BLOCKSEC

Security Audit Report for IFTieredSale Contracts

Date: August 30, 2024 **Version:** 1.1

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Impossible Finance
Target	IFTieredSale Contracts

Version History

Version	Date	Description
1.0	July 10, 2024	First release
1.1	August 30, 2024	Add new commits (`Version 3` & `Version 4`)

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-08-30 10:34:24+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is the IFTieredSale Contracts of Impossible Finance [1]. These contracts facilitate tiered node sales with a whitelist mechanism and promotional code discounts. Whitelisted users can make purchases in permitted tiers, with different pricing, owner rewards, and promo code configurations.

Please note that only contracts inside the `contracts` folder in the repository are within the scope of this audit. Other files are not included. Additionally, all dependencies of the smart contract within the audit scope are considered reliable in terms of both functionality and security and are therefore not included in the audit.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
IFTieredSale Contracts	Version 1	17042f18ee4b647787d83b7da7f061328f554b4c
	Version 2	f96ab456c1a7cae9dbf4de092dd2a32d9d340f18
	Version 3	4c770b4e8553de8dd86ec8d93b8938b751e0077c
	Version 4	2b3530c08afadf3ab9c9d54e6d21a9f5a7b69d57

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit can-

[1] https://github.com/ImpossibleFinance/impossible-node-sale/tree/tiered-sale

not be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **six** recommendations and **three** notes.

- Medium Risk: 1
- Low Risk: 1
- Recommendation: 6
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Incorrect logic for loading promoCodeAddress from memory	Software Security	Fixed
2	Low	Incorrect codePurchaseAmount accounting	DeFi Security	Fixed
3	-	Apply sanity checks on parameters	Recommendation	Fixed
4	-	Remove redundant code	Recommendation	Fixed
5	-	Correct the typo in the function name	Recommendation	Fixed
6	-	Implement the setter function for claimRewardsEnabled	Recommendation	Fixed
7	-	Remove unused variable	Recommendation	Fixed
8	-	Avoid unexpected reverts due to underflows	Recommendation	Fixed
9	-	Potential centralized risks	Note	-
10	-	Lack of support for non-standard ERC20 tokens	Note	-
11	-	Lack of implementation for user withdrawals	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Incorrect logic for loading promoCodeAddress from memory

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The IFTieredSale contract allows users to use an *address promo code* to purchase saleTokens. The promo code is considered an *address promo code* when its length equals 42 characters (i.e., the length of an address in hexadecimal format). In such cases, the promoCodeOwnerAddress of the promo code is the code itself. However, the contract incorrectly processes these promo codes, resulting in loading of incorrect addresses.

For example, in the executePurchase function, it attempts to load the first 20 bytes of the _promocode using mload(add(_promoCode, 20)) on line 273. This approach is problematic as mload directly loads the raw bytes, not the actual characters. As a result, the output is not the correct address but rather truncated UTF-8 bytes.

```
function _isAddressPromoCode(string memory _promoCode) internal pure returns (bool) {
    return bytes(_promoCode).length == 42;
}
```

Listing 2.1: contracts/IFTieredSale.sol

```
function executePurchase (string memory _tierId, uint256 _amount, uint256 _price, string memory _promoCode) private nonReentrant {
    Tier storage tier = tiers[_tierId];
    require(!tier.isHalt, "Purchases in this tier are currently halted");
    require(tier.startTime <= block.timestamp && block.timestamp <= tier.endTime, "Tier is not active");
    require(_amount > 0, "Can only purchase non-zero amounts");
    require(
        tier.maxAllocationPerWallet == 0 || purchasedAmountPerTier[_tierId][msg.sender] + _amount <= tier.maxAllocationPerWallet,
        "Amount exceeds wallet's maximum allocation for this tier"
    );
    require(
        tier.maxTotalPurchasable == 0 || saleTokenPurchasedByTier[_tierId] + _amount <= tier.maxTotalPurchasable,
        "Amount exceeds tier's maximum total purchasable"
    );

    totalPaymentReceived += _amount * _price;
    purchasedAmountPerTier[_tierId][msg.sender] += _amount;
    saleTokenPurchasedByTier[_tierId] += _amount;

    uint256 totalCost = _amount * _price; // in gwei

    // no need to validate address promo code at purchase
    if (_isAddressPromoCode(_promoCode)) {
        if (promoCodes[_promoCode].promoCodeOwnerAddress == address(0)) {
            address promoCodeAddress;
            assembly {
                promoCodeAddress := mload(add(_promoCode, 20))
            }
            promoCodes[_promoCode].promoCodeOwnerAddress = promoCodeAddress;
        }
        uint256 ownerRewards = totalCost * addressPromoCodePercentage / 100;
        totalRewardsUnclaimed += ownerRewards;
        promoCodes[_promoCode].promoCodeOwnerEarnings += ownerRewards;
        promoCodes[_promoCode].totalPurchased += totalCost;
    }
```

Listing 2.2: contracts/IFTieredSale.sol

The same issue is also present in the _validatePromoCode function.

```
function _validatePromoCode(string memory _promoCode) internal view {
    require(bytes(_promoCode).length != 0, "Promo code is empty");

    // if the promo code is an address, check if it has purchased a node code
    // if the promo code is not an address, check if it is added by the admin (by checking the discount percentage)
    if (!_isAddressPromoCode(_promoCode)) {
        require(promoCodes[_promoCode].discountPercentage != 0, "Invalid promo code discount percentage");
        return;
    }

    // prceed to check if the address has purchased a node
    address promoCodeAddress;
    assembly {
        promoCodeAddress := mload(add(_promoCode, 20))
    }
```

Listing 2.3: contracts/IFTieredSale.sol

Impact The owner of the address promo code cannot be loaded correctly.

Suggestion Revise the code logic accordingly.

2.2 DeFi Security

2.2.1 Incorrect `codePurchaseAmount` accounting

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `IFTieredSale` contract, the `codePurchaseAmount` mapping variable is assigned values incorrectly. According to the code annotation, it should track the total purchased amount. However, in the `whitelistedPurchaseInTierWithCode` function, only the price is counted. Although this variable remains unused in the current contract implementation, the discrepancy may introduce potential issues if its value is used elsewhere.

```
function whitelistedPurchaseInTierWithCode(
    string memory _tierId,
    uint256 _amount,
    bytes32[] calldata _merkleProof,
    string memory _promoCode,
    uint256 _allocation
) public {
    // Ensure promo codes are allowed for the tier and the promo code is valid
    require(tiers[_tierId].allowPromoCode, "Promo code is not allowed for this tier");
    _validatePromoCode(_promoCode);
    bytes32 tierWhitelistRootHash = tiers[_tierId].whitelistRootHash;
    if (tierWhitelistRootHash != bytes32(0)) {
        require(checkTierWhitelist(_tierId, msg.sender, _merkleProof, _allocation), "Invalid proof");
        require(purchasedAmountPerTier[_tierId][msg.sender] + _amount <= _allocation, "Purchase exceeds allocation");
    }

    uint8 discount = calculateDiscount(_promoCode);
    uint256 discountedPrice = tiers[_tierId].price * (100 - discount) / 100; // in gwei
    codePurchaseAmount[_promoCode] += discountedPrice;
    executePurchase(_tierId, _amount, discountedPrice, _promoCode);
}
```

Listing 2.4: contracts/IFTieredSale.sol

Impact The use of an incorrect `codePurchaseAmount` value may lead to unexpected behavior.

Suggestion Revise the `codePurchaseAmount` accounting.

2.3 Additional Recommendation

2.3.1 Apply sanity checks on parameters

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `IFTieredSale` contract, several functions lack parameters sanity checks. It is advisable to implement these checks to ensure proper configuration.

1. The `addOperator` function should check that the `operator` address is not a zero address.

```
    function addOperator(address operator) public onlyRole(DEFAULT_ADMIN_ROLE) {
        grantRole(OPERATOR_ROLE, operator);
    }
```

Listing 2.5: contracts/IFTieredSale.sol

2. In the `setTier` function, a check should be added to ensure that the `startTime` and `endTime` of a tier satisfy the following condition:
$block.timestamp < tiers[\_tierId].startTime < tiers[\_tierId].endTime < endTime.$
Additionally, the function emits two identical `TierUpdated` events, which may be redundant.

```
    function setTier(
        string memory _tierId,
        uint256 _price,
        uint256 _maxTotalPurchasable,
        uint256 _maxAllocationPerWallet,
        bytes32 _whitelistRootHash,
        uint8 _bonusPercentage,
        bool _isHalt,
        bool _allowPromoCode,
        bool _allowWalletPromoCode,
        uint256 _startTime,
        uint256 _endTime
    ) public onlyOperator {
        // Validate input data
        require(_bonusPercentage <= 100, "Invalid bonus percentage");
        require(_price > 0, "Invalid price");
        require(_bonusPercentage <= MAX_BONUS_PERCENTAGE, "Invalid bonus percentage");

        tiers[_tierId] = Tier({
            price: _price,
            maxTotalPurchasable: _maxTotalPurchasable,
            maxAllocationPerWallet: _maxAllocationPerWallet,
            whitelistRootHash: _whitelistRootHash,
            bonusPercentage: _bonusPercentage,
            isHalt: _isHalt,
            allowPromoCode: _allowPromoCode,
            allowWalletPromoCode: _allowWalletPromoCode,
            startTime: _startTime,
            endTime: _endTime

        });
        emit TierUpdated(_tierId);
```

Listing 2.6: contracts/IFTieredSale.sol

3. The addPromoCode function adds a new owner promo code specified by the _code parameter. Thus, it should ensure that _code is not an empty string and does not match the length of an address to differentiate it from an *address promo code*.

```
    function addPromoCode(
        string memory _code,
        uint8 _discountPercentage,
        address _promoCodeOwnerAddress,
        address _masterOwnerAddress,
        uint8 _baseOwnerPercentageOverride,
        uint8 _masterOwnerPercentageOverride
    ) public onlyOperator {
        if (promoCodes[_code].discountPercentage != 0 || promoCodes[_code].
             promoCodeOwnerAddress != address(0)){
            revert("Promo code already exists");
        }
        // Validate the discount percentage and owner addresses
        _validatePromoCodeSetting(_discountPercentage, _promoCodeOwnerAddress,
             _masterOwnerAddress, _baseOwnerPercentageOverride,
             _masterOwnerPercentageOverride);

        // Add the promo code
        promoCodes[_code] = PromoCode({
            discountPercentage: _discountPercentage,
            promoCodeOwnerAddress: _promoCodeOwnerAddress,
            masterOwnerAddress: _masterOwnerAddress,
            promoCodeOwnerEarnings: 0,
            masterOwnerEarnings: 0,
            totalPurchased: 0,
            baseOwnerPercentageOverride: _baseOwnerPercentageOverride,
            masterOwnerPercentageOverride: _masterOwnerPercentageOverride
        });
        ownerPromoCodes[_promoCodeOwnerAddress].push(_code);
        ownerPromoCodes[_masterOwnerAddress].push(_code);
        allPromoCodes.push(_code);
        emit PromoCodeAdded(_code, _discountPercentage, _promoCodeOwnerAddress,
            _masterOwnerAddress);
    }
```

Listing 2.7: contracts/IFTieredSale.sol

Impact The misconfigurations may lead to unexpected behavior.

Suggestion Add sanity checks on the function parameters.

2.3.2 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Redundant code exists in the `IFTieredSale` contract and can be safely removed.

1. In the `setTier` function, the check on line 126 is redundant, as the check on line 128 ensures that `_bonusPercentage` is below `MAX_BONUS_PERCENTAGE` (i.e., 5).

```
125     // Validate input data
126     require(_bonusPercentage <= 100, "Invalid bonus percentage");
127     require(_price > 0, "Invalid price");
128     require(_bonusPercentage <= MAX_BONUS_PERCENTAGE, "Invalid bonus percentage");
```

Listing 2.8: contracts/IFTieredSale.sol

2. The `IFTieredSale` contract includes an unused inheritance from `IFWhitelistable`.

```
    contract IFTieredSale is ReentrancyGuard, AccessControl, IFFundable, IFWhitelistable
        {
        ...
```

Listing 2.9: contracts/IFTieredSale.sol

3. Within the `Tier` struct, a field named `allowWalletPromoCode` is defined but never used. It can be removed from the struct if it is not reserved for other design purposes.

```
    struct Tier {
        uint256 price; // Price per tier in gwei.
        uint256 maxTotalPurchasable; // Total limit per tier (0 means no limit), specified
             in ether.
        uint256 maxAllocationPerWallet; // Limit per wallet (0 means no limit), specified
            in ether.
        uint8 bonusPercentage; // Additional bonus percentage applicable for this tier.
        bytes32 whitelistRootHash; // Merkle root hash for whitelisting.
        bool isHalt; // Flag to halt transactions for this tier if set to true.
        bool allowPromoCode; // Flag to allow promo codes for this tier.
        bool allowWalletPromoCode; // Flag to allow promo codes specific to wallets.
        uint256 startTime; // Start time for this tier.
        uint256 endTime; // End time for this tier.
    }
```

Listing 2.10: contracts/IFTieredSale.sol

4. In the withdrawPromoCodelRewards function, the non-zero length check on line 344 is redundant, as the same check is performed in the _validatePromoCode function.

```
342    function withdrawPromoCodelRewards (string memory _promoCode) public nonReentrant {
343        require(claimRewardsEnabled, "Claim rewards is disabled");
344        require(bytes(_promoCode).length > 0, "Invalid promo code");
345        _validatePromoCode(_promoCode);
```

Listing 2.11: contracts/IFTieredSale.sol

Impact N/A

Suggestion Remove redundant codes from the IFTieredSale contract.

2.3.3 Correct the typo in the function name

Status Fixed in Version 2

Introduced by Version 1

Description There is a function named withdrawPromoCodelRewards in the IFTieredSale contract, which appears to be a potential typo.

Impact N/A

Suggestion Revise the function name as withdrawPromoCodeRewards.

2.3.4 Implement the setter function for claimRewardsEnabled

Status Fixed in Version 2

Introduced by Version 1

Description The claimRewardsEnabled variable in the IFTieredSale contract is used in the withdrawReferenceRewards and withdrawPromoCodelRewards functions to determine if reward withdrawal is enabled. However, this variable is initialized as true and does not have a public setter function in the contract to update its value.

```
bool public claimRewardsEnabled = true;
```

Listing 2.12: contracts/IFTieredSale.sol

```
function withdrawReferenceRewards () public nonReentrant {
    address promoCodeOwner = msg.sender;
    require(claimRewardsEnabled, "Claim rewards is disabled");
```

Listing 2.13: contracts/IFTieredSale.sol

```
function withdrawPromoCodelRewards (string memory _promoCode) public nonReentrant {
    require(claimRewardsEnabled, "Claim rewards is disabled");
    require(bytes(_promoCode).length > 0, "Invalid promo code");
    _validatePromoCode(_promoCode);
```

Listing 2.14: contracts/IFTieredSale.sol

Impact The reward withdrawals are always enabled, rendering the function checks redundant.

Suggestion Add a privileged setter function to update the `claimRewardsEnabled` variable, or remove all related code.

2.3.5 Remove unused variable

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the `validateWalletPromoCode` function, the `sum` variable is updated within loops but never used. To optimize gas, this variable can be safely removed.

```
function validateWalletPromoCode(address promoCodeAddress) public view returns (bool) {
   if (promoCodeAddress == address(0)) {
      return false;
   }

   uint256 sum = 0;
   for (uint i = 0; i < tierIds.length; i++) {
      if (tiers[tierIds[i]].price == 0) {
         continue;
      }
      if (purchasedAmountPerTier[tierIds[i]][promoCodeAddress] > 0) {
         // return true if the address has purchased at least one node
         sum += purchasedAmountPerTier[tierIds[i]][promoCodeAddress];
         return true;
      }
   }
   return false;
}
```

Listing 2.15: contracts/IFTieredSale.sol

Impact N/A

Suggestion Remove the unused variable.

2.3.6 Avoid unexpected reverts due to underflows

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the `getAllPromoCodeInfo` function, the requirement `fromIdx < toIdx` should be moved after resetting `toIdx` to `allPromoCodes.length`. Failing to do so could cause an unexpected revert due to underflow in cases where the input parameters meet the condition `allPromoCodes.length < fromIdx < toIdx`. Specifically, after `toIdx` is reset on line 576, it may becomes less than `fromIdx`, leading to a function revert due to underflow rather than returning a formatted error message. The same issue is present in the `getAllPromoCodes` function.

```
function getAllPromoCodeInfo(uint256 fromIdx, uint256 toIdx) public view returns (PromoCode[] memory) {
    require(fromIdx < toIdx, "Invalid range");
    if (toIdx > allPromoCodes.length) {
        toIdx = allPromoCodes.length;
    }
    PromoCode[] memory promoCodeInfos = new PromoCode[](toIdx - fromIdx);
    for (uint i = fromIdx; i < toIdx; i++) {
        promoCodeInfos[i - fromIdx] = promoCodes[allPromoCodes[i]];
    }
    return promoCodeInfos;
}
```

Listing 2.16: contracts/IFTieredSale.sol

```
function getAllPromoCodes(uint256 fromIdx, uint256 toIdx) public view returns (string[] memory) {
    require(fromIdx < toIdx, "Invalid range");
    if (toIdx > allPromoCodes.length) {
        toIdx = allPromoCodes.length;
    }
    string[] memory promoCodeList = new string[](toIdx - fromIdx);
    for (uint i = fromIdx; i < toIdx; i++) {
        promoCodeList[i] = allPromoCodes[i];
    }
    return promoCodeList;
}
```

Listing 2.17: contracts/IFTieredSale.sol

Impact These functions may revert without a formatted error message in certain cases.

Suggestion Revise the code accordingly.

2.4 Note

2.4.1 Potential centralized risks

Introduced by `Version 1`

Description In the `IFTieredSale` contract, there are privileged roles capable of modifying critical configurations, such as adding or removing operators and changing the time of an active sale tier. This introduces a risk of centralization. If the private keys of these privileged accounts are leaked, the launchpad could potentially be compromised.

2.4.2 Lack of support for non-standard ERC20 tokens

Introduced by `Version 1`

Description The `IFTieredSale` contract should only support standard ERC20 tokens. Supporting non-standard ERC20 tokens (e.g., deflationary tokens) can introduce potential security risks. For instance, with deflationary tokens, the actual amount received by users may differ from their expectations.

2.4.3 Lack of implementation for user withdrawals

Introduced by `Version 1`

Description The `IFTieredSale` contract inherits from `IFFundable`, which supports funding the contract with sale tokens. Users should be allowed to withdraw their purchased sale tokens after the sale ends. However, the contract does not implement an external withdraw function to facilitate user withdrawals.

Feedback from the Project We don't plan to implement this for now since it is a node sale. We'll airdrop the nodes accordingly, so the users don't need the sale token.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS