#HashEx
BLOCKCHAIN SECURITY

JETI Launchpad

smart contracts final audit report

December 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 JETI Launchpad team to perform an audit of their smart contract. The audit was conducted between 18/10/2022 and 23/10/2022.

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 @JETI-ONE/Contracts GitHub repository after the 3f69597 commit.

Update: the JETI Launchpad team has responded to this report. The updated code is located in the same GitHub repository after the 192ccc0 commit.

It should be noted that the audit was done on the contracts' code published to Github and it does not guarantee that the deployed version will be the same. Users must check the code of the deployed contracts they are interacting with.

2.1 Summary

Project name	JETI Launchpad
URL	https://services.jeti.one
Platform	Binance Smart Chain
Language	Solidity

2.2 Contracts

Name	Address
Incentive.sol	
Sale.sol	
SaleVault.sol	
IncentiveFactory.sol	
SaleFactory.sol	
SaleMaster.sol	
VaultFactory.sol	
Blacklist.sol	
Coupon.sol	
JETIToken.sol	
Multiple contracts	

3. Found issues

High	10 (31%)
Medium	8 (25%)
Low	9 (28%)
Info	5 (16%)

C1. Incentive.sol

ID	Severity	Title	Status
C1-01	Low	Gas optimization	Resolved
C1-02	Info	On-chain random function	Acknowledged

C2. Sale.sol

ID	Severity	Title	Status
C2-01	High	Users may not receive their tokens	Resolved
C2-02	High	Transfer is out of else-if clause	Resolved
C2-03	Low	Gas optimization	Resolved

C3. SaleVault.sol

ID	Severity	Title	Status
C3-01	High	Exaggerated owners rights	Resolved
C3-02	High	createLock() call will fail with a high probability if pair exists	Resolved
C3-03	Low	Gas optimization	Resolved

C4. IncentiveFactory.sol

ID	Severity	Title	Status
C4-01	Low	Gas optimization	Resolved

C5. SaleFactory.sol

ID	Severity	Title	Status
C5-01	Low	Gas optimization	Resolved

C6. SaleMaster.sol

ID	Severity	Title	Status
C6-01	High	The contract aggregates BNB	Resolved

C7. VaultFactory.sol

ID	Severity	Title	Status
C7-01	Low	Gas optimization	Resolved

C8. Blacklist.sol

ID	Severity	Title	Status
C8-01	High	'memory' is used instead of 'storage'	Resolved
C8-02	Medium	Permanent ban logic inconsistency	Resolved
C8-03	Medium	Unavailable function for blocked users	Resolved
C8-04	Low	View function doesn't work	Resolved
C8-05	Low	Gas optimization	Resolved

C9. Coupon.sol

ID	Severity	Title	Status
C9-01	High	'memory' is used instead of 'storage'	Resolved
C9-02	Medium	Requirement will always be satisfied	Resolved
C9-03	Medium	Reversed requirement	Resolved
C9-04	Low	Gas optimization	Resolved
C9-05	Info	Owner can buy coupon for free	Resolved

C10. JETIToken.sol

ID	Severity	Title	Status
C10-01	High	Fees may reach 100%	Resolved
C10-02	High	Exaggerated owners rights	Resolved
C10-03	Medium	Gas optimization	Resolved
C10-04	Medium	Wrong variable in blacklist check	Resolved

C11. Multiple contracts

ID	Severity	Title	Status
C11-01	High	Missing tests	Resolved
C11-02	Medium	Few events	Resolved
C11-03	Medium	Lack of validation of input parameters	Resolved
C11-04	Info	Floating Pragma	Resolved
C11-05	Info	No support of tokens with commissions	Acknowledged
C11-06	Info	Swaps don't have min amount and deadlines	Resolved

4. Contracts

C1. Incentive.sol

Overview

A lottery contract for sale participants. Prizes are drawn during the sale round finalisation. Depending on the lottery setting prize may be in a native currency or selling tokens, the prize amount is also customizable and individual for each sale.

Issues

C1-01 Gas optimization

Low Resolved

a. The state variable `incentiveFactoryAddress`, `token`, `_incentiveFactory` can be declared as `immutable`;

b. `saleSetting` variable is redundunt.

C1-02 On-chain random function

Info Acknowledged

The result of the `random()` function call can be predicted in advance. Winner ballot choosing algorithm certainty may be abused by the owner or the admin to win in a lottery.

Recommendation

It is recommended to use blockchain oracles such as Chainlink to generate random numbers.

C2. Sale.sol

Overview

The core launchpad contract with fundraising and sale token sharing functionality. The sale consists of three rounds maximum. If a round finished with an unsuccessful status it can be restarted. After round finalization, raised funds and liquidity sale token part are sent to the vault contract, where they are used for liquidity adding to the Uniswap pair, obtained liquidity tokens are locked in the vault for a set period of time.

Issues

C2-01 Users may not receive their tokens

High Resolved

The `setSaleCancel()` function gifts the owner an extraordinary right to cancel finalized round. Assuming the situation a sale reaches hardcap, no deposits are admitted anymore. Some of the users finalize a round with `finalizeSale()`, accumulated BNB is sent to the pair and liquidity tokens are locked in the vault. As soon as this happens, the owner sets `sale_details.canceled` to 1 and call `userWithdrawBaseTokens()`. All left sale tokens are returned to the owner and as liquidity is already on the pair he can sell them for BNB, users lose funds and don't get promised tokens.

Recommendation

Disallow `setSaleCancel()` for finalized rounds.

C2-02 Transfer is out of else-if clause

High Resolved

In L525 left BNB are sent to `msg.sender` after rounds of fundraising, fees payout, and prize transfer to incentive contract. This transfer should take place only when `_incentive.getType(_round) == 2`, but not during finalization of each round. As a result, sale rounds with `_incentive` type equal to 1 can't be succeeded as the transaction will be reverted due to insufficient BNB balance.

```
function finalizeSale (uint8 _round) public nonReentrant IsSaleEnded(_round) {
    ...

    uint256 feeAmount = ((sale_status[_round].sale_raised_amount * (10000 -
sale_info[_round].mult)) / 10000);
    uint256 liquidityAmount = (sale_status[_round].sale_raised_amount *
sale_info[_round].mult / 10000 * sale_info[_round].liquidity / 100);
    uint256 liquidityToken = (sale_status[_round].sale_raised_amount *
sale_info[_round].mult / 10000 * sale_details.listRate / convert *
sale_info[_round].liquidity / 100);

    // Fee collection
    if(feeAmount > 0) {
        payable(feeAddress).transfer(feeAmount);
    }

    ...

    // Liquidity lock
    if(liquidityAmount > 0) {
        ERC20(sale_details.sale_token).transfer(address(saleVaultAddress),
liquidityToken);
        _saleVault.createLock{value:liquidityAmount}(_round, sale_details.sale_token,
liquidityToken, liquidityAmount, sale_info[_round].end, sale_info[_round].end +
(sale_info[_round].liquidityLock * 1 minutes));
    }

    if(_incentive.getType(_round) == 1) {
        ...
        payable(msg.sender).transfer(sale_status[_round].sale_raised_amount - feeAmount -
liquidityAmount);
    } else if(_incentive.getType(_round) == 2) {
        _incentive.calcPrize(_round, sale_status[_round].sale_raised_amount);
        _incentive.findWinnerBase{value:_incentive.getPrizeAmount(_round)}(_round);
    }

    payable(msg.sender).transfer(sale_status[_round].sale_raised_amount - feeAmount -
liquidityAmount - _incentive.getPrizeAmount(_round));

    setSaleFinalized(_round);
}
```

Recommendation

Replace transfer to else-if clause.

C2-03 Gas optimization

Low Resolved

a. `saleFactoryAddress`, `_saleFactory` should be immutable or even deleted;

b. `saleVaultAddress`, `_saleVault`, `incentiveAddress`, `_incentive`, `convert`, `launchFee` should me immutable;

c. L365 can be changed to `sale_sold_amount` to avoid the multiplication step.

C3. SaleVault.sol

Overview

To the contract are sent raised funds of each successfully ended sale round along with specially allocated sale token part. Received funds are added as liquidity to the Uniswap pair, if the pair doesn't exist it is created. Liquidity tokens are locked on the contract for a period set in the sale contract. After the lock period expires, liquidity tokens can be withdrawn by the project owner.

Issues

C3-01 Exaggerated owners rights

High Resolved

a. The owner can change the router address to a malicious one using the `setRouterContract()` function, then all raised funds can be sent directly to the owner bypassing the liquidity lock period;

b. `saleMasterAddress` can change `saleContract` using `setSaleContract()` function what will break liquidity transfer from Sale contract.

Recommendation

a. Make the `router` constant or redesign the `setRouterContract()` method;

b. Make `saleContract` constant and remove the `setSaleContract()` method.

C3-02 createLock() call will fail with a high probability if pair exists

High Resolved

During liquidity addition, tokens and BNB surpluses are returned back. Since the contract doesn't have receive functionality such transactions will be most likely reverted leading to round finalization failure.

```
function createLock(uint8 _round, address _token, uint256 _token_amount, uint256
_weth_amount, uint256 _start, uint256 _end) public nonReentrant payable {
    ...
    LiquidityLockDetails storage lock = _liquidityLockDetails[_round];

    address _pair = IPancakeFactory(IPancakeRouter(router).factory()).getPair(_token,
IPancakeRouter(router).WETH());

    if(_pair == address(0)) {
        _pair = IPancakeFactory(IPancakeRouter(router).factory()).createPair(_token,
IPancakeRouter(router).WETH());
    }

    ERC20(token).approve(address(router), _token_amount);

    (uint amountToken, uint amountETH, uint liquidity) =
IPancakeRouter(router).addLiquidityETH{value:_weth_amount}
(token,_token_amount,_token_amount,_weth_amount,address(this),(block.timestamp + delay));
    ...
}
```

Also returned BNB and tokens may be locked on the contract, since there is no function for balances withdrawing.

Recommendation

Add `recieve()` function and methods to withdraw BNB and tokens from the contact.

C3-03 Gas optimization

Low Resolved

a. The state variable `token`, `vaultFactoryAddress`, `_vaultFactory` can be declared as `immutable` to save gas.

b. L82 unnecessary `owner` state variable setting.

C4. IncentiveFactory.sol

Overview

Factory of Incentive contracts.

Issues

C4-01 Gas optimization

Low Resolved

The functionality of the `Address`, `ERC20` imports is not used in this contract.

C5. SaleFactory.sol

Overview

Factory of Sale contracts.

Issues

C5-01 Gas optimization

Low Resolved

The functionality of the `Address`, `ERC20` imports is not used in this contract.

C6. SaleMaster.sol

Overview

The contract is the entry point for creators willing to launch their projects on the platform. The parameters regulating each of the three sale rounds are passed to the sale creation method, where they are propagated to Sale, Incentive, and SaleVault factories resulting in the creation of separate interconnected parts of one sale. For a sale creation, a fee in BNB is charged. The total fee depends on the royalty percentage applied to raised funds and active coupons with discounts.

Issues

C6-01 The contract aggregates BNB

High Resolved

If a user has an active coupon, he can get a discount on sale creation. However, discount existence is determined only in the process of method execution and the user has to attach the full price. If there is one, `feeAddress` is sent `launchFee[_sale_details[0]])*(100-((_sale_details[5]))/100`, whereas overpayment stays on the contract without withdraw possibility.

```
function _createSale(
  address _sale_token,
  uint256[9] memory _sale_details,
  uint256[16] memory _seed_details,
  uint32[5] memory _seed_vesting,
  uint256[16] memory _presale_details,
```

```
    uint32[5] memory _presale_vesting,
    uint256[16] memory _community_details,
    uint32[5] memory _community_vesting
) external payable returns (address[3] memory createAddresses) {
    require(_blacklist.getBlacklist(msg.sender, address(this)) == 0, "Blacklisted");
    require(msg.value >= launchFee[_sale_details[0]], "Flat fee");
    require(tokenStatus[_sale_token] == 0, "Already listed");

    refundExcessiveFee(_sale_details[0]);
    ...
    if (_sale_details[8] > 0 && _coupon.getCouponActive(msg.sender,_sale_details[8]) == 1)
{
        payable(feeAddress).transfer((launchFee[_sale_details[0]])*(100-
((_sale_details[5]))/100));
    } else {
        payable(feeAddress).transfer(launchFee[_sale_details[0]]);
    }

    if(_sale_details[8] > 0 && _coupon.getCouponActive(msg.sender,_sale_details[8]) == 1)
{
        _coupon.useCoupon(msg.sender,_sale_details[8]);
    }
    ...
}
```

Recommendation

Return the remaining BNB to a user.

C7. VaultFactory.sol

Overview

Factory of Vault contracts.

Issues

C7-01 Gas optimization

Low Resolved

The functionality of the `Address`, `ERC20` imports is not used in this contract.

C8. Blacklist.sol

Overview

An auxiliary contract that has the functionality to block and unblock users.

Issues

C8-01 'memory' is used instead of 'storage'

High Resolved

In the `setBlacklist()` memory, the variable `blacklist` is used instead of the storage variable (L50). Because of this, all changes made to this variable are not saved, which leads to incorrect work of these functions.

```
function setBlacklist(uint256 _blacklistTime, address _address, address _contract, uint256
_type) public {
    require((msg.sender == blacklist_admins._blacklistAdmins)&&(_contract ==
blacklist_admins._blacklistContracts), "You cannot blacklist!");
    BlacklistInfo memory blacklist = _blacklistInfo[_address][_contract]; // memory
instead of storage! error!
    uint256 blacklistStart = block.timestamp;

    blacklist.blacklistStart=blacklistStart;
    blacklist.blacklistEnd=(_blacklistTime * 1 minutes) + blacklistStart;

    if(_type == 0) {
        blacklist.blacklist_type = BlacklistType.WARNING;
        _blacklistCount[_address]++;
        blacklist._isBlacklisted = 1;
    } else if(_type == 1) {
```

```
        blacklist.blacklist_type = BlacklistType.PROBATION;
        _blacklistCount[_address]++;
        blacklist._isBlacklisted = 1;
    } else if(_type == 2) {
        blacklist.blacklist_type = BlacklistType.PERMANENT;
        _blacklistCount[_address]++;
        blacklist._isBlacklisted = 1;
    } else if(_type == 3) {
        blacklist.blacklist_type = BlacklistType.FREE;
        blacklist.blacklistStart=0;
        blacklist.blacklistEnd=0;
        blacklist._isBlacklisted = 0;
    }
}
```

Recommendation

It is necessary to change the work with these variables by changing the modifier to `storage`. Be careful, as you can create non-optimization for gas if you frequently access storage variables.

C8-02 Permanent ban logic inconsistency

Medium Resolved

When a user's `blacklistCount` reaches 3 and more, a user automatically gets banned on a corresponding contract. From the naming it follows permanent ban is constant and can not be revoked, however, it still has an expiration time and can be canceled with `resetBlacklist()` when `blacklistEnd` the timestamp is passed. On the other hand, the admin doesn't have a right to cancel someone's permanent ban as the user blacklist adjustment will end in the PERMANENT else-if section during a `setBlacklist()` call.

```
function setBlacklist(uint256 _blacklistTime, address _address, address _contract, uint256
_type) public {
    ...
    if(_type == 0 || _type == 1 || _type == 2) {
        blacklist.blacklistCount++;
    }

    if(_type == 0 && blacklist.blacklistCount < 3) {
        blacklist.blacklist_type = BlacklistType.WARNING;
```

```
            blacklist._isBlacklisted = 1;
        } else if(_type == 1 && blacklist.blacklistCount < 3) {
            blacklist.blacklist_type = BlacklistType.PROBATION;
            blacklist._isBlacklisted = 1;
        } else if(_type == 2 || blacklist.blacklistCount >= 3) {
            blacklist.blacklist_type = BlacklistType.PERMANENT;
            blacklist._isBlacklisted = 1;
        } else if(_type == 3) {
            blacklist.blacklist_type = BlacklistType.FREE;
            blacklist.blacklistStart=0;
            blacklist.blacklistEnd=0;
            blacklist._isBlacklisted = 0;
        }

        emit AddBlacklist(_blacklistTime, _address, _contract, _type);
    }

    function resetBlacklist(address _address, address _contract) public {
        BlacklistInfo storage blacklist = _blacklistInfo[_address][_contract];

        uint256 remainTime = blacklist.blacklistEnd > block.timestamp ? blacklist.blacklistEnd-
    block.timestamp : 0;

        if (remainTime <= 0) {
            blacklist.blacklist_type = BlacklistType.FREE;
            blacklist.blacklistStart=0;
            blacklist.blacklistEnd=0;
            blacklist._isBlacklisted = 0;
        }
    }
```

Recommendation

Disallow `resetBlacklist()` for PERMANENT bans or redesign blacklisting logic, an admin could cancel PERMANENT ban to remove ambiguity.

C8-03 Unavailable function for blocked users

Medium Resolved

The `blacklistTimeLeft()` function unblocks a user after the timeout expires, but it will throw an error when called, because when called the `setBlacklist()` function, the user will fail the

check on L49 and remain banned.

```
function setBlacklist(uint256 _blacklistTime, address _address, address _contract, uint256
_type) public {
    require((msg.sender == blacklist_admins._blacklistAdmins)&&(_contract ==
blacklist_admins._blacklistContracts), "You cannot blacklist!"); // error!
    ...
}

function blacklistTimeLeft(address _address, address _contract) public returns(uint256) {
    uint256 remainTime = _blacklistInfo[_address][_contract].blacklistEnd >
block.timestamp ? _blacklistInfo[_address][_contract].blacklistEnd-block.timestamp : 0;
    if (remainTime == 0) {
        setBlacklist(0, _address, _contract, 3);
    }
    return remainTime;
}
```

Recommendation

Add either a separate function to change the blocking status for blocked users or rewrite the `setBlacklist()` function.

C8-04 View function doesn't work

Low Resolved

The `permaBan()` function should change or check `blacklist_info.blacklist_type` (it is not clear what the function should do), but it does not perform these actions.

```
function _permaBan() public view {
    if (blacklist_info._blacklistCount > 3) {
        blacklist_info.blacklist_type == BlacklistType.PERMANENT;
    }
}
```

C8-05 Gas optimization

Low Resolved

a. `_addBlacklister()` has to be called each time a setting contract differs from active in `blacklist_admins`;

b. `isBlacklisted()`, `blacklistTimeIsOVer()` modifiers are not used;

c. Unused storage variable `seeBlacklist`.

C9. Coupon.sol

Overview

An auxiliary contract that has the functionality to create and use discount coupons for sales.

Issues

C9-01 'memory' is used instead of 'storage'

High Resolved

In `setCoupon()`, `buyCoupon()` , `useCoupon()` a memory variable `coupon` is used instead of a storage variable (L55, L184, L198). Because of this, all changes made to this variable are not saved, which leads to incorrect works of these functions.

```
function buyCoupon(uint256 _discountRate, uint256 _tokenNumber) external {
        ...
        CouponInfo memory coupon = _couponInfo[msg.sender][coupon_number];
        coupon.couponUser = msg.sender;
        coupon.couponDiscount = _discountRate;
        uint256 _couponStart = block.timestamp;
        coupon.couponStart = _couponStart;
        coupon.coupon_type = CouponType.LNE;
        coupon.couponEnd = (9125 days) +_couponStart;
        coupon.couponUses = 1;
        coupon.couponActive = 1;
        ...
```

```
    }
```

Recommendation

It is necessary to change the work with these variables by changing the modifier to `storage`. Be careful, as you can create non-optimization for gas if you frequently access storage variables.

C9-02 Requirement will always be satisfied

Medium Resolved

On L182 has a requirement that is always satisfied, `_couponInfo[_address][_couponNumber].couponUses` will always be greater than or equal to 0, which means that the coupon can be used after its validity expired and no uses left. And considering that changes in `useCoupon()` are never saved because the coupon variable is stored in the `memory` variable, then the coupon can always be used.

```
function useCoupon(address _address, uint256 _couponNumber) external {
    ...
    require((_couponInfo[_address][_couponNumber].couponEnd > block.timestamp)||
(_couponInfo[_address][_couponNumber].couponUses >= 0), "Your coupon is expired!");

    CouponInfo memory coupon = _couponInfo[_address][_couponNumber];
    coupon.couponUses = coupon.couponUses - 1;
    ...
}
```

Recommendation

Change the boolean operator to `>` instead of `>=` and work with `storage`, but not `memory` variable.

C9-03 Reversed requirement

Medium Resolved

On L181 there is a reverse requirement, it should check that the coupon has started to work, but does the opposite, which leads to the fact that you can use the coupon even before the start of its action.

```
function useCoupon(address _address, uint256 _couponNumber) external {
    ...
    require((_couponInfo[_address][_couponNumber].couponStart >= block.timestamp), "Your
coupon has not started yet!");
    ...
}
```

Recommendation

Change the boolean operator to `<=` instead of `>=`.

C9-04 Gas optimization

Low Resolved

a. `deadaddr` should be `const`;

b. An `activeCoupon()` modifier is not used;

c. `couponTimeLeft()` should have `view` modifier;

d. Multiple reads `_couponCount` on L54,87.

C9-05 Owner can buy coupon for free

Info Resolved

Using the `setCoupon()` function, the owner of the contract can create coupons for himself for free.

C10. JETIToken.sol

Overview

Pausable RFI token with blacklist.

Issues

C10-01 Fees may reach 100%

High Resolved

Total fees on token transfers may reach up to 100%.

Recommendation

Make the upper bound more reasonable for a fees sum.

C10-02 Exaggerated owners rights

High Resolved

a. The owner can set total fees exceeding 100% with `setTaxFeePercent()` and `setLiquidityFeePercent()` what will cause token transfers to halt;

b. The owner can zero as max transaction amount in `setMaxTxAmount()` leading to a token transfers stop.

Recommendation

It is recommended to remove these functions or make them more stringent validation of input parameters.

C10-03 Gas optimization

Medium Resolved

a. The state variable `_contract`, `_name`, `_symbol`, `_decimals` can be declared as `constant` to save gas;

b. The `whenNotPaused()` modifier and blacklist check duplication in private methods;

c. L603 `from` check is redundant;

d. `excludedAccount` is the same as `!takeFee` in `_transfer()`;

e. Multiple `_getRate()` calculation during transfer, which iterates the array and may exceed gas

block limit.

C10-04 Wrong variable in blacklist check

Medium Resolved

The state variable `_address` is the contract deployer, but not the real `msg.sender` calling a transfer transaction. Because of this, only the token deployer can be blacklisted, but not a user.

Recommendation

Change `_address` on `msg.sender` where needed.

C11. Multiple contracts

Overview

The issues in this section relate to multiple contracts.

Issues

C11-01 Missing tests

High Resolved

No tests were provided for the project. As the audit revealed a lot of errors in boolean comparisons and incorrect work of many functions we strongly advise covering main business logic scenarios with integration tests with coverage of at least 90% until we can not guarantee the project works in accordance with the documentation.

C11-02 Few events

Medium Resolved

The project lacks events. We recommend emitting events on important value changes to be easily tracked off-chain. No events are emitted in important project functions.

C11-03 Lack of validation of input parameters

Medium Resolved

In most contracts, there is no validation of numeric input parameters, this is very important because, for example, some fields directly affect the size of the fee, crucial round sale parameters, prize amounts, and owner addresses. Their incoherence or impropriety may result in various unfavorable sales results.

Recommendation

Validate all important input parameters in constructors and methods of all contracts.

C11-04 Floating Pragma

Info Resolved

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

C11-05 No support of tokens with commissions

Info Acknowledged

The launchpad doesn't support tokens with commissions, rebase, or RFI.

C11-06 Swaps don't have min amount and deadlines

Info Resolved

Swap and liquidity addition inside `JETIToken` and `SaleVault` contracts are executed with zero slippage and `block.timestamp` deadline which does not affect the transaction's success. These parameters should be obtained off-chain, otherwise, a buyback operation is vulnerable to sandwich attacks.

5. Conclusion

10 high, 8 medium, 9 low severity issues were found during the audit. 10 high, 8 medium, 9 low issues were resolved in the update.

The audit revealed numerous problems that usually become evident during the testing stage. On recheck, the code was provided with sufficient test coverage, which ensures to a certain extent the code doesn't have other hidden logic violations, and miscalculations and all business scenarios work appropriately.

This audit includes recommendations on code improvement and the prevention of potential attacks.

It should be noted that the audit was done on the contracts' code published to Github and it does not guarantee that the deployed version will be the same. Users must check the code of the deployed contracts they are interacting with.

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.
- **Informational.** Issues that do not impact the contract operation. Usually, informational 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