SECURITY AUDIT OF

LS TRADE

Public Report

Apr 12, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Apr 12, 2023. We would like to thank the LS Trade for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the LS Trade. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About LS Trade

LS Trade is a decentralized binary options trading platform of the next generation, designed to provide investors with a simple, transparent, and decentralized way of trading binary options.

LS Trade is an ecosystem comprises three core products, namely:

- Long Short X: enables users to trade binary options based on the price of Bitcoin using price data from ChainLink Oracle. All trading sessions and option contracts on Long Short X are stored and managed by smart contracts on the BNB Chain (BEP20) network.
- LS Coin (LSC): serves as the application and governance token of the LS Trade ecosystem.
- Burn To Earn: allows users to burn LSC to receive a reward equivalent to 125% of the value of the LSC burned.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the LS Trade.

SHA256 Sum	File
4e559bf40436d1e73d92d6daab7486b3fa9bb4b4643f48455c36d20c646c3857	bull-or-bear/contracts/LSTrade.sol
13b66dd9898643d11260ca89dc6de4185be9d4c35cefdc65bb42cd46ded8e92f	bull-or-bear/contracts/LSTradeState.sol
79354e6d9dc1826575c0762fe1cc8a847fe3605f94296065f9d3e4e0fe8f96b3	bull-or-bear/contracts/LSTradeAdmin.sol
cd197f75a6a34ccfa955c4f7f282723522a3e55aa9e3a1d95ab36fe3d6402b42	burn-to-earn/contracts/SafeMathX.sol
c0ea4c04c6050d4d9def7fae179bef48ee5620f50eac9579770e60180ce6b8f1	burn-to-earn/contracts/BurnToEarn.sol
f44f7967665c6e986dc7c8f6b707e856b80099e8abe6ed0c458ae5a60fd87df4	dapp-referral/contracts/Claim.sol

f7c2c6fefae28565e194e55d518f2dfcdf2e8caf86aa07eef6f121380 4fe4855	dapp-referral/contracts/USDTToken.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.

SEVERITY LEVEL	DESCRIPTION
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The LS Trade was written in `Solidity` language, with the required version to be `0.8.9`.

2.1.1. bull-or-bear

In this project, there are four primary smart contracts that serve different functions:

- `LSTrade.sol`: This contract allows users to interact with the system to order ticket bull or bear (Long or Short). Users can choose their desired option and the amount they want to order on the chosen option.
- `LSTradeState.sol`: This contract stores all the states of the project, including the price of Bitcoin, the current epoch, and the current ticket price. It also contains functions to update these states and ensure that they are synchronized across the system.
- `LSTradeAdmin.sol`: This contract allows the admin to set states and operate the contract. The admin can update the price of Bitcoin, set the ticket price for each epoch, and claim rewards for the winners of each epoch.

During each transaction for placing an order, there will be a waiting time that includes a period where orders cannot be placed. The party that loses the trade will provide liquidity to the winning party and will be charged a fee upon receiving the reward. After a certain number of fixed rounds (determined by the admin), the user will be eligible to claim additional LSC token rewards. These rewards will gradually decrease with each cycle.

2.1.2. burn-to-earn

BurnToEarn smart contract was written in Solidity programming language. The contract is designed to enable users to burn a certain `ERC20` token and receive a new token that will be vested for a period of time, with a specified Annual Percentage Yield (APY).

The contract utilizes `OpenZeppelin` libraries for features like `PausableUpgradeable`, `ReentrancyGuardUpgradeable`, `SafeERC20Upgradeable`, and `ERC20BurnableUpgradeable`.

Users can create a new vesting by sending a specified ERC-20 token to the contract and paying a specified native fee. The amount of the token must be greater than or equal to the minimum amount for vesting, which is set by the contract owner. The vesting is created based on the APY specified by the contract owner and the vesting period, which is also set by the contract owner. The contract creates a new vesting ID for each user and tracks their vesting through a mapping.

Users can claim the vested tokens once they become available. The contract calculates the amount of vested tokens that are available for claim based on the vesting period, the APY, and the time since the last claim. If a user attempts to claim their tokens before they are available, the transaction will fail.

The contract also includes features to allow the contract owner to change certain parameters, such as the minimum amount for vesting, the vesting period, the APY, and the native fee. There is also an event log that records certain activities, such as pausing the contract, creating a new vesting, and claiming vested tokens.

2.1.3. dapp-referral

The project allows for a payment claim mechanism, which requires a valid signature from an operator address to initiate a payment to the caller. The contract uses several libraries from the `OpenZeppelin` library, including `IERC20Upgradeable` and `SafeERC20Upgradeable`, and has four state variables: `admin`, `token`, `operator`, and a mapping `usedNonces`. The contract has several functions, including `initialize`, `changeToken`, `emergencyWithdraw`, `pause`, `unpause`, `changeOperator`, `changeAdmin`, and `claimPayment`, which initialize the contract, change the ERC20 token address, allow the admin to withdraw funds in an emergency, pause and unpause the contract, change the operator and admin addresses, and initiate a payment to the caller using a valid signature from the operator address.

2.2. Findings

During the audit process, the audit team found some vulnerability issues in the given version of LS Trade.

2.2.1. [CRITICAL] Logic flaws lead to lost of player's claimable minting reward

Positions:

- `LSTrade.sol#L209`

2.2.1.1.Description

The `claimMinting()` function only calculated `addedReward` once instead of accumulating it. This means that if a player trades on both `bull` and `bear` and the round ends in a draw, they should not be able to claim the entire minting reward.

```
for (uint256 round = periodInfo.roundStart; round <= periodInfo.roundEnd; round++) {
    Struct.RoundInfo memory roundInfo = rounds[round];
    require(roundInfo.timeRoundInfo.startTimestamp != 0, "BullBear-CLMM2: Round has not
started");
    require(block.timestamp > roundInfo.timeRoundInfo.closeTimestamp, "BullBear-CLMM3:
Round has not ended");

    uint256 addedReward = 0;
    if (roundInfo.oracleInfo.oracleCalled && roundInfo.bearCounter != 0 &&
roundInfo.bullCounter != 0) {
        bool bearClaimable = mintingClaimable(round, msg.sender, Enum.Position.Bear);
        bool bullClaimable = mintingClaimable(round, msg.sender, Enum.Position.Bull);
        require(bearClaimable || bullClaimable, "BullBear-CLMM4: Not eligible for claim");
        if (bearClaimable) {
            addedReward = cycles[roundInfo.cycle].amountMintRound / (2 *
roundInfo.userBearCounter);
        }

        if (bullClaimable) {
            addedReward = cycles[roundInfo.cycle].amountMintRound / (2 *
roundInfo.userBullCounter); // INCORRECT
        }
    }

    ledger[round][msg.sender].mintTokenClaimed = true;
    reward += addedReward;
}
```

RECOMMENDATION

The code can be fixed as below:

```
if (bullClaimable) {
    addedReward += cycles[roundInfo.cycle].amountMintRound / (2 *
roundInfo.userBullCounter); // FIXED
}
```

UPDATES

- *Apr 12, 2023*: This issue has been acknowledged and fixed by LS Trade team.

2.2.2. [CRITICAL] Logic flaws lead to lost of player's claimable fund

Positions:

- `LSTrade.sol#L247`

2.2.2.1. Description

The `claim()` function only calculated `addedReward` once instead of accumulating it. This means that if a player trades on both `bull` and `bear` and the round ends in a draw, they should not be able to claim the reward.

```
for (uint256 i = 0; i < epochs.length; i++) {
    Struct.RoundInfo memory roundInfo = rounds[epochs[i]];
    require(roundInfo.timeRoundInfo.startTimestamp != 0, "BullBear-CLM1: Round has not
started");
    require(block.timestamp > roundInfo.timeRoundInfo.closeTimestamp, "BullBear-CLM2: Round
has not ended");
```

```
    uint256 addedReward = 0;
    Struct.BetInfo storage betInfo = ledger[epochs[i]][msg.sender];

    // Round valid, claim rewards
    if (roundInfo.oracleInfo.oracleCalled && roundInfo.bearCounter != 0 &&
roundInfo.bullCounter != 0) {
        bool bearClaimable = claimable(epochs[i], msg.sender, Enum.Position.Bear);
        bool bullClaimable = claimable(epochs[i], msg.sender, Enum.Position.Bull);
        require(bearClaimable || bullClaimable, "BullBear-CLM3: Not eligible for claim");

        if (bearClaimable) {
            addedReward =
                (betInfo.bearCounter * roundInfo.ticketPrice * roundInfo.rewardAmount) /
                roundInfo.rewardBaseCalAmount;
        }

        if (bullClaimable) {
            addedReward = // INCORRECT
                (betInfo.bullCounter * roundInfo.ticketPrice * roundInfo.rewardAmount) /
                roundInfo.rewardBaseCalAmount;
        }
    }
    // Round invalid, refund bet amount
    else {
        require(refundable(epochs[i], msg.sender), "BullBear-CLM4: Not eligible for
refund");
        addedReward = (betInfo.bullCounter + betInfo.bearCounter) * betInfo.ticketPrice;
    }

    betInfo.claimed = true;
    reward += addedReward;

    emit Claim(msg.sender, epochs[i], addedReward);
}

if (reward > 0) {
    payable(msg.sender).transfer(reward);
}
```

RECOMMENDATION

The code can be fixed as below:

```
if (bullClaimable) {
    addedReward += // FIXED
        (betInfo.bullCounter * roundInfo.ticketPrice * roundInfo.rewardAmount) /
        roundInfo.rewardBaseCalAmount;
}
```

UPDATES

- *Apr 12, 2023*: This issue has been acknowledged and fixed by LS Trade team.

2.2.3. [HIGH] Using wrong `ticketPrice` lead to miscalculation claiming reward.

Positions:

- `LSTrade.sol#betBull()`
- `LSTrade.sol#betBear()`

2.2.3.1. Description

It's possible to change the `ticketPrice` for each round in the trading system. Therefore, to avoid any miscalculations when claiming rewards, the order functions should reference `rounds[epoch].ticketPrice` instead of the `ticketPrice` variable. Using a fixed ticket price could lead to inaccurate calculations, which can result in players receiving the wrong amount of reward. By using `rounds[epoch].ticketPrice`, the order functions can ensure that the correct ticket price is used to calculate rewards, regardless of any changes made to the ticket price per round.

Spot the bug in `betBull()`:

```
function betBull(uint256 epoch, uint256 ticketAmount) external payable whenNotPaused
nonReentrant notContract {
    Struct.BetInfo storage betInfo = ledger[epoch][msg.sender];
    uint256 amount = msg.value;
    if (betInfo.bullCounter == 0) {
        rounds[epoch].userBullCounter++;
    }
    if (betInfo.bullCounter + betInfo.bearCounter == 0) {
        betInfo.ticketPrice = ticketPrice; // INCORRECT
        userRounds[msg.sender].push(epoch);
    }

    require(amount >= ticketPrice * ticketAmount, "BullBear-BBU1: Bet amount is less than
ticket price"); // INCORRECT

    // Refund dust amount
    if (amount > ticketPrice * ticketAmount) { // INCORRECT
        uint256 dust = amount.sub(ticketPrice * ticketAmount); // INCORRECT
        payable(msg.sender).transfer(dust);
    }

    require(epoch == currentEpoch, "BullBear-BBU2: Bet is too early/late");
    require(_bettable(epoch), "BullBear-BBU3: Round not bettable");
    require(betInfo.bullCounter + ticketAmount <= maxTicketAmount, "BullBear-BBU4: Out of
bets in round");
```

```
    // Update round data
    Struct.RoundInfo storage round = rounds[epoch];
    round.bullCounter += ticketAmount;

    // Update user data
    betInfo.bullCounter += ticketAmount;

    emit BetBull(
        msg.sender,
        epoch,
        ticketPrice, // INCORRECT
        betInfo.bullCounter,
        round.bearCounter,
        round.bullCounter,
        ticketAmount
    );
}
```

RECOMMENDATION

- Init a local variable, `roundTicketPrice` assign to `rounds[epoch].ticketPrice`
- Replace `ticketPrice` with `roundTicketPrice` in function

UPDATES

- *Apr 12, 2023*: This issue has been acknowledged and fixed by LS Trade team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 12, 2023*	Public Report	Verichains Lab

Table 2. Report versions history