Security Audit Report

X2Y2 Protocol

SECBIT

Feb 22, 2022

1. Introduction

The X2Y2 Protocol is a decentralized NFT marketplace, which offers NFT trade matching and NFT auction services. SECBIT Labs conducted an audit from February 12th to February 22th, 2022, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the X2Y2 Protocol contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Discussion of the parameter `rewardToken`.	Info	Fixed
Design & Implementation	4.3.2 Failure may occur when a user calls the `withdraw()` function to claim unlocked X2Y2 tokens and rewards.	Low	Fixed
Design & Implementation	4.3.3 The use of `treasuryWithdraw()` function may affect the contract's normal functionality.	Low	Fixed
Design & Implementation	4.3.4 After multiple auctions, the seller revokes the auction resulting in a loss of funds for the latest bidder.	Info	Discussed
Design & Implementation	4.3.5 When the auction fund is ether, using the	Medium	Fixed

	`_transferTo()` function to refund the previous bidder's auction fund may cause a DOS attack.		
Design & Implementation	4.3.6 Discussion of the parameter `dataMask`.	Low	Fixed
Design & Implementation	4.3.7 The parameter `itemHash` in `_takePayment()` function is not used.	Info	Discussed
Design & Implementation	4.3.8 Discussion on permission design of partial functions.	Info	Discussed
Design & Implementation	4.3.9 Discussion on order signature verification logic.	Info	Discussed
Design & Implementation	4.3.10 A discussion of how some of the parameters are set.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the X2Y2 Protocol contract is shown below:

- Project website
  - https://x2y2.io
- Smart contracts for audit
  - ERC721Delegate - 0xf849de01b080adc3a814fabe1e2087475cf2e354
  - X2Y2_r1 - 0x6d7812d41a08bc2a910b562d8b56411964a4ed88
  - Presale - 0xc2f44bc508b6b50047a2f3afb1984ed105070be1
  - VestingContractWithFeeSharing - 0x6d11992a247ae0d726cb967a70fc981e8308b723

2.2 Contract List

The following content shows the contracts included in the X2Y2 protocol, which the SECBIT team audits:

Name	Lines	Description
ERC721Delegate.sol	100	Processes the transfer of NFT tokens to be traded.
X2Y2_r1.sol	508	A core contract provides NFT token trade matching and NFT token auction services.
AddressUpgradeable.sol	73	Collection of functions related to the address type.
MarketConsts.sol	91	A library contract that defines the data structure for NFT token transactions.
Presale.sol	204	The X2Y2 token pre-sale contract.
ITokenStaked.sol	3	A library contract of token staking.
VestingContractWithFeeSharing.sol	83	A contract locks the X2Y2 token and releases it in batches according to the expected timeline.

3. Contract Analysis

This part describes code assessment details, including two items: "role classification" and "functional analysis".

3.1 Role Classification

There are two key roles in X2Y2 Protocol: Governance Account and Common Account.

- Governance Account

    - Description

      Contract administrator

    - Authority
        - Control the use of core contract functions
        - Transfer ownership
        - Update signers and delegates
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by transferring the governance account.

- Common Account
    - Description

      Participate in NFT token transactions

    - Authority
        - Sell or buy NFT token
        - Launch or participate in an auction of NFT tokens
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The X2Y2 protocol aims to build a decentralized NFT market and give it back to the community. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into several parts:

ERC721Delegate

This contract primarily handles the transfer of NFT tokens between buyers and sellers. The NFT token for auction will be transferred from the seller to this contract.

The main functions in `ERC721Delegate` are as below:

- `executeSell()`

  When the order type is `COMPLETE_SELL_OFFER`, the NFT will be transferred directly from the seller to the buyer.

- `executeBuy()`

  When the order type is `COMPLETE_BUY_OFFER`, the NFT will be transferred directly from the seller to the buyer.

- `executeBid()`

  When a buyer starts to bid on an NFT token auction, the NFT will be transferred from the seller to this contract first.

- `executeAuctionComplete()`

  After successful bidding, the buyer will receive the NFT token.

- `executeAuctionRefund()`

  If the bid is not completed, the seller will get back the NFT token he transferred in.

X2Y2_r1

This contract implements the core functions of the X2Y2 protocol: the NFT token sale and the NFT token auction.

The main functions in `X2Y2_r1` are as below:

- `cancel()`

  The signer can revoke the transaction before the NFT token has been traded.

- `run()`

  This function implements the sale and auction of NFT tokens.

Presale

This contract is a pre-sale contract for the X2Y2 token. The X2Y2 token purchased by the user will be released linearly.

The main functions in `Presale` are as below:

- `deposit()`

  Users can purchase a specified share of X2Y2 tokens at a specified price. These X2Y2 tokens will not be transferred to the user directly but will be released linearly after the pre-sale closes.
- `harvest()`

  Users claim reward tokens.
- `withdraw()`

  Users claim the reward tokens and the unlocked X2Y2 tokens.

4. Audit Detail

This part describes the process, and the detailed results of the audit also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and inline assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓

13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Discussion of the parameter `rewardToken`.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Compatibility issue	Fixed

Description

The user can purchase a specified share of X2Y2 tokens through a pre-sale. The `rewardToken` parameter indicates the type of reward the user can receive for holding the X2Y2 token. It is necessary to determine whether the `rewardToken` is wrapped ether or any other coin. It may need to be considered for token compatibility if it allows any token. In this case, the `safeTransfer()` function will need to be used.

```
//@audit located in Presale.sol
function _harvest(address user) internal returns (uint256) {
        (uint256 _pending, uint256 _debt) =
_pendingReward(user);

        if (_pending > 0) {
            totalRewardDistributed += _pending;
            userInfo[user].rewardDebt = _debt;

            //@audit It is recommended to use safeTransfer()
            //       instead of the transfer() function.
            rewardToken.transfer(user, _pending);
            emit Harvest(user, _pending);
        }
        return _pending;
    }
```

Status

The team has adopted this suggestion, and used the `safeTransfer()` function instead of the `transfer()` function.

4.3.2 Failure may occur when a user calls the `withdraw()` function to claim unlocked X2Y2 tokens and rewards.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Functional failure	Fixed

Description

The `_harvest()` function will fail to be executed when there are not enough rewards in the contract to pay the user. Further, the user will not call the `withdraw()` function. It directly affects the withdrawal of the user's principal (X2Y2 token).

```
//@audit located in Presale.sol
function withdraw() external nonReentrant {
        require(currentPhase == SalePhase.Staking, 'Withdraw:
Phase must be Staking');
        require(userInfo[msg.sender].hasShare, 'Withdraw: User
not eligible');

        uint256 pending = _pendingTokens(msg.sender);

        require(pending > 0, 'Withdraw: No pending token');

        //@audit maybe failed
        _harvest(msg.sender);

        userInfo[msg.sender].tokensClaimed += pending;
        x2y2Token.safeTransfer(msg.sender, pending);

        emit Withdraw(msg.sender, pending);
    }
```

Status

The team adds the `emergencyWithdraw()` function to prevent this situation.

4.3.3 The use of `treasuryWithdraw()` function may affect the contract's normal functionality.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Design logic	Fixed

Description

The `treasuryWithdraw()` function is intended to withdraw rewards that do not belong to any user, such as rewards that have been excluded due to unstake. However, the implementation allows the administrator to withdraw funds after the staking is over. In extreme cases, this may affect the withdrawal of rewards by users. In addition, "rewards that do not belong to any user" may be difficult to calculate. Suppose the calculation result is inaccurate (too large). In that case, the last user to withdraw may not be able to collect the reward properly.

```
// @audit located in Presale.sol
function treasuryWithdraw(uint256 amount) external onlyOwner
nonReentrant {
        require(block.number > stakingEndBlock, 'Owner:
staking have not ended yet');
        require(amount > 0, 'Owner: withdraw > 0');

        tokenRewardTreasuryWithdrawn += amount;
        rewardToken.safeTransfer(msg.sender, amount);
        emit TreasuryWithdraw(amount);
    }
```

Status

The team has postponed the time when administrators can take out rewards. Users will have enough time to claim their rewards.

4.3.4 After multiple auctions, the seller revokes the auction resulting in a loss of funds for the latest bidder.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

After the auction has started, the contract allows the seller to withdraw the auction before the closing date. Then, the contract will return the most recent bidder's bid funds. However, according to the current code logic, a portion of the latest bidder's funds will be used as an incentive to compensate the previous bidder. In this case, if the seller revoked the auction, it would directly result in the latest bidder losing part of his bid.

```
// @audit located in X2Y2_r1.sol
function _run(
        Market.Order memory order,
        Market.SettleShared memory shared,
        Market.SettleDetail memory detail
    ) internal virtual returns (uint256) {
        uint256 nativeAmount = 0;

        Market.OrderItem memory item =
order.items[detail.itemIdx];
        bytes32 itemHash = _hashItem(order, item);

        ......
        } else if (
            detail.op == Market.Op.REFUND_AUCTION ||
            detail.op == Market.Op.REFUND_AUCTION_STUCK_ITEM
        ) {
            require(
                inventoryStatus[itemHash] ==
Market.InvStatus.AUCTION,
                'cannot cancel non-auction order'
            );
            Market.OngoingAuction storage auc =
ongoingAuctions[itemHash];

            if (auc.netPrice > 0) {
                //@audit The netPrice will be sent to the
current bidder.
                _transferTo(order.currency, auc.bidder,
auc.netPrice);
```

```
                emit EvAuctionRefund(
                    itemHash,
                    address(order.currency),
                    auc.bidder,
                    auc.netPrice,
                    0
                );
            }
            _assertDelegation(order, detail);

        ......

        _emitInventory(itemHash, order, item, shared, detail);
        return nativeAmount;
    }
```

Suggestion

This issue needs to be confirmed if it is within the design considerations.

Status

The developer team explains the issue. Cancellation of auctions is only allowed in unusual cases to solve the problem of not being able to operate a specific NFT. It is controlled by the backend program and authorized by signature.

4.3.5 When the auction fund is ether, using the `_transferTo()` function to refund the previous bidder's auction fund may cause a DOS attack.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	DOS	Fixed

Description

The _transferTo() function processes the previous bidder's funds. It will actively transfer the funds under this contract to the relevant bidder. The normal logic would be as follows: the core contract would call the sendValue() function to transfer the corresponding amount of ether to the previous bidder. The sendValue() function calls the call() function internally, which will determine the return value success. When success == true, the transfer will succeed.

Consider the following scenario: a bidder participates in the auction using a custom contract and maliciously overrides the fallback function to consume gas (e.g., an infinite loop). When the next bidder joins the auction, the core contract will use the _transferTo() function to initiate a transfer of the previous bidder's funds and compensation back. The malicious gas-consuming fallback function will be triggered. It could deplete the gas of the new bidder and directly cause the bid to fail or increase the cost of the new bid, which seriously affects the normal contract logic functionality.

```
//@audit located in X2Y2_r1.sol
function _run(
        Market.Order memory order,
        Market.SettleShared memory shared,
        Market.SettleDetail memory detail
    ) internal virtual returns (uint256) {
        uint256 nativeAmount = 0;

        Market.OrderItem memory item =
order.items[detail.itemIdx];
        bytes32 itemHash = _hashItem(order, item);

        ......
        } else if (detail.op == Market.Op.BID) {
            require(order.intent == Market.INTENT_AUCTION,
'intent != auction');

            ......
            if (!firstBid) {
                ......

                uint256 bidRefund = auc.netPrice;
                uint256 incentive = (detail.price *
detail.bidIncentivePct) / RATE_BASE;
                if (bidRefund + incentive > 0) {
                    //@audit Send the previous bidder's funds
                    //       and compensation to current
bidder.
                    _transferTo(order.currency, auc.bidder,
bidRefund + incentive);
                    emit EvAuctionRefund(
                        itemHash,
                        address(order.currency),
                        auc.bidder,
                        bidRefund,
                        incentive
                    );
                }

              ......
            }
        } else if (
            detail.op == Market.Op.REFUND_AUCTION ||
            detail.op == Market.Op.REFUND_AUCTION_STUCK_ITEM
        ) {
            if (auc.netPrice > 0) {
                //@audit Send the last bidder's bid funds
                //       (minus incentives) to that bidder.
                _transferTo(order.currency, auc.bidder,
auc.netPrice);
                emit EvAuctionRefund(
                    itemHash,
                    address(order.currency),
                    auc.bidder,
                    auc.netPrice,
                    0
                );
            }
           ......
        _emitInventory(itemHash, order, item, shared, detail);
        return nativeAmount;
    }

function _transferTo(
        IERC20Upgradeable currency,
        address to,
        uint256 amount
    ) internal virtual {
        if (amount > 0) {
            //@audit Using the following function
            //       when the funds are ether.
            if (_isNative(currency)) {
                AddressUpgradeable.sendValue(payable(to),
amount);
            } else {
                ......
            }
        }
    }

//@audit located in AddressUpgradeable.sol
function sendValue(address payable recipient, uint256 amount)
internal {
        require(address(this).balance >= amount, "Address:
insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value,
recipient may have reverted");
    }
```

Suggestion

One strategy is to ignore the boolean value `success` judgment and cap the gas to prevent consuming all available gas. An example is as follows:

```
function sendValue(address payable recipient, uint256 amount)
internal {
        require(address(this).balance >= amount, "Address:
insufficient balance");

        (bool success, ) = recipient.call{value: amount,gas:
40000}("");

        //require(success, "Address: unable to send value,
recipient may have reverted");
    }
```

Status

The backend checks whether the auction user is an EOA address. Only EOA addresses are allowed to join the auction currently. In the new version, the team has modified the `_transferTo()` function to impose a gas consumption limit on the `call()` function and ignore the return value.

4.3.6 Discussion of the parameter `dataMask`.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Data manipulation	Fixed

Description

The `dataMask` variable in the `Order` struct can be set to a special value by the order initiator to enable the replacement of subsequent trade targets. Presumably, it is used for an NFT buyer to make a buy offer to a specified NFT collection and buy anyone in that collection at a fixed price.

However, the contract code does not constrain the parameter dataMask scope of use. Using it in non-COMPLETE_BUY_OFFER scenarios introduces potential risks.

For example, if an NFT seller uses this feature, it corresponds to the COMPLETE_SELL_OFFER branch in the code. When sellers hold multiple NFT tokens of the same collection, they usually use the setApprovalForAll() function to authorize all NFT tokens they hold for trading. With the seller's permission, the current code allows the buyer to modify the data variable, leading to some unintended NFT tokens being sold.

```
//@audit link: https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC721/ERC721.sol#L136
function setApprovalForAll(address operator, bool approved)
public virtual override {
    _setApprovalForAll(_msgSender(), operator, approved);
}
```

A more specific example is as follows. Suppose the seller holds five NFT tokens of the same collection with token ids 1, 2, and 3. The seller only wants to sell the NFT tokens with1 and 2 token ids. The seller calls the setApprovalForAll() function. All three NFT tokens held by the seller are authorized at once for gas saving. In this case, a buyer can change the data parameter to replace the tokenID 1 or 2 with 3. It creates an unexpected situation where an unintended NFT token is sold. In extreme cases, both the NFT type and token ids can be replaced.

```
function _run(
        Market.Order memory order,
        Market.SettleShared memory shared,
        Market.SettleDetail memory detail
    ) internal virtual returns (uint256) {
        uint256 nativeAmount = 0;

        Market.OrderItem memory item =
order.items[detail.itemIdx];
        bytes32 itemHash = _hashItem(order, item);
        ......

        bytes memory data = item.data;
        {
            //@audit modifying data data with the permission
of the seller
            if (order.dataMask.length > 0 &&
detail.dataReplacement.length > 0) {
                _arrayReplace(data, detail.dataReplacement,
order.dataMask);
            }
        }
       ......
    }
```

The parameters require a final signature by the X2Y2 backend, so it relies on the backend to check these core parameters. The above issue would only occur in the extreme case of a user signing an order blindly, leading to abuse of the `dataMask` function while the backend failed to verify due to failure of the check logic or a signer private key leak.

Suggestion

Confirm the scope of `dataMask` usage and check the backend verification logic. The possibility of signer private key leakage is not as low as one might think. So we recommend refining the contract validation logic to ensure that user assets remain secure in extreme cases.

Status

The team has adopted our suggestion. In the new version, the `dataMask` feature is only allowed when performing the `COMPLETE_BUY_OFFER` operation, eliminating the previously mentioned risk in extreme cases.

4.3.7 The parameter `itemHash` in `_takePayment()` function is not used.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Redundant variable	Discussed

Description

The `itemHash` argument in the `_takePayment()` function is not yet used.

```
//@audit located in X2Y2_r1.sol
function _takePayment(
        bytes32 itemHash, // @audit never used?
        IERC20Upgradeable currency,
        address from,
        uint256 amount
    ) internal virtual returns (uint256) {
        if (amount > 0) {
            if (_isNative(currency)) {
                return amount;
            } else {
                currency.safeTransferFrom(from, address(this),
amount);
            }
        }
        return 0;
    }
```

Suggestion

We recommend checking that the design matches the implementation.

Status

The itemHash variable was initially being used in an event. Currently, it is a redundant variable here, which is harmless.

4.3.8 Discussion on permission design of partial functions.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Permission issue	Discussed

Description

The current implementation does not check if the order initiator is the transaction initiator for canceling orders and auctions. It only checks the backend's signature. It means that anyone with a legitimate signature can initiate a cancellation transaction.

```
//@audit located in X2Y2_r1.sol
// @audit anyone get the sig from signers could call this
function cancel(
      bytes32[] memory itemHashes,
      uint256 deadline,
      uint8 v,
      bytes32 r,
      bytes32 s
  ) public virtual nonReentrant whenNotPaused {}

  function _run(
      Market.Order memory order,
      Market.SettleShared memory shared,
      Market.SettleDetail memory detail
  ) internal virtual returns (uint256) {
      ...
      else if (detail.op == Market.Op.CANCEL_OFFER) {
      } else if (
          detail.op == Market.Op.REFUND_AUCTION
      ) {
      }
  }
```

Suggestion

It needs to be confirmed whether this issue is within the design considerations. If there is no specific need, we recommend adding strict checks to the contract to avoid potential risks (e.g., signatures exposed early in the memory pool, signer private key leakage, centralization risks).

Status

The `cancel()` function is designed to be executed by anyone who gets the signature. And the `run()` function checks the order initiator in the backend. The team chose to check these parameters on the backend to save gas.

4.3.9 Discussion on order signature verification logic.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

The current implementation does not include the core contract address for calculating the `orderHash`.

```
//@audit located in X2Y2_r1.sol
function _verifyOrderSignature(Market.Order memory order)
internal view virtual {
        address orderSigner;

        if (order.signVersion == Market.SIGN_V1) {
            bytes32 orderHash = keccak256(
                abi.encode(
                    order.salt,
                    order.user,
                    order.network, // @audit-ok prevent replay
                    order.intent,
                    order.delegateType,
                    order.deadline,
                    order.currency,
                    order.dataMask,
                    order.items.length,
                    order.items
                )
            );
            ...
        }
    }
```

Suggestion

The main contract is currently deployed in an upgradeable pattern, so a new version in the future will not change the contract address. Therefore the risk of signature replay is very low. If the contract address needs to be changed in the future, the way `orderHash` is calculated will have to be changed.

Status

This risk has been discussed with the team. A new version of the signature method will be used in the future upgrade and will not introduce the above risk.

4.3.10 A discussion of the way some of the parameters are set.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Fixed

Description

Important parameters in SettleDetail including bidIncentivePct, aucMinIncrementPct, aucIncDurationSecs, and fees are passed in by the user and signed by the backend before being passed to the smart contract for validation. The checks on the contract for these parameters are weak.

```
//@audit located in MarketConsts.sol
struct SettleDetail {
        Market.Op op;
        uint256 orderIdx;
        uint256 itemIdx;
        uint256 price;
        bytes32 itemHash;
        IDelegate executionDelegate;
        bytes dataReplacement;
        uint256 bidIncentivePct;
        uint256 aucMinIncrementPct;
        uint256 aucIncDurationSecs;
        Fee[] fees;
    }
```

Suggestion

Consider the single-point risk of the current design and strengthen backend logic checks and private key protection.

Status

This issue has been discussed. The team has taken our suggestion and added checks for the variables bidIncentivePct, aucMinIncrementPct, and aucIncDurationSecs in the new version.

4.4 Risks

4.4.1 Discussion on the potential risk of introducing an off-chain `signer` role

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

The X2Y2 core contract takes a novel hybrid design and implementation approach. The core business logic is verified and executed by the smart contract. Besides, additional validation work and parameter settings are moved off-chain. The validation is performed by the backend and signed and authorized on the server by a pre-defined signer role. This design can effectively simplify smart contract logic, reduce gas consumption, and flexibly adjust parameters and validation logic. It can even improve the difficulty of phishing attacks on users. Considering that this introduces a new off-chain role called `signer`, whose private key is stored on the server, it is necessary to think well in advance what risks the protocol will suffer in the extreme case of signer private key leakage. The risks that may be caused by private key leakage have been mentioned in related issues in the previous sections.

Status

This risk has been discussed with the team. The risk has been significantly reduced in the improved version of the code. The most severe consequences now are only the tampering of the fee recipients and the fee ratio, which is actually less impactful and easily detectable. The team will further restrict the privileges of a single signer in a future major release.

5. Conclusion

After auditing and analyzing the X2Y2 protocol contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. SECBIT Labs holds the view that the X2Y2 protocol has good code quality and concise implementation.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

🌎 https://secbit.io

✉ audit@secbit.io

🐦 @secbit_io