#HashEx
BLOCKCHAIN SECURITY

WannaStart

smart contracts final audit report

January 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the WannaSwap team to perform an audit of their WannaStart contract. The audit was conducted between January 19 and January 21, 2021.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at the WannaStart GitHub repository. The code was audited after the commit fd0b00a.

Update: A recheck was made on commit 21b3997. The same code was deployed to the Aurora network at the address 0x72dC6953F48b44488BAB73Ec3EA3BcBefd119A0A.

2.1 Summary

Project name	WannaStart
URL	https://wannaswap.finance/
Platform	Aurora
Language	Solidity

2.2 Contracts

Name	Address
WannaStart	0x72dC6953F48b44488BAB73Ec3EA3BcBefd119A0A

3. Found issues

High	2 (22%)
Medium	1 (11%)
Low	2 (22%)
Info	4 (45%)

C7b. WannaStart

ID	Severity	Title	Status
C7bI90	High	Sale parameters can be changed after sale is started	Resolved
C7bI92	High	Pool can be finalized several times	Resolved
C7bI91	Medium	Wrong token balance check	Resolved
C7bI8d	Low	Lack of tests	Acknowledged
C7bI8c	Low	Lack of checks on the input parameters	Resolved
C7bI8b	Info	Unused variable	Resolved
C7bI8e	Info	Lack of NatSpec documentation	Acknowledged
C7bI8f	Info	Supported token types restrictions	Acknowledged
C7bI93	Info	Require statement in a view-function	Resolved

4. Contracts

C7b. WannaStart

Overview

The WannaStart contract is a decentralized launchpad. There are 4 stages:

1. Users deposit purchase tokens (named LP tokens in the contract). Based on the time-weighted value of the deposited token maximum commitment value is calculated.

2. Users deposit purchase tokens to the contract, the value that can be deposited is capped by the maximum commitment value.

3. After end time users claim their reward token and get a refund if more tokens than max cap were committed.

Issues

C7bI90 Sale parameters can be changed after sale is started

High Resolved

The owner can break contract calculations by updating the pool after the start time. Moreover, the owner can block all withdrawals by setting the `startTime` parameter far in the future.

```
function withdraw(uint _pid, uint _amount) external nonReentrant {
    require(_pid < poolInfo.length, "withdraw: BAD POOL");
    PoolInfo storage pool = poolInfo[_pid];
    require(block.timestamp >= pool.startTime, "withdraw: NOT NOW");
    ...
}
```

Recommendation

Disable pool updates after `startTime`.

```
    function setPool(uint _pid, address _lpToken, address _token, address _mustHoldToken,
uint _totalAmount, uint _mustHoldAmount, uint _totalLp, uint _startTime, uint _commitTime,
uint _endTime, uint _claimTime) external onlyOwner {
        require(_pid < poolInfo.length, "setPool: BAD POOL");
        require(poolInfo[_pid].startTime > block.timestamp, "setPool: RESTRICTED");
        ...
    }
```

C7bI92 Pool can be finalized several times

High Resolved

The function `finalizePool()` sends `totalRaised` amount to the `feeTo` and `_fundTo` addresses. The value of the totalRaised variable is supposed to be not more than the sale cap (the totalLp variable). This check is done in L299:

```
        uint totalRaised = totalCommitment > totalLp ? totalLp : totalCommitment;
        uint balance = lpToken.balanceOf(address(this));
        if (totalRaised > balance) totalRaised = balance;
        uint totalFee = totalRaised.mul(fee).div(100e18);
        uint amount = totalRaised.sub(totalFee);
        // send fee to converter
        lpToken.safeTransfer(feeTo, totalFee);
        // send fund to offerer
        lpToken.safeTransfer(_fundTo, amount);
```

The excess of `totalCommitment` is claimable by users via a refund mechanism. However if the function `finalizePool()` is called twice with the same ID, it sends the excess to the `feeTo` and `_fundTo` addresses and users won't be able to get refunds.

By this mechanism, all the deposits and commitments that haven't been withdrawn can be withdrawn by the project owner.

Recommendation

Add a check that the `finalizePool()` can be called only once for a pool.

Update on commit 6cab18b

This function now can be called only once. But because of the reentrancy vulnerability in the first call there can be multiple internal calls of this function. It is recommended to use ReentrancyGuard on this function.

C7bI91 Wrong token balance check

Medium Resolved

The function claim() in L281 checks the balance of the `token`, but sends the `lpToken`.

```
function claim(uint _pid) external nonReentrant {
    ...
    if(pendingRefund > 0) {
        uint balanceRefund = token.balanceOf(address(this));
        if (pendingRefund > balanceRefund) {
            pendingRefund = balanceRefund;
        }

        user.claimedRefundAmount = user.claimedRefundAmount.add(pendingRefund);
        IERC20(pool.lpToken).safeTransfer(address(msg.sender), pendingRefund);
    }
    emit Claim(msg.sender, _pid, pending, pendingRefund);
}
```

C7bI8d Lack of tests

Low Acknowledged

There are no tests in the contract repo. We strongly recommend having a full unit-test coverage to ensure that the contracts work as expected and minimize the possibility of bugs.

C7bI8c Lack of checks on the input parameters

Low Resolved

The function `setFeeTo()` lacks a zero-check on the `_feeTo` parameter. If it is mistakenly initialized with a zero address the funds sent to `_feeTo` maybe burnt.

```
    function setFeeTo(address _feeTo) external onlyOwner {
         feeTo = _feeTo;

         emit SetFeeTo(_feeTo);
     }
```

C7bI8b Unused variable

Info Resolved

The variable `_mustHoldAmount` declared in L107 is not used.

```
function setPool(uint _pid, address _lpToken, address _token, address _mustHoldToken, uint
_totalAmount, uint _mustHoldAmount, uint _totalLp, uint _startTime, uint _commitTime, uint
_endTime, uint _claimTime) external onlyOwner {
  ...
}
```

Recommendation

Update the mustHoldAmount in the pool or remove the parameter from the function.

C7bI8e Lack of NatSpec documentation

Info Acknowledged

There is no NatSpec documentation in the audited contract. We recommend adding documentation on all public and external functions to provide a better user experience for users interacting with the smart contract.

C7bI8f Supported token types restrictions

Info Acknowledged

The contract does not support deflationary tokens or tokes with rebase. Also no `lpToken` in the pool can be equal to the offering `token` as it will break calculations.

C7bI93 Require statement in a view-function

Info Resolved

The function `maxCommitment()` has a view-modifier and has a `require` statement inside. In case of a requirement failure, the node on an RPC call may not return failure but an arbitrary value.

Recommendation

Refactor the function to return 0 on a wrong `_pid` parameter.

5. Conclusion

2 high and 1 medium severity issues were found.

This audit includes recommendations on the code improving and preventing potential attacks.

Update: issues that could potentially allow the owner to withdraw users' funds are fixed in the update.

Appendix A. Issues severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Info.** Issues that do not impact the contract operation. Usually, info severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY