the same block: This function will not revert because endTime == block.timestamp. Alice successfully stole the USDC. **Resolution 2:** The function has been removed

Issue	**Lack of support for transfer-tax tokens results in multiple ground-breaking issues**
Severity	High
Description	Anyone can create a new pool via addPool with an arbitrary staking and reward token. First of all it must be noted that the staking token can be the same token as the reward token, which is, in itself no issue, as long as the correct amounts are transferred and accounted for. The problem will now arise when tokens with a transfer-tax are being added, if a pool creator now adds a token with transfer-tax as staking and reward token, the transferred reward amount: rewardTokenInterface.safeTransferFrom(msg.sender, address(this), totalReward); will be insufficient and will not cover the balance, now if users deposit tokens, at some point rewards will be taken from the staked tokens. Additionally it goes without saying that also during deposits users get more accounted for than the contract receives, this will not only have the impact that the last withdrawer will not ger any tokens but will also break the reward mechanism. Rebase tokens are not supported and any rebalance gain will not be reflected for users. —---

APPENDIX: Transfer-tax tokens and reward manipulation

Interestingly, we could observe that the contract uses the following accounting for the totalStaked value:

pool.totalStaked += _amount;

Whereas the amount is based on the input value of the user. This totalStaked variable is then used as divisor for the reward accumulation:

uint256 lpSupply = pool.totalStaked;

pool.accTokenPerShare = pool.accTokenPerShare + (rewards * pool.precision) / lpSupply;

Which is perfectly fine. However, certain masterchefs do not implement the accounting of this value but simply use the contract balance of the specific erc20 token as divisor, which, in itself is just a low severity issue since users can directly transfer tokens into the contract which would then decrease the rewards.

But things become interesting if the balance is used as divisor while the transfer-tax issue is present.
This can get exploited by a malicious user as follows:

1. Deposit 100e18 tokens, which is directly assigned to the user.amount with wallet X

2. Deposit with another wallet tokens and withdraw these, multiple times. This will with each step decrease the contract balance because the contract receives less tokens but the user can withdraw the full amount.

3. This executed multiple times will decrease the ERC20 balance of the said token and result in a very low value, however, remember, in wallet X the user.amount is still 100.

What happens now? During a pool update, the rewards are divided by the ERC20 balance, which is lets say 1, but during a claim, the

	user.amount is multiplied with the accumulate reward value, which was divided by 1 and is now multiplied by 100 due user.amount being 100. This will not artificially inflate rewards and allows for more tokens being claimed than initially desired as rewards. In this contract this could've been exploited by creating pools with reward tokens that are also staking tokens for other pools and then execute this practice to steal all staking tokens. Fortunately, this is not possible due to the correct accounting of the lp supply. **With this example we want to highlight how important the correct accounting is and are proud that the client already implements the correct accounting pattern, which proves that thorough research was conducted during the development cycle of the contract.**
Recommendations	Consider simply following the before-after pattern. A reentrancy guard is mandatory.
Comments / Resolution	Resolved, the before/after pattern has been implemented for the following spots: addPool deposit It is important to call initializePoolV2 directly after initialization to ensure that no one can reenter in this function to disable the reentrancy guard to then reenter in another function which is now unguarded.

Issue	Flaw in emergencyWithdraw allows to steal all tokens from first pool
Severity	High

Description	The emergencyWithdraw function allows users to withdraw tokens without caring about their rewards. However, there is a critical flaw implemented which allows for stealing tokens from the pool with index0:
	
	`PoolInfo storage pool = poolInfo[0];`
	`UserInfo storage user = userInfo[msg.sender][poolId];`
	
	The flaw lies within the fact that the poolInfo is from index0 instead of poolId
	
	This allows an attacker to create a pool, deposit tokens and then call emergencyWithdraw, due to the flaw in the emergencyWithdraw, the stakingToken from pool 0 is drained:
	
	`pool.stakingToken.safeTransfer(address(msg.sender), amount);`
	
	while the user.amount parameter is from our own poolId:
	
	`UserInfo storage user = userInfo[msg.sender][poolId];`
	`uint256 amount = user.amount;`
	
	This allows for stealing all tokens from the pool with index0

Recommendations	Consider using the correct poolId for the PoolInfo mapping.
Comments / Resolution	Resolved.

Issue	Redundant user flexibility
Severity	Medium
Description	Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider taking the following actions in an effort to limit user flexibility: 1. Ensure the timeframe for the addPool function between startTime and endTime is not unreasonably one, one could argue about 1 or maximum 2 years. 2. Ensure the provided decimals for the addPool function are between 12 and 36 3. Ensure the poolId for the deposit function is existent. 4. Ensure the poolId for the withdraw function is existent. 5. Ensure the poolId for the claimReward function is existent. 6. Ensure the poolId for the emergencyWithdraw function is existent. 7. Ideally, a whitelist is added which allows the contract owner to whitelist which tokens can be used as reward and which as staking token. 8. Implement a minimum grace period between adding and stopping pools, such as 1 day as example.

	9. Ensure _pid for updatePool is existent. 10. Ensure relation between startTime and endTime is sufficiently validated, ie. startTime < endTime and a sufficient minimum period is required. In the scenario of startTime = endTime, this will lock all rewards in the contract due to a **division by zero** revert.
Comments / Resolution	Partially resolved, the following checks have been implemented: 1. Start and endTime has been validated to ensure startTime is smaller than endTime and the maximum duration has been limited to 1825 days. 2. The precision is validated to be between 6 and 36. 3. PoolID parameters are now validated for all important functions. 4. Logic has been added which prevents from calling stopReward whenever the pool has started and the overall duration is less than 3600 seconds. However, in our opinion this logic does not really make sense, consider just removing it. The initial idea was to prevent immediate adding and stopping of rewards to counter potential issues which might occur due to repetitive calls of these functions.

Issue	Arbitrary staking and reward tokens may allow for reentrancy occasions
Severity	Medium
Description	Within the addPool function, there is no validation for the provided staking and reward tokens. A user can simply provide any custom token with any logic, including reentrancy. This might result into reentrancy vulnerabilities throughout the codebase.

Recommendations	Consider implementing a reentrancy guard on every external function.
Comments / Resolution	Resolved, a reentrancyGuard has been implemented on: addPool stopReward deposit withdraw claimReward withdrawEmptyPool It is important to call initializePoolV2 directly after deployment to ensure that no one can reenter in this function to disable the reentrancy guard to then reenter in another function which is now unguarded.

Issue	**Typographical Issues**
Severity	**Informational**

Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below:
	
	L 56:
	
	error PoolNotEnded(uint256 endTime);
	
	This error is unused.
	
	L 58
	
	error WithdrawAmountTooBig();

	This error is unused. L 62: The contract should disable the initializer method on the implementation.
Recommendations	
Comments / Resolution	Resolved

StakingPool - Extension

The client provided us with an updated commit, with the task of auditing the changes compared to the last audited commit.

Last audited commit:

https://github.com/trustswap/teamfinance-contract-stackingpool/blob/b7c09e9f2eaf2040ed7c4209844cd3b175c0ce48/src/contracts/StakingPool.sol

New provided commit:

https://github.com/trustswap/teamfinance-contract-stackingpool/blob/88ef81f269544036444090e76b0f7b89daa350d5/src/contracts/StakingPool.sol

Diffchecker link:

https://www.diffchecker.com/FUuTwMJK/

The updated commit incorporates the following changes:

1. poolStakeLimit(poolId => limit) mapping: Allows the pool owner to set a stakeLimit for deposit, this limit is enforced in the deposit function.

2. rewardCredit(address => poolId => pending) mapping: Determines the pending rewards for a user from a specific poolId, is increased whenever a deposit happens by the pending amount.

3. pendingReward function: Adjusted to attach the rewardCredit (pending).

4. deposit function: Incorporates the mentioned poolStakeLimit check. Increases rewardCredit by the current pending amount (user.amount * accRewardPerShare / precision) - rewardDebt.

5. withdraw function: Ensures users can only withdraw *after* pool.endTime has passed. Includes the rewardCredit into the pending calculation.

6. claim function: Ensures users can only claim *after* pool.endTime has passed. Includes the rewardCredit into the pending calculation.

7. setPoolStakeLimit: Allows a pool owner to set a valid stakeLimit.

8. emergencyWithdraw function has been removed.

Updated commit with fixes:

https://github.com/trustswap/teamfinance-contract-stackingpool/blob/49a3005d5c03c8d000e48b9d97a17d01764c2c0f/src/contracts/StakingPool.sol

Issue	Override of rewardCredit during deposit will result in permanently lost rewards for users
Severity	High
Description	Whenever a deposit happens, the possibility exists that a pending claimable reward amount is existent, if the position already has a balance. Therefore, all subsequent deposits must correctly handle any accumulated rewards, which is done by aggregating them into the rewardCredit mapping: rewardCredit[msg.sender][poolId] = (user.amount * pool.accTokenPerShare) / pool.precision - user.rewardDebt; The problem here lies within the fact that the previous rewardCredit is overridden instead of increased. This will result in a loss of all previous rewards.
Recommendations	Consider increasing instead of overriding the rewardCredit. Moreover, this issue indicates that the new contract is insufficiently tested, ie. lack of tests for subsequent deposits by the same user. It is mandatory to implement edge-case tests for this mechanism.
Comments / Resolution	Resolved.

Issue	Non-existent emergencyWithdraw function
Severity	Medium
Description	The contract does not implement the emergencyWithdraw function. In any scenario where there is a flaw in the reward update or insufficient rewards, the withdrawal will revert, resulting in funds being permanently stuck.

Recommendations	Consider implementing an emergencyWithdraw function, which is callable after pool end.
Comments / Resolution	Resolved