BEOSIN
Blockchain Security

Surf protocol

Smart Contract Security Audit

No. 202311080924

Nov 08th, 2023

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

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

Medium	Fixed : 1 Acknowledged: 1
Low	Fixed: 2
Info	Fixed: 2

- **Risk Description:**

(1) The default maximum leverage for the perpetual contract is set at 50x. The Owner has the authority to update the maximum leverage. Excessive leverage may lead to issues such as delayed liquidations.

(2) There is relatively limited precision expansion within the contract. When the Owner configures the base parameters, it is essential to consider precision issues to prevent potential losses due to precision loss.

- **Project Description:**

Business overview

Surf protocol project is a perpetual contract project derived from a collateral pool and consists of three main components: the Collateral Pool Contract, Perpetual Contract, and Order Contract.

In the Collateral Pool Contract, users can create collateral pools with different fee rates and provide liquidity to these pools. After collateral is provided, users of the Perpetual Contract can open positions within these collateralized pools, and the losses incurred by Perpetual Contract users become the source of profit for the collateral pool participants. When users increase or decrease their positions, a portion of the trading fees will also be allocated as rewards to the top contributors and initiators.

Within the Perpetual Contract, users can open and close positions in different collateral pools. Currently, the default maximum leverage is set at 50x, and the total position size for opening cannot exceed the total liquidity within the collateral pool. All actions related to opening and closing positions need to be conducted through the higher-level Order Contract.

The Order Contract manages the logic for opening and closing positions, take profit, stop loss, and liquidation. When performing these actions, users and keepers need to initiate the corresponding request events to proceed with real-time weighted price retrieval through an oracle. When handling market orders, if the user creates a market order and the retrieved price doesn't slip beyond the specified threshold, the contract automatically executes the corresponding opening or closing operation. For limit orders, users must create the order, and a keeper is required to assess and confirm the order. After keeper approval, users can proceed with subsequent actions like opening or closing positions, but they are required to pay a fee to the keeper for their services.

1 Overview

1.1 Project Overview

Project Name	Surf protocol
Project language	Solidity
Platform	Base chain
GitHub	https://github.com/surf-exchange/v1-core
Commit	ef6dad964ab746533ab1cb86ec299497d35eef16 56a075f90d2b92f0f974cfef5b411230cbd0995d 93b36785b8a30b9305062ba9d1ba7b3ff9707e51 6ef9938281365b83cad248d340a87fc115248af1 8d5de477dd233e30b83be13393fd49dadd1e714a

1.2 Audit Overview

Audit work duration: Oct 8, 2023 – Nov 8, 2023

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-01	Centralization Risk	Medium	Acknowledged
Surf protocol-02	Gas Blocking Risk	Medium	Fixed
Surf protocol-03	Unreasonable Slippage Setting	Low	Fixed
Surf protocol-04	Fees Lack Approval	Low	Fixed
Surf protocol-05	Incorrect Parameter Usage	Info	Fixed
Surf protocol-06	Redundant Code	Info	Fixed

Finding Details:

[Surf protocol-01] Centralization Risk

Severity Level	Medium
Type	Business Security
Lines	OrderBook.sol #L335-380 OrderBookBase.sol #L337-381

Description	In the OrderBook contract, the `executeOrder` function for handling limit orders allows manual input of order information by the keeper address. This order can be maliciously forged under the control of a malicious keeper, and such a forged order may lead to users purchasing positions at non-ideal prices.
	`executeOrder`:
	`function executeOrder(`
	`    ExecuteOrder calldata _order`
	`) external nonReentrant notContract whenNotPaused {`
	`    address keeper = msg.sender;`
	`    require(config.isKeeperApproved(msg.sender),`
	`"NOT_APPROVED_KEEPER");`
	`    IOrderBookBase(config.orderBookBase()).verifyExecuteOrder(`
	`        config,`
	`        _order,`
	`        keeper`
	`    );`
	`_canExecute`:
	`if (o.order.isLong) {`
	`    return o.order.triggerPrice >= _order.price;`
	`} else {`
	`    return o.order.triggerPrice <= _order.price;`
	`}`

Recommendation	It is recommended to manage keeper addresses using multi-signature wallets or similar methods.
Status	**Acknowledged.**

[Surf protocol-02] Gas Blocking Risk

Severity Level	**Medium**
Type	Business Security
Lines	Award.sol #L674-677
Description	In the Award contract, the `_findTopContributor` function, when dealing with invalid contributors, directly removes the corresponding data from the contributors array without adjusting the array's length. This can result in the contributors array length remaining the same, even after performing a delete operation. Additionally, new contributors are added to the end of the array, causing the array to grow longer over time. This can lead to increased gas consumption when querying the array. Furthermore, since the `_findTopContributor` function is called by the upper-level function `recordContributor` during liquidity addition and removal, there is a risk of gas blocking. Malicious attackers might exploit this to repeatedly add invalid contributors to a pool, potentially leading to a DOS attack.

```
    function _findTopContributor(
        address _pool,
        uint _minimum
    ) private returns (address, uint) {
        address[] storage contributors = poolAllContributors[_pool];
        address topContributor = address(0);
        uint topContribution = 0;
        for (uint i = 0; i < contributors.length; i++) {
            address contributor = contributors[i];
            uint contribution = IERC20(_pool).balanceOf(contributor);
            if (_calcTvl(contribution, _pool, contributor) < _minimum)
{
                delete contributors[i];
                continue;
            }
            if (contribution > topContribution) {
                topContributor = contributor;
                topContribution = contribution;
            }
        }
        return (topContributor, topContribution);}
```

Recommendation	It is recommended to use the approach inside the `_removeContributor` function to pop invalid contributors from the array.
Status	**Fixed.** The project team has made modifications to the `_findTopContributor` function and included the `_removeContributor` function in the logic.

`_findTopContributor:`

```
            if (_calcTvl(contribution, _pool, contributor) < _minimum)
{
                _scanContributor(_pool, contributor, _minimum);
                continue;
            }
```

`_scanContributor:`

```
        uint contribution = IERC20(_pool).balanceOf(last);
        if (_calcTvl(contribution, _pool, last) >= _minimum) {
            _removeContributor(_pool, _trader);
        } else {
            _removeContributor(_pool, last);
        }
```

`_removeContributor:`

```
        if (last != _trader) {
            uint index = contributorsIndex[_pool][_trader];
            contributorsIndex[_pool][last] = index;
            poolAllContributors[_pool][index] = last;
        }
        poolAllContributors[_pool].pop();
        delete contributorsIndex[_pool][_trader];
    }
```

[Surf protocol-03] Unreasonable Slippage Setting

Severity Level	Low
Type	Business Security
Lines	Callbacks.sol #L242-247
Description	In the Callbacks contract, the `_openTradeMarketCallback` function, when handling slippage for market orders, only considers slippage during position addition and does not take into account slippage evaluation for position reduction. This results in a reversed slippage logic. For instance, when a user attempts to reduce a long position, the oracle-obtained answer price needs to be greater than `wantedPrice` plus the slippage price in the current slippage logic to execute the reduction. This contradicts the actual slippage logic.

```
        bool shouldCancel = _a.price == 0 ||
            (
                o.order.isLong
                    ? price > o.order.wantedPrice + maxSlippage
                    : price < o.order.wantedPrice - maxSlippage
            );
```

Recommendation	It is recommended to separately evaluate slippage for position reduction.
Status	**Fixed.** The project team has modified the corresponding slippage logic.

```
if (!shouldCancel) {
    if (o.order.isLong) {
        if (o.order.buy) {
            shouldCancel = price > o.order.wantedPrice + maxSlippage;
        } else {
            shouldCancel = price < o.order.wantedPrice - maxSlippage;
        }
    } else {
        if (o.order.buy) {
            shouldCancel = price < o.order.wantedPrice - maxSlippage;
        } else {
            shouldCancel = price > o.order.wantedPrice + maxSlippage;
         }}}
```

[Surf protocol-04] Fees Lack Approval

Severity Level	Low
Type	Business Security
Lines	PoolRouter.sol #L648

Description

In the PoolRouter contract, the _executeLiquidity function, when liqFee is present, sends the removeFee from the sender's address to the vault. However, according to the higher-level logic, it has been observed that the contract does not check the approval value of the sender for the PoolRouter contract. If users do not approve in a timely manner, this may lead to the Oracle callback not executing correctly, resulting in callback failures.

```
        if (liqFee > 0) {
            uint256 removeFee =
what.mul(liqFee).div(BASIS_POINTS_DIVISOR);
            IERC20(baseToken).transferFrom(sender, vault, removeFee);
            IVault(vault).distributeFeeRewardWithoutId(
                removeFee,
                baseToken,
                IVault.RewardType.RemoveFee
            );
        }
```

Recommendation

It is recommended to handle this portion of the fees in the BurnFrom rewards, instead of additional approve transfers.

Status

Fixed. The project team covered this fee as part of the BurnFrom reward.

```
        (address to, uint256 withdrawFee) = IPoolRouter(
            _getPairConfig().poolRouter()
        ).beforeBurnfrom(what);
        require(
            IERC20Minimal(baseToken).transfer(_account, what -
withdrawFee),
            "burnFrom: Token transfer failed"
        );
        require(
            IERC20Minimal(baseToken).transfer(to, withdrawFee),
            "burnFrom: Token transfer failed"
        );
```

[Surf protocol-05] Incorrect Parameter Usage

Severity Level	Info
Type	Business Security
Lines	PoolRouter.sol #L840-847
Description	In the PoolRouter contract, the `_lpUnpnl` function incorrectly passes parameters when using the `getUnrealizedPNL` function to calculate long and short profits. In the `getUnrealizedPNL` function, when the `isLong` parameter is true, it returns data for long positions, and when it's false, it returns data for short positions. However, in the `_lpUnpnl` function, the parameters were passed incorrectly, with the long parameter being passed as false and the short parameter being passed as true.

```
    function _lpUnpnl(
        address _pool,
        uint256 _price
    ) internal view returns (uint256 _unpnl, bool _isProfit) {
        (uint256 longAmount, bool longProfit) =
IPair(_pool).getUnrealizedPNL(
            _price,
            false
        );
        (uint256 shortAmount, bool shortProfit) =
IPair(_pool).getUnrealizedPNL(
            _price,
            true
        );
```

Recommendation	It is recommended to correct this parameter passing to the correct values.
Status	**Fixed.** The project team has corrected the flawed parameter logic.

```
    function _lpUnpnl(
        address _pool,
        uint256 _price
    ) internal view returns (uint256 _unpnl, bool _isProfit) {
        (uint256 longAmount, bool longProfit) =
IPoolUtil(config.poolUtil())
            .calcUnPNL(
                _price,
```

	```
	                IPair(_pool).longPrice(),
	                IPair(_pool).longAmount(),
	                true
	            );
	        (uint256 shortAmount, bool shortProfit) =
	IPoolUtil(config.poolUtil())
	            .calcUnPNL(
	                _price,
	                IPair(_pool).shortPrice(),
	                IPair(_pool).shortAmount(),
	                false
	            );
	```

[Surf protocol-06] Redundant Code

Severity Level	Info
Type	Coding Conventions
Lines	PoolRouter.sol #L608

Description In the PoolRouter contract, the _addLiquidity function authorizes the basetoken of the PoolRouter contract to the pool. However, based on the logic review, this authorization is redundant and not used subsequently.

```
    function _addLiquidity(uint _reqId, uint256 _price) private {
        LiqRequest storage req = requests[_reqId];
        IERC20(IPair(req.pool).baseToken()).approve(req.pool,
req.amount);
        (bool success, uint256 price, uint256 what) =
IPair(req.pool).mint(
            req.sender,
            req.amount,
            _price
        );
        require(success, "PoolRouter:mint pool fail");
        emit AddLiquidityEvent(req.sender, what, price, req.pool,
_reqId);
    }
```

Recommendation It is recommended to remove the corresponding redundant code.

Status **Fixed.** The project team has removed the corresponding redundant code.

```
    function _addLiquidity(uint _reqId, uint256 _price) private {
        LiqRequest storage req = requests[_reqId];
        (bool success, uint256 what) = IPair(req.pool).mint(
            req.sender,
            req.amount,
            _price
        );
        require(success, "PoolRouter: mint pool fail");
        emit AddLiquidityEvent(req.sender, what, _price, req.pool,
_reqId);
    }
```

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.

Impact / Likelihood	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.

3.1.4 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.

3.1.5 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	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		s.ContractRef.MsgSender Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

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 Solidity 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