BEOSIN
Blockchain Security

Surf Protocol V2

Smart Contract Security Audit

No. 202403281200

Mar 28th, 2024

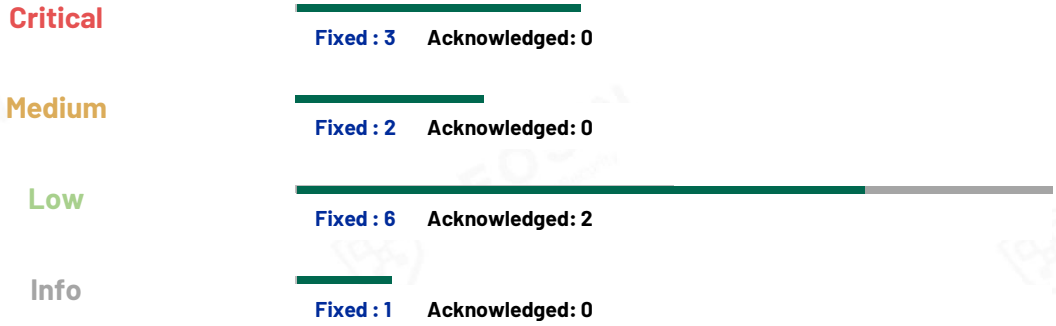
SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

After auditing, 3 Critical, 2 Medium, 8 Low and 1 Info items were identified in the Surf Protocol V2 project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:



- **Risk Description:**

1. Due to the partial fix of Surf Protocol V2-08, if the leakage of keeper will possibly lead to the loss of contract funds, please keep the private key properly by the project side.
2. When the pool experiences significant losses, the `calcNetValue` function may calculate a zero net value, which subsequently prevents users from adding liquidity.

- **Project Description:**

1. Business overview:

Surf Protocol is a decentralized trading protocol specifically designed for on-chain users, offering leveraged trading services. In this protocol, liquidity providers serve as counterparties to all traders, creating a unique trading ecosystem. When traders incur losses due to operational mistakes or market fluctuations, liquidity providers stand to gain; conversely, if traders profit, liquidity providers may face losses. This mechanism ensures market balance and trading fairness.

Surf Protocol has established various fee types, with the flow of position fees being particularly unique. It is determined solely based on the comparison of the number of bulls and bears in the market. When the number of bulls exceeds the number of bears, position fees are paid by the bulls to the bears; conversely, if the number of bears is greater than the number of bulls, position fees are paid by the bears to the bulls. This design not only helps balance market supply and demand but also incentivizes traders to participate in trading more rationally.

Furthermore, SURF Protocol has introduced an innovative referrer mechanism. Users can enjoy certain transaction discounts simply by entering the referrer code during trading. This not only reduces users' transaction costs and enhances their trading experience but also provides a source of income for the owner of the referrer code, who receives a portion of the user's transaction fees as a reward. This mechanism effectively promotes mutual assistance and cooperation among users, driving the healthy development of the SURF Protocol community.

1 Overview

1.1 Project Overview

Project Name	Surf Protocol V2	
Project language	Solidity	
Platform	Merlin	
Codebase	https://github.com/surf-exchange/v2-core https://github.com/surf-exchange/v2-core/tree/collateral	
Commit Hash	0c54145e058303dea75187a97c48763369bf793a (Initial in main) 0e8d30f1c2e409d16aa80f083ea48e108ff0d63e c1fe207ad3a2852390ae56e82a78ee6f91458749 47ae9e1ccb743c2e76f14bc906ca0b6e1464695b cf2a8e50513178cd18aabc1a70022799a9f99369 9dc99f947d6849af0b49e7780ce04bd09e274510 db99a214d9de2c92b2a39282b2a80d91fcf1c636	
Contract address	swapAdapter	0x453eCD4B0De64C26a0390007bAFC8373ff7b592E
	oracle	0x7Caef04019317eFBE50dD0C58e61739736527B6b
	PairConfig	0xF27F0E1B24271Ffd2C5F894513D8B804B84550f9
	LpTokenWBTC	0xa07c05536608A58236bae5e6a9790179eeE80053
	LpTokenMBTC	0x8af2A32dFF890FE09Af1006F2dF19789022D66d3
	lpConfig	0x7dE83F326eE73Fd625F5F98a271f9B63E0c4c8e5
	Vault	0x5FDF2c442D209c9EE95D6Cf64DD13314E7C5e5B1
	LpManager	0x583d7560CD23aB20817c0a520b6a924FCaa3A4cf
	PoolManager	0x613E2c64a5B6Dda3d0f627fEAD2aEa91dE98A1fe
	positionBaseContract	0x1E13428D9DC75ec783c21ea98eaA5c861cEF62ed
	PositionManager	0x613cF8060D1a01411CB1A79c8F9E2c8e295c9077
	PositionFeeContract	0xB328D8645D6D75e4e4cA2beaC2e66bcF77B4B8A5
	OrderBook	0x9F6BF14316fF796e351B37B00CFCd191B1A64208
	OrderBookStorage	0x17d68ed5B16C31EDa4def83F3d688b17f48ECF1A

Contract address	OrderBookBase	0xEE694A4fe74F2923b3c967761B4a6A5e40d9ce29
	Callbacks	0x7cFd2162efeE302D12D46eEb5635Ce894c7216bd
	Referral	0x403a63f077B42B100CAEdad174a909E9955abeD9
	LpManagerReader	0x4F8D8AB8bD5D17C8Ad7688Ec94777e8b08fC0c32
	OrderBookReader	0x6C3CB48a625053353d6B5753528D1E44D9D847ad
	PoolManagerReader	0x5E42aeB542d0a2e4eB8aEBE5B58638EbBD7e68e8
	PositionReader	0x1c06C77eA6255e144941AFA71187B4f22ec5ecA1

1.2 Audit Overview

Audit work duration: Mar 11, 2024 – Mar 28, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's library of security expert rules. These rules call into the contracts under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

2. Manual Review

Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's business logic.

The manual audit is divided into three groups to cover the entire auditing process:

The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Findings

Index	Risk description	Severity level	Status
Surf Protocol V2-01	Lack of checks on the actual number of transfers	Critical	Fixed
Surf Protocol V2-02	Lack of updated fundingFeePerSizeForShort	Critical	Fixed
Surf Protocol V2-03	Incorrect use of prior position for liquidation pricing	Critical	Fixed
Surf Protocol V2-04	Setting up referrer logic errors	Medium	Fixed
Surf Protocol V2-05	Referral contracts will not be available	Medium	Fixed
Surf Protocol V2-06	Object error	Low	Fixed
Surf Protocol V2-07	The getAllTokensWeights function is poorly designed	Low	Fixed
Surf Protocol V2-08	Flawed signature mechanism	Low	Partially Fixed
Surf Protocol V2-09	The _executeRemoveLiquidityRequest function is poorly designed	Low	Fixed
Surf Protocol V2-10	No judgment logic exists for the input of executionFee	Low	Acknowledged
Surf Protocol V2-11	Missing latestExecuteTimestamp assignment function	Low	Fixed
Surf Protocol V2-12	Deletion of tokens is not possible	Low	Acknowledged
Surf Protocol V2-13	Missing ids to compare with requestIds for judgment	Low	Fixed
Surf Protocol V2-14	Incorrect event triggering	Info	Fixed

Finding Details:

[Surf Protocol V2-01] Lack of checks on the actual number of transfers

Severity Level	**Critical**
Type	Business Security
Lines	LpManager.sol #L124-131
Description	In the `addLiquidityRequest` function of the LpManager contract, it is designed to increase liquidity. This function offers two methods for adding liquidity: using warp tokens or non-warp tokens. However, when adding liquidity using warp tokens, the function does not verify whether the actual amount of tokens transferred by the user (msg.value) matches the quantity entered by the user. This could potentially result in a situation where a user transfers a small amount of funds (for example, 1 USD) but enters a significantly larger quantity (such as 1 million USD). Subsequently, if the user cancels the liquidity addition operation, they would be able to withdraw a much larger amount of funds than they had actually transferred (in this case, 1 million USD).
Recommendation	It is recommended to verify whether the entered quantity matches the actual amount of tokens transferred.
Status	**Fixed.**

```
        if (_request.shouldWrap) {
            address wrapper = config.wrapperAddr();
            require(msg.value > 0, "LpManager: Invalid request amount");
            require(
                _request.indexToken == config.wrapperAddr(),
                "LpManager:mismatch wrapper address error"
            );
            IWETH(wrapper).deposit{value: msg.value}();
```

```
        if (_request.shouldWrap) {
            address wrapper = config.wrapperAddr();
            require(
                msg.value == _request.amount,
                "LpManager: Invalid request amount"
            );
            IWETH(wrapper).deposit{value: msg.value}();
```

[Surf Protocol V2-02] Lack of updated fundingFeePerSizeForShort

Severity Level	**Critical**
Type	Business Security
Lines	PositionFee.sol #L142-158
Description	In the `updateUserFundingState` function, when updating the PerSize of a user's position, the function only updates the PerSize for the current position type and fails to synchronize the update to the counterparty's PerSize value. To illustrate with a specific scenario, let's assume that the current pool status indicates a higher number of long positions than short positions. User A performs a long operation worth 1 USDT when the `fundingFeeAmountPerSize` for long positions and the `claimableTokenAmountPerSize` for short positions are both set at 100. After one hundred days, the `fundingFeeAmountPerSize` for long positions has increased to 10,000, while the `claimableTokenAmountPerSize` for short positions has changed to 8,000. If User A then performs a short operation worth 100 million USDT, the system will only settle the funding fees for User A's previous 1U long position and update the `fundingFeeAmountPerSize` for the long position. However, the `claimableTokenAmountPerSize` for the short position remains unchanged at 100. Subsequently, if User A withdraws the 100 million short position, they will be able to directly claim the 100 million short position reward calculated over 100 days, thus exploiting this vulnerability for profit.

```
        if (position.isLong) {
            userInfo
                .fundingFeePerSizeForLong
                .lastUpdateFundingFeeAmountPerSize =
fundingFeeAmountPerSize;
            userInfo
                .fundingFeePerSizeForLong
                .lastUpdateTokenClaimableFundingAmountPerSize =
claimableTokenAmountPerSize;
        } else {
            userInfo
                .fundingFeePerSizeForShort
                .lastUpdateFundingFeeAmountPerSize =
fundingFeeAmountPerSize;
```

```
            userInfo
                .fundingFeePerSizeForShort
                .lastUpdateTokenClaimableFundingAmountPerSize =
claimableTokenAmountPerSize;
        }
```

Recommendation It is recommended to update all types of PerSize for a user when updating their Funding status.

Status **Fixed.**

```
        userInfo
            .fundingFeePerSizeForLong
            .lastUpdateFundingFeeAmountPerSize =
fundingFeeAmountPerSizeForLong;
        userInfo
            .fundingFeePerSizeForLong
            .lastUpdateTokenClaimableFundingAmountPerSize =
claimableTokenAmountPerSizeForLong;
        userInfo
            .fundingFeePerSizeForShort
            .lastUpdateFundingFeeAmountPerSize =
fundingFeeAmountPerSizeForShort;
        userInfo
            .fundingFeePerSizeForShort
            .lastUpdateTokenClaimableFundingAmountPerSize =
claimableTokenAmountPerSizeForShort;
```

[Surf Protocol V2-03] Incorrect use of prior position for liquidation pricing

Severity Level	**Critical**
Type	Business Security
Lines	PositionBase.sol#L237-262
Description	In the `decreaseCollateral` function of the PositionBase contract, there exists a prominent error: the function mistakenly uses the collateral value from the passed-in _position when calculating the liquidation price, instead of employing the updated collateral value that should be used. This flaw allows attackers to engage in counterparty trading, utilizing one position for profit while maintaining another position that is perilously close to liquidation. Despite this, the attacker can still withdraw the collateral, thus evading losses.
Recommendation	It is recommended to use the updated collateral to calculate the liquidation

```
    function decreaseCollateral(
        IPositionBase.Position calldata _position,
        uint _purchaseTokenAmount,
        uint _price
    ) external view returns (uint newCollateral, uint newLeverage) {
        IPositionBase.Position memory userPos = _position;
        require(
            _purchaseTokenAmount < userPos.collateral,
            "PosBase: Decrease amount exceeds collateral"
        );
        newCollateral = userPos.collateral - _purchaseTokenAmount;
        newLeverage = (userPos.size * FEE_P_BASE) / newCollateral;
        uint maxLeverage = config.maxMaxLeverage() * FEE_P_BASE;
        require(newLeverage <= maxLeverage, "PosBase: Leverage too
high");
        // Calculate the liquidation price
        uint liqPrice = _getLiqPrice(_position, 0, 0);
        // Check if the position would be liquidated at the given price
        bool isLiq = (userPos.isLong && _price <= liqPrice) ||
            (!userPos.isLong && _price >= liqPrice);
        require(!isLiq, "PosBase: Liquidation condition met");
        return (newCollateral, newLeverage);
    }
```

price.

Status **Fixed.**

```
    function decreaseCollateral(
        IPositionBase.Position calldata _position,
        uint _purchaseTokenAmount,
        uint _price
    ) external view returns (uint newCollateral, uint newLeverage) {
        IPositionBase.Position memory userPos = _position;
        require(
            _purchaseTokenAmount < userPos.collateral,
            "PosBase: Decrease amount exceeds collateral"
        );
        newCollateral = userPos.collateral - _purchaseTokenAmount;
        newLeverage = (userPos.size * FEE_P_BASE) / newCollateral;
        uint maxLeverage = config.maxMaxLeverage() * FEE_P_BASE;
        require(newLeverage <= maxLeverage, "PosBase: Leverage too
high");
        userPos.collateral = newCollateral;
        userPos.leverage = newLeverage;
        // Calculate the liquidation price
        (uint beforeLiqPrice, ) = _getLiqPriceAndHoldFee(_position,
_price);
        (uint afterLiqPrice, ) = _getLiqPriceAndHoldFee(userPos,
_price);
        // Check if the position would be liquidated at the given price
        bool isLiqBefore = (userPos.isLong && _price <= beforeLiqPrice)
||
            (!userPos.isLong && _price >= beforeLiqPrice);
        bool isLiqAfter = (userPos.isLong && _price <= afterLiqPrice) ||
            (!userPos.isLong && _price >= afterLiqPrice);
        require(
            !isLiqBefore && !isLiqAfter,
            "PosBase: Liquidation condition met"
        );
        return (newCollateral, newLeverage);
    }
```

[Surf Protocol V2-04] Setting up referrer logic errors

Severity Level	**Medium**
Type	Business Security
Lines	Referral.sol#L328-344
Description	In the `setTraderReferralCodeByUser` function of the Referral contract, it is essential to ensure that the user's isReferred flag is set to false before proceeding, to prevent the traderReferred value from being erroneously reset to 0 due to the user repeatedly entering the same _code. This can lead to issues where other users are unable to change their referral codes. Similarly, attention should be paid to this matter in the `setTraderReferralCode` function as well.

```
    function setTraderReferralCodeByUser(string memory _code) external
{
        (string memory originCode, address originReferrer, , ) =
getTraderInfo(
            msg.sender
        );
        address codeReferrer = codeOwners[_code];
        require(msg.sender != codeReferrer, "cannot be yourself");
        require(codeReferrer != address(0), "not registered");
        if (bytes(originCode).length != 0 && originReferrer !=
address(0)) {
            traderReferred[originReferrer] -= 1;
        }
        if (!isReferred[codeReferrer][msg.sender]) {
            traderReferred[codeReferrer] += 1;
            isReferred[codeReferrer][msg.sender] = true;
        }
        _setTraderReferralCode(msg.sender, _code);
    }
```

Recommendation	It is recommended to adding "isReferred[originReferrer][msg.sender] = false;" to this function.
Status	**Fixed.**

```
    function setTraderReferralCodeByUser(string memory _code) external
{
        (string memory originCode, address originReferrer, , ) =
getTraderInfo(
```

```
            msg.sender
        );
        address codeReferrer = codeOwners[_code];
        require(msg.sender != codeReferrer, "cannot be yourself");
        require(codeReferrer != address(0), "not registered");
        require(!isReferred[codeReferrer][msg.sender], "has invited");
        if (bytes(originCode).length != 0 && originReferrer !=
address(0)) {
            traderReferred[originReferrer] -= 1;
            isReferred[originReferrer][msg.sender] = false;
        }
        traderReferred[codeReferrer] += 1;
        isReferred[codeReferrer][msg.sender] = true;
        _setTraderReferralCode(msg.sender, _code);
    }
```

[Surf Protocol V2-05] Referral contracts will not be available

Severity Level	Medium
Type	Business Security
Lines	Referral.sol#L400
Description	As the invocation of the `setTraderReferralCode` function relies on the onlyOwner permission, the contract is unable to directly call the function, thereby leading to a failed function call.

```
    function setTraderReferralCode(
        address _account,
        string memory _code
    ) external override onlyOwner {
        (string memory originCode, address originReferrer, , ) =
getTraderInfo(
            _account
        );
```

Recommendation	It is recommended to replace onlyOwner permissions with other permissions.
Status	**Fixed.**

```
    function setTraderReferralCode(
        address _account,
        string memory _code
    ) external override onlyHandler {
        (string memory originCode, address originReferrer, , ) =
getTraderInfo(
            _account
        );
```

[Surf Protocol V2-06] Object error

Severity Level	Low
Type	Business Security
Lines	Callbacks.sol#L316
Description	In the `_openTradeMarketCallback` function, the information of the keeper was mistakenly recorded using o.trader, which does not comply with the expected data recording method.

```
            uint positionSize = o.order.purchaseTokenAmount *
o.order.leverage;
            IPositionManager.UpdatePositionResp memory resp;
            IVault.TriggeredLimitId memory triggerId =
IVault.TriggeredLimitId(
                o.trader,
                o.trader,
                o.order.spotTokenKey,
                o.index,
                IOrderBook.KeeperOrder.MARKET
            );
```

Recommendation	It is recommended to changing one of TriggeredLimitId's o.trader to n.keeper.
Status	**Fixed.**

```
            uint positionSize = o.order.purchaseTokenAmount *
o.order.leverage;
            IPositionManager.UpdatePositionResp memory resp;
            IVault.TriggeredLimitId memory triggerId =
IVault.TriggeredLimitId(
                n.keeper,
                o.trader,
                o.order.spotTokenKey,
                o.index,
                IOrderBook.KeeperOrder.MARKET
            );
```

[Surf Protocol V2-07] The getAllTokensWeights function is poorly designed

Severity Level	Low
Type	Business Security
Lines	Referral.sol#L118
Description	In the `getAllTokensWeights` function of the LpConfig contract, although all token weights are retrieved in bulk, the status of each token is checked to ensure it is active. If any one of the tokens is inactive, the entire function cannot be called.

```
function getAllTokensWeights()
    external
    view
    returns (bytes32[] memory, uint256[] memory)
{
    uint256 numTokens = tokens.length;
    bytes32[] memory spotTokenKeys = new bytes32[](numTokens);
    uint256[] memory tokenWeights = new uint256[](numTokens);
    for (uint256 i = 0; i < numTokens; i++) {
        Token memory t = token[tokens[i]];
        require(t.isActive, "Token is not active");
        spotTokenKeys[i] = t.spotTokenKey;
        tokenWeights[i] = t.tokenWeight;
    }
    return (spotTokenKeys, tokenWeights);
}
```

Recommendation	It is recommended to assign a null value to inactive tokens, ensuring that the function can still be normally invoked and return results even when encountering inactive tokens.
Status	**Fixed.**

```
function getAllTokensWeights()
    external
    view
    returns (bytes32[] memory, uint256[] memory)
{
    uint256 numTokens = tokens.length;
    bytes32[] memory spotTokenKeys = new bytes32[](numTokens);
```

```
        uint256[] memory tokenWeights = new uint256[](numTokens);
        uint256 ws;
        for (uint256 i = 0; i < numTokens; i++) {
            Token memory t = token[tokens[i]];
            spotTokenKeys[i] = t.spotTokenKey;
            tokenWeights[i] = t.tokenWeight;
            ws += t.tokenWeight;
        }
        require(ws == 1 * BASIS_POINTS_DIVISOR, "LpCfg:invalid
weights");
        return (spotTokenKeys, tokenWeights);
    }
```

[Surf Protocol V2-08] Flawed signature mechanism

Severity Level	Low
Type	Business Security
Lines	Utils.sol#L18-27
Description	In the `verifySignature` function of the Utils.sol file, we found that the crucial line of code require(signer == signatureKey, "Price verify fail");, which was originally used to verify the identity of the signer, has been commented out. This line is essential for ensuring that the function performs its signature verification function correctly, so it is recommended to uncomment this line of code to restore its verification function. Additionally, when examining the current computation of messageHash, we noticed that it does not include complete signature information, such as tk and tp, among other key components. The absence of these information elements may affect the integrity and accuracy of the signature verification.

```
    function verifySignature(
        Price memory price,
        address signatureKey
    ) internal pure {
        bytes memory signature = bytes.concat(price.r, price.s,
price.v);
        bytes32 messageHash = keccak256(abi.encode(price.p,
price.tm, price.n));
        address signer = ECDSA.recover(messageHash, signature);
        // require(signer == signatureKey, "Price verify fail");
    }
```

Recommendation	It is recommended to include tk and tp data in the calculation of messageHash. Furthermore, to enhance the reliability of signatures, it is advisable to incorporate verification of signature timeliness and implement mechanisms to prevent the reuse of signatures.
Status	**Partially Fixed.**

```
    function verifySignature(
        Price memory price,
        address signatureKey
    ) internal pure {
        bytes memory signature = bytes.concat(price.r, price.s,
```

```
price.v);
        bytes32 messageHash = keccak256(abi.encode(price.p, price.tm,
price.n));
        address signer = ECDSA.recover(messageHash, signature);
        require(signer == signatureKey, "Price verify fail");
    }
```

[Surf Protocol V2-09] The _executeRemoveLiquidityRequest function is poorly designed

Severity Level	Low
Type	Business Security
Lines	LpManager.sol#L360-411
Description	In the `_executeRemoveLiquidityRequest` function, the 0th order is processed first. If the order fails to meet the conditions for maximum liquidity removal, despite the corresponding decrease in _taskAmount, the related request remains unprocessed and unremoved, remaining in the 0th position, thereby blocking the execution of subsequent orders.

```
    function _executeRemoveLiquidityRequest(uint256 _taskAmount)
private {

        _taskAmount = _taskAmount > allRemoveLiquidity.length

            ? allRemoveLiquidity.length

            : _taskAmount;

        while (_taskAmount > 0) {

            RemoveLiquidity memory r = allRemoveLiquidity[0];

        .... ....

            _taskAmount--;

        }

    }
```

Recommendation	It is recommended to use a for loop to execute the order from front to back.
Status	**Fixed.**

```
    function _executeRemoveLiquidityRequest(
        IPoolManager poolManager,
        uint256 _taskAmount,
        Utils.Price[] memory prices
    ) private {
        _taskAmount = _taskAmount > allRemoveLiquidity.length

            ? allRemoveLiquidity.length

            : _taskAmount;
```

```
        for (uint index = 0; index < _taskAmount; index++) {
            RemoveLiquidity memory r = allRemoveLiquidity[0];
            .... ....
            poolManager.unlock(r.request.lpToken, r.request.amount);
            _manageLiquidityRequest(
                r.request.account,
                r.id,
                LiquidityActionType.Remove
            );
        }
    }
```

[Surf Protocol V2-10] No judgment logic exists for the input of executionFee

Severity Level	Low
Type	Business Security
Lines	OrderBook.sol#108-119

Description	In the orderBook contract, there is a lack of judgment logic for the user's input of executionFee, which may allow users to execute operations without paying transaction fees.
	`function openLimitOrder(`
	`LimitOrder calldata _order`
	`) external payable nonReentrant notContract whenNotPaused {`
	`IOrderBookStorage orderBookStorage = IOrderBookStorage(`
	`config.orderBookStorage()`
	`);`
	`address collateralToken = ILpConfig(config.lpConfig())`
	`.getToken(_order.spotTokenKey)`
	`.collateralToken;`
	`_transferInETH(_order.executionFee);`

Recommendation	It is recommended to incorporate appropriate judgment mechanisms to guarantee the reasonableness of fees paid by users for executing operations in the orderBook contract.
Status	**Acknowledged.**The project team is internally aware that executeFee is currently not in use, and the contract will be configured with a value of 0 while keeping the field reserved for future use.

[Surf Protocol V2-11] Missing latestExecuteTimestamp assignment function

Severity Level	Low
Type	Business Security
Lines	OrderBook.sol#302
Description	In the contract, there is no function to assign a value to the latestExecuteTimestamp parameter, which will result in the parameter defaulting to zero.

```
    function executeRequests(
        ExecuteRequests calldata _request
    ) external returns (bool) {
        require(pairConfig.isKeeperApproved(msg.sender),
"NOT_APPROVED_KEEPER");
        require(!executeStatus[_request.taskId], "LpManager: invalid
task id");
        require(
            block.timestamp - latestExecuteTimestamp >
config.maxDelay(),
            "LpManager: frequently execute"
        );
```

Recommendation	It is recommended to add a function to assign a value to the latestExecuteTimestamp parameter.
Status	**Fixed.**

```
        latestExecuteTimestamp = block.timestamp;
        emit ExecuteBatchRequestsEvent(
            _request,
            executeRemoveStatus[_request.removeId]
        );
        return true;
```

[Surf Protocol V2-12] Deletion of tokens is not possible

Severity Level	Low
Type	Business Security
Lines	OrderBook.sol#28-48
Description	In the `setTokenConfig` function of the LpConfig contract, there is only a push operation on tokens, and there is a lack of a corresponding pop operation, so it is not possible to realize the deletion of tokens.

```
    function setTokenConfig(Token[] calldata _tokens) external
onlyOwner {
        for (uint i = 0; i < _tokens.length; i++) {
            Token storage currentToken =
token[_tokens[i].spotTokenKey];
            currentToken.tokenIndex = tokenIndex;
            currentToken.spotTokenKey = _tokens[i].spotTokenKey;
            currentToken.indexToken = _tokens[i].indexToken;
            currentToken.lpToken = _tokens[i].lpToken;
            currentToken.collateralToken = _tokens[i].collateralToken;
            currentToken.tokenWeight = _tokens[i].tokenWeight;
            currentToken.maxGlobalShortSize =
_tokens[i].maxGlobalShortSize;
            currentToken.maxGlobalLongSize =
_tokens[i].maxGlobalLongSize;
            currentToken.maxDailyRemoveSize =
_tokens[i].maxDailyRemoveSize;
            currentToken.isStable = _tokens[i].isStable;
            currentToken.isActive = _tokens[i].isActive;
            if (_tokens[i].indexToken == address(0)) {
                currentToken.isNative = true;
            }
            tokens.push(_tokens[i].spotTokenKey);
            tokenIndex++;
        }
    }
```

Recommendation	It is recommended to add a "pop" function to enable the deletion of tokens. This function can be implemented to remove a specific token from the list of tokens, based on its position or some other identifying criteria. By incorporating the

	pop feature, users will have greater flexibility and control over managing and manipulating their token collections.
Status	**Acknowledged.** It is known internally on the project side that if a token wants to be deleted, it will set the status to inActive.

[Surf Protocol V2-13] Missing ids to compare with requestIds for judgment

Severity Level	Low
Type	Business Security
Lines	LpManager.sol#734-739

Description	In the removal of liquidity function within the `_updateLiquidityRequestStorage` function, if there is no prior comparison and judgment between the passed-in id and the requestIds, and data processing is carried out directly, it may lead to subsequent failure in the operation of removing liquidity.
	`uint[] storage requestIds = pendingRemoveLiquidityIds[_account];`
	`for (uint i = 0; i < requestIds.length; i++) {`
	`removeLiquidityCount[_account]--;`
	`requestIds[i] = requestIds[requestIds.length - 1];`
	`requestIds.pop();`
	`}`

Recommendation	It is recommended to first compare and judge the passed-in id with the requestIds before proceeding with data processing.

Status	**Fixed.**
	`for (uint i = 0; i < requestIds.length; i++) {`
	`if (requestIds[i] == _id) {`
	`removeLiquidityCount[_account]--;`
	`requestIds[i] = requestIds[requestIds.length - 1];`
	`requestIds.pop();`
	`uint id = removeLiquidityIds[_account][_id];`
	`allRemoveLiquidity[id] = allRemoveLiquidity[`
	`allRemoveLiquidity.length - 1`
	`];`
	`removeLiquidityIds[allRemoveLiquidity[id].request.account][`
	`allRemoveLiquidity[id].id`
	`] = id;`
	`delete removeLiquidityIds[_account][_id];`
	`allRemoveLiquidity.pop();`
	`}`
	`}`

[Surf Protocol V2-14] Incorrect event triggering

Severity Level	Info
Type	Coding Conventions
Lines	Callbacks.sol#L310
Description	In the `_openTradeMarketCallback` function, there is an event triggering error.

```
        if (shouldCancel) {
            if (o.order.buy) {
                position.withdrawFund(
                    _a.orderId,
                    true,
                    o.collateralToken,
                    o.trader,
                    o.order.purchaseTokenAmount + o.order.tradeFee
                );
                //orderBook.marketCancellationRefund(_a.orderId);
            }
            orderBookStorage.unregisterPendingMarketOrder(_a.orderId)
;
            emit MarketOpenCanceled(_a.orderId, o, _a.price);
```

Recommendation	The _a.orderId should be changed to o.index to ensure the correct event is triggered.
Status	**Fixed.**

```
        if (shouldCancel) {
            if (o.order.buy) {
                position.withdrawFund(
                    o.index,
                    true,
                    o.collateralToken,
                    o.trader,
                    o.order.purchaseTokenAmount + o.order.tradeFee
                );
            }
            orderBookStorage.unregisterPendingMarketOrder(o.index);
            emit MarketOpenCanceled(o.index, o, _a.price);
```

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1(Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.3 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.4 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Deprecated Items
		Redundant Code
		Default Values
2	General Vulnerability	Insufficient Address Validation
		Lack Of Address Normalization
		Variable Override
		DoS (Denial Of Service)
		Function Call Permissions
		Returned Value Security
		Mathematical Risk
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Arbitrage Attack
		Access Control

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Rust language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com