Smart Contract
Security Audit Report

Prepared for Sigma Money

Prepared by Supremacy

May 17, 2025

Contents

1 Introduction

Given the opportunity to review the design document and related codebase of the Sigma Money, we outline in the report our systematic approach to evaluate potential security issues in the smart contract(s) implementation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Client

Sigma Money is bring binance launchpool yield to stablecoin hodlers via fx protocol.

Item	Description
Client	Sigma Money
Type	Smart Contract
Languages	Solidity
Platform	EVM-compatible

1.2 Audit Scope

In the following, we show the Git repository of reviewed file and the commit hash used in this security audit:

- Repository: https://github.com/SigmaMoney/contracts/tree/feat/sigma/contracts
- Commit Hash: 2ece4d41116e3dde886b19c9515fe9904080a697

Below are the files in scope for this security audit and their corresponding MD5 hashes.

Filename	MD5
`./base/price-oracle/PancakeV3SpotPriceReader.sol`	`34643fc743d1f78f1ba0b97f78dbe683`
`./core/PoolManager.sol`	`4891771302bc0e791744652c0f6b113e`
`./core/pool/AaveFundingPool.sol`	`53e230d23b740a790d10735ac968bd42`
`./price-oracle/SigmaClisBNBPriceOracle.sol`	`55ab59159b31254614d3558da88ef713`
`./rate-provider/ListaDAORateProvider.sol`	`8ba18a4fef3cf9543a071a70f6525536`
`./sigma/SigmaController.sol`	`9c54ba578a60979d1abd6c6136ebdf80`

1.3 Changelogs

Version	Date	Description
0.1	May 07, 2025	Initial Draft
1.0	May 17, 2025	Final Release

1.4 About Us

Supremacy is a leading blockchain security firm, composed of industry hackers and academic researchers, provide top-notch security solutions through our technology precipitation and innovative research.

We are reachable at X (https://x.com/SupremacyHQ), or Email (contact@supremacy.email).

1.5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- Likelihood represents the likelihood of a finding to be triggered or exploited in practice
- Impact specifies the technical and business-related consequences of a finding
- Severity is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Severity

Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

2 Findings

The table below summarizes the findings of the audit, including status and severity details.

ID	Severity	Description	Status
1	Critical	Potential theft of assets	Fixed
2	High	Potential denial of service due to lack of fees	Acknowledged
3	High	Potential denial of service due to logic issue	Fixed
4	Low	Lack of necessary assertion	Acknowledged
5	Low	Use `SafeERC20` library	Fixed
6	Low	Lack of address validation	Fixed
7	Low	Lack of `nonReentrant` modifier	Fixed
8	Low	Use `Ownable2Step` library	Fixed
9	Informational	Redundant Code Removal	Fixed
10	Informational	Immutable variables	Fixed
11	Informational	Follow the Check-Effects-Interactions Pattern	Fixed
12	Informational	Lack of ownership verification	Acknowledged
13	Informational	Lack of event record	Fixed
14	Informational	Lack of comment	Fixed

2.1 Critical

1. Potential theft of assets [Critical]

`Severity: Critical` `Likelihood: High` `Impact: High`

`Status: Fixed`

Description

The `deposit` function in the `SigmaController` contract enables users to deposit collateral (e.g., slisBNB) and mint debt assets (fxUSD) to facilitate the protocol's leverage and borrowing operations. However, a issue exists due to the lack of validation for fxUSD balance changes, allowing malicious actor to steal the entire fxUSD balance held by the contract. This flaw permits malicious actor to extract significantly more fxUSD than the legitimate newDebt amount, resulting in substantial financial losses.

```
41    function deposit(address _pool, uint256 amount, uint256 positionId, int256
    newColl, int256 newDebt) external {
42        require(amount >= 0, "Amount must be greater than or equal to 0");
43        require(amount == uint256(newColl), "New collateral must be equal to
    amount");
44        require(newDebt >= 0, "New debt must be greater than or equal to 0");
45
46        // transfer
47        IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
48        deposits[msg.sender] += amount;
49
50        // mint SigmaClisBNBSY
51        slisBNB.approve(address(sy), amount);
52        sy.deposit(address(this), address(slisBNB), amount, 0);
53
54        // deposit sy to fx pool
55        sy.approve(address(fxPoolManager), uint256(newColl));
56        uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl,
    newDebt);
57
58        // transfer fxUSD to the user
59        uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
60        IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
61        IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);
62
63        // transfer xBNB to the user
64        IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
65    }
```

SigmaController.sol

Recommendation

Consider always verifying balance changes before and after.

2.2 High

2. Potential denial of service due to lack of protocol fees [High]

`Severity: High` `Likelihood: High` `Impact: Medium`

`Status: Acknowledged`

Description

The `deposit` function in the `SigmaController` contract allows users to deposit collateral (e.g., slisBNB) and mint debt assets (fxUSD) to support the protocol's leverage and borrowing operations. However, a issue exists due to improper handling of protocol fees, which can lead to a denial of service. This flaw causes transactions to revert when users fail to provide sufficient slisBNB to cover both the collateral and protocol fees, preventing legitimate users from completing deposits and disrupting protocol functionality.

```
41    function deposit(address _pool, uint256 amount, uint256 positionId, int256
    newColl, int256 newDebt) external {
42        require(amount >= 0, "Amount must be greater than or equal to 0");
43        require(amount == uint256(newColl), "New collateral must be equal to
    amount");
44        require(newDebt >= 0, "New debt must be greater than or equal to 0");
45
46        // transfer
47        IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
48        deposits[msg.sender] += amount;
49
50        // mint SigmaClisBNBSY
51        slisBNB.approve(address(sy), amount);
52        sy.deposit(address(this), address(slisBNB), amount, 0);
53
54        // deposit sy to fx pool
55        sy.approve(address(fxPoolManager), uint256(newColl));
56        uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl,
    newDebt);
57
58        // transfer fxUSD to the user
59        uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
60        IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
61        IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);
62
63        // transfer xBNB to the user
64        IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
65    }
```

SigmaController.sol

In the following, we show the `operate` function, responsible for processing the deposit and managing protocol fees, requires the caller `SigmaController` to transfer both the `newColl` and `protocolFees`.

Its relevant logic is implemented as follows:

```
128    /*****************************
129     * Public Mutated Functions *
```

```
130     ****************************/
131
132   /// @inheritdoc IPoolManager
133   function operate(
134     address pool,
135     uint256 positionId,
136     int256 newColl,
137     int256 newDebt
138   ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns (uint256) {
139     address collateralToken = IPool(pool).collateralToken();
140     uint256 scalingFactor = _getTokenScalingFactor(collateralToken);
141
142     int256 newRawColl = newColl;
143     if (newRawColl != type(int256).min) {
144       newRawColl = _scaleUp(newRawColl, scalingFactor);
145     }
146
147     uint256 rawProtocolFees;
148     // the `newRawColl` is the result without `protocolFees`
149     (positionId, newRawColl, newDebt, rawProtocolFees) = IPool(pool).operate(
150       positionId,
151       newRawColl,
152       newDebt,
153       _msgSender()
154     );
155
156     newColl = _scaleDown(newRawColl, scalingFactor);
157     uint256 protocolFees = _scaleDown(rawProtocolFees, scalingFactor);
158     _changePoolDebts(pool, newDebt);
159     if (newRawColl > 0) {
160       _accumulatePoolOpenFee(pool, protocolFees);
161       _changePoolCollateral(pool, newColl, newRawColl);
162       IERC20(collateralToken).safeTransferFrom(_msgSender(), address(this), uint256(newColl) + protocolFees);
163     } else if (newRawColl < 0) {
164       _accumulatePoolCloseFee(pool, protocolFees);
165       _changePoolCollateral(pool, newColl - int256(protocolFees), newRawColl - int256(rawProtocolFees));
166       _transferOut(collateralToken, uint256(-newColl), _msgSender());
167     }
168
169     if (newDebt > 0) {
170       IFxUSDRegeneracy(fxUSD).mint(_msgSender(), uint256(newDebt));
171     } else if (newDebt < 0) {
172       IFxUSDRegeneracy(fxUSD).burn(_msgSender(), uint256(-newDebt));
173     }
174
175     emit Operate(pool, positionId, newColl, newDebt, protocolFees);
176
177     return positionId;
178   }
```

PoolManager.sol

Recommendation

Revise the code logic accordingly.

3. Potential denial of service due to logic issue [High]

`Severity: High` `Likelihood: High` `Impact: Medium`

`Status: Fixed`

Description

The `deposit` function in the `SigmaController` contract enables users to deposit collateral (e.g., slisBNB) and mint debt assets (fxUSD) to support the protocol's leverage and borrowing operations. A critical design flaw exists in the handling of NFT position transfers, specifically the `IERC721(_pool).transferFrom(address(this), msg.sender, positionId)` call, which reverts during the second deposit invocation for an existing position. This issue prevents users from updating or managing existing positions, leading to a denial of service condition that disrupts protocol functionality.

```
41    function deposit(address _pool, uint256 amount, uint256 positionId, int256
    newColl, int256 newDebt) external {
42        require(amount >= 0, "Amount must be greater than or equal to 0");
43        require(amount == uint256(newColl), "New collateral must be equal to
    amount");
44        require(newDebt >= 0, "New debt must be greater than or equal to 0");
45
46        // transfer
47        IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
48        deposits[msg.sender] += amount;
49
50        // mint SigmaClisBNBSY
51        slisBNB.approve(address(sy), amount);
52        sy.deposit(address(this), address(slisBNB), amount, 0);
53
54        // deposit sy to fx pool
55        sy.approve(address(fxPoolManager), uint256(newColl));
56        uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl,
    newDebt);
57
58        // transfer fxUSD to the user
59        uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
60        IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
61        IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);
62
63        // transfer xBNB to the user
64        IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
65    }
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

2.3 Low

4. Lack of necessary assertion [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Acknowledged`

Description

In the `SigmaController::deposit()` function, it enables users to deposit collateral (e.g., `slisBNB`) and mint debt assets (`fxUSD`) to support the protocol's leverage and borrowing operations. Due to the absence of validation to ensure that the `fxUSD` amount transferred to the user (`fxUSDOut`) equals the minted debt amount (`newDebt`). The function transfers the entire `fxUSD` balance of the contract without checking if it matches `newDebt`, which could lead to unintended transfers of excess `fxUSD` if the contract holds additional `fxUSD` from other operations. This could result in minor financial discrepancies or user confusion, though the impact is limited under normal protocol conditions.

```
41    function deposit(address _pool, uint256 amount, uint256 positionId, int256
    newColl, int256 newDebt) external {
42        require(amount >= 0, "Amount must be greater than or equal to 0");
43        require(amount == uint256(newColl), "New collateral must be equal to
    amount");
44        require(newDebt >= 0, "New debt must be greater than or equal to 0");
45
46        // transfer
47        IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
48        deposits[msg.sender] += amount;
49
50        // mint SigmaClisBNBSY
51        slisBNB.approve(address(sy), amount);
52        sy.deposit(address(this), address(slisBNB), amount, 0);
53
54        // deposit sy to fx pool
55        sy.approve(address(fxPoolManager), uint256(newColl));
56        uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl,
    newDebt);
57
58        // transfer fxUSD to the user
59        uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
60        IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
61        IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);
62
63        // transfer xBNB to the user
64        IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
65    }
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

5. Use `SafeERC20` library [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Fixed`

Description

To ensure robust and secure interaction with ERC20 tokens, it is recommended to use OpenZeppelin's `SafeERC20` library. The library provides wrapper functions that contain additional checks and protections to ensure that token transfers and other operations are performed safely and correctly.

Recommendation

It is recommended to use the `SafeERC20` method `transfer` and `transferFrom`, instead of unsafe `transfer` and `transferFrom`.

6. Lack of address validation [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Fixed`

Description

In the `SigmaController` contract, several functions lack zero-address validation.

```
41    function deposit(address _pool, uint256 amount, uint256 positionId, int256
      newColl, int256 newDebt) external {
42        require(amount >= 0, "Amount must be greater than or equal to 0");
43        require(amount == uint256(newColl), "New collateral must be equal to
      amount");
44        require(newDebt >= 0, "New debt must be greater than or equal to 0");
45
46        // transfer
47        IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
48        deposits[msg.sender] += amount;
49
50        // mint SigmaClisBNBSY
51        slisBNB.approve(address(sy), amount);
52        sy.deposit(address(this), address(slisBNB), amount, 0);
53
54        // deposit sy to fx pool
55        sy.approve(address(fxPoolManager), uint256(newColl));
56        uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl,
      newDebt);
57
58        // transfer fxUSD to the user
59        uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
60        IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
61        IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);
62
63        // transfer xBNB to the user
64        IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
65    }
66
```

```
67    function redeem(address _pool, uint256 amount, uint256 positionId, int256
      newColl, int256 newDebt) external {
68        // check if the user has enough deposits
69        require(deposits[msg.sender] >= amount, "Insufficient deposits");
70        require(amount >= 0, "Amount must be greater than or equal to 0");
71        require(amount == uint256(-newColl), "New collateral absolute value must be
      equal to amount");
72        require(newDebt <= 0, "New debt must be less than or equal to 0");
73
74        if (newDebt < 0) {
75          // transfer fxUSD to this contract
76          IERC20(IPool(_pool).fxUSD()).transferFrom(msg.sender, address(this),
      uint256(-newDebt));
77          IERC20(IPool(_pool).fxUSD()).approve(address(fxPoolManager), uint256(-
      newDebt));
78        }
79
80        // transfer xBNB to this contract
81        IERC721(_pool).transferFrom(msg.sender, address(this), positionId);
82
83        // withdraw sy from fx pool
84        uint256 syBalance = sy.balanceOf(address(this));
85        fxPoolManager.operate(_pool, positionId, newColl, newDebt);
86        uint256 syDelta = sy.balanceOf(address(this)) - syBalance;
87
88        // transfer fxUSD to the user
89        IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
90
91        if (syDelta > 0) {
92          // burn sy, redeem slisBNB
93          sy.redeem(address(this), syDelta, address(slisBNB), 0, false);
94
95          // transfer slisBNB to user and update deposits
96          slisBNB.transfer(msg.sender, syDelta);
97        }
98
99        deposits[msg.sender] -= amount;
100     }
```

SigmaController.sol

Recommendation

Consider adding zero address validation

7. Lack of `nonReentrant` modifier [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Fixed`

Description

In the `SigmaController` contract, several functions involves an external call. There are no obvious issues in the current implementation, but to increase security and prevent unexpected behavior, it is a good idea to include the `nonReentrant` modifier. This is a

safer approach and prevents potential issues in the event of future updates or unexpected situations.

Recommendation

Add a `nonReentrant` type modifier to the functions to further improve security.

8. Use `Ownable2Step` library [Low]

`Severity: Low` `Likelihood: Low` `Impact: Low`

`Status: Fixed`

Description

A single-step ownership transfer means that if an incorrect address is passed when transferring ownership or administrative privileges, it means that the role will be lost forever. The ownership model for this protocol is implemented in `Ownable.sol`, which implements single-step transfers. This can cause problems for all methods marked as `onlyOwner` throughout the protocol, some of which are core functionality of the protocol.

Recommendation

A two-step ownership transfer model is recommended, where the ownership transfer is `pending` and the new owner should assert its new rights, otherwise the old owner remains in control of the contract. Consider using OpenZeppelin's `Ownable2Step` contract.

2.4 Informational

9. Redundant Code Removal [Informational]

`Status: Fixed`

Description

In the `SigmaController::deposit` function, it approve the SigmaController contract itself (address(this)) to manage fxUSDOut amount of fxUSD tokens. However, the subsequent transfer operation, `IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut)`, does not require any approve. Thus, this approve call is redundant.

```
60      IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

10. Immutable variables [Informational]

`Status: Fixed`

Description

Variables such as `slisBNB`, `sy`, `slisBNBProvider`, `fxPoolManager` and `listaLpDelegateTo` are defined only in the `constructor()`. Therefore, it can be `immutable`, since `immutable` values are cheaper to read.

Recommendation

Consider making variables immutable.

11. Follow the Check-Effects-Interactions Pattern [Informational]

`Status: Fixed`

Description

In the `deposit` and `redeem` functions of the `SigmaController` contract, the control flow does not follow the check-effects-interact pattern.

Recommendation

Revise the code logic accordingly.

12. Lack of ownership verification [Informational]

`Status: Acknowledged`

Description

Lack of ownership verification of `positionId` in the `deposit` and `redeem` functions of `SigmaController` contract.

Recommendation

Revise the code logic accordingly.

13. Lack of event record [Informational]

`Status: Fixed`

Description

In the `SigmaController` contract, the `deposit()` and `redeem()` functions are missing event records. However, events are important because off-chain monitoring tools rely on them to index important state changes to the smart contract(s).

Recommendation

Always ensure that all functions that trigger state changes have event logging capabilities.

14. Lack of comment [Informational]

`Status: Fixed`

Description

Throughout the codebase there are numerous functions missing or lacking documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally,

comments improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Recommendation

Consider thoroughly documenting all functions (and their parameters) that are part of the smart contracts' public interfaces. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing comments, consider following the Ethereum Natural Specification Format (NatSpec).

3 Disclaimer

This security audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. This security audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues, also cannot make guarantees about any additional code added to the assessed project after the audit version. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contract(s). Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.