SUPREMACY

Smart Contract
Security Audit Report

Prepared for Sigma Money

Prepared by Supremacy

July 01, 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:

Version	Repository	Commit Hash
1	contracts	`2ece4d41116e3dde886b19c9515fe9904080a697`
2	contracts	`8c0753a4c076183494daa0f89483be795133f450`
3	contracts	`a7047fba8f7c79f5b9dc8a83fc97c09da11a1bc4`

1.3 Changelogs

Version	Date	Description
0.1	May 07, 2025	Initial Draft
1.0	May 17, 2025	Final Release
1.1	June 17, 2025	Post-Final Release #1
1.2	July 01, 2025	Post-Final Release #2

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.

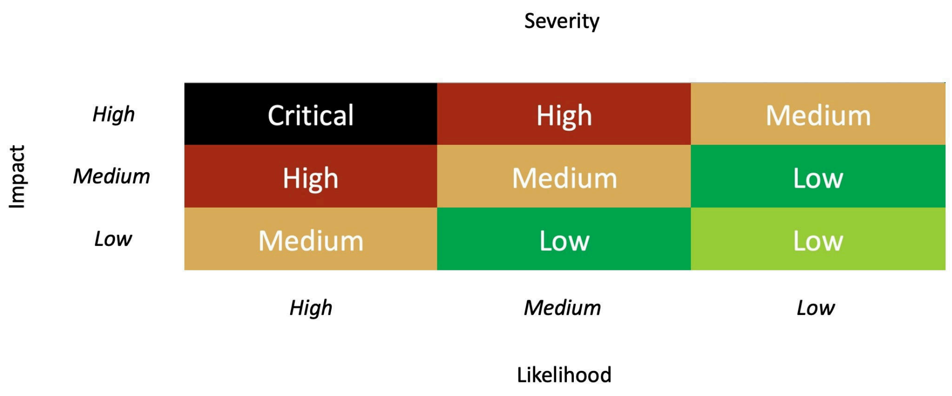
We are reachable at X (https://x.com/SupremacyHQ), or Email (contact@supremacy.email).

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

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	High	Potential denial of service due to fees hardcoded	Fixed
5	Medium	Lack of slippage check	Fixed
6	Low	Lack of necessary assertion	Acknowledged
7	Low	Use `SafeERC20` library	Fixed
8	Low	Lack of address validation	Fixed
9	Low	Lack of `nonReentrant` modifier	Fixed
10	Low	Use `Ownable2Step` library	Fixed
11	Informational	Redundant code removal for self approval	Fixed
12	Informational	Immutable variables	Fixed
13	Informational	Follow the Check-Effects-Interactions Pattern	Fixed
14	Informational	Lack of ownership verification	Acknowledged
15	Informational	Lack of event record	Fixed
16	Informational	Lack of comment	Fixed
17	Informational	Lack of pause check for `listaStakeManager`	Fixed
18	Informational	Potential arbitrary external call	Acknowledged
19	Informational	Redundant state variable removal	Fixed
20	Informational	Merging redundant functions	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.

```
  function deposit(address _pool, uint256 amount, uint256 positionId, int256 newColl, int256 newDebt) external {
    require(amount >= 0, "Amount must be greater than or equal to 0");
    require(amount == uint256(newColl), "New collateral must be equal to amount");
    require(newDebt >= 0, "New debt must be greater than or equal to 0");

    // transfer
    IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
    deposits[msg.sender] += amount;

    // mint SigmaClisBNBSY
    slisBNB.approve(address(sy), amount);
    sy.deposit(address(this), address(slisBNB), amount, 0);

    // deposit sy to fx pool
    sy.approve(address(fxPoolManager), uint256(newColl));
    uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl, newDebt);

    // transfer fxUSD to the user
    uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
    IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
    IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);

    // transfer xBNB to the user
    IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
  }
```

SigmaController.sol

Recommendation

Consider always verifying balance changes before and after.

2.2 High

2. Potential denial of service due to lack of 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.

```
  function deposit(address _pool, uint256 amount, uint256 positionId, int256 newColl, int256 newDebt) external {
    require(amount >= 0, "Amount must be greater than or equal to 0");
    require(amount == uint256(newColl), "New collateral must be equal to amount");
    require(newDebt >= 0, "New debt must be greater than or equal to 0");

    // transfer
    IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
    deposits[msg.sender] += amount;

    // mint SigmaClisBNBSY
    slisBNB.approve(address(sy), amount);
    sy.deposit(address(this), address(slisBNB), amount, 0);

    // deposit sy to fx pool
    sy.approve(address(fxPoolManager), uint256(newColl));
    uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl, newDebt);

    // transfer fxUSD to the user
    uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
    IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
    IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);

    // transfer xBNB to the user
    IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
  }
```

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:

```
  /*****************************
   * Public Mutated Functions *
   *****************************/

  /// @inheritdoc IPoolManager
  function operate(
    address pool,
    uint256 positionId,
    int256 newColl,
    int256 newDebt
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256) {
    address collateralToken = IPool(pool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    int256 newRawColl = newColl;
    if (newRawColl != type(int256).min) {
      newRawColl = _scaleUp(newRawColl, scalingFactor);
    }

    uint256 rawProtocolFees;
    // the `newRawColl` is the result without `protocolFees`
    (positionId, newRawColl, newDebt, rawProtocolFees) = IPool(pool).operate(
      positionId,
      newRawColl,
      newDebt,
      _msgSender()
    );

    newColl = _scaleDown(newRawColl, scalingFactor);
    uint256 protocolFees = _scaleDown(rawProtocolFees, scalingFactor);
    _changePoolDebts(pool, newDebt);
    if (newRawColl > 0) {
      _accumulatePoolOpenFee(pool, protocolFees);
      _changePoolCollateral(pool, newColl, newRawColl);
      IERC20(collateralToken).safeTransferFrom(_msgSender(), address(this),
uint256(newColl) + protocolFees);
    } else if (newRawColl < 0) {
      _accumulatePoolCloseFee(pool, protocolFees);
      _changePoolCollateral(pool, newColl - int256(protocolFees), newRawColl -
int256(rawProtocolFees));
      _transferOut(collateralToken, uint256(-newColl), _msgSender());
    }

    if (newDebt > 0) {
      IFxUSDRegeneracy(fxUSD).mint(_msgSender(), uint256(newDebt));
    } else if (newDebt < 0) {
      IFxUSDRegeneracy(fxUSD).burn(_msgSender(), uint256(-newDebt));
    }

    emit Operate(pool, positionId, newColl, newDebt, protocolFees);

    return positionId;
  }
```

PoolManager.sol

Recommendation

Revise the code logic accordingly.

Feedback

When the user's balance is insufficient to deduct the processing fee, the transaction will be revert.

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.

```
  function deposit(address _pool, uint256 amount, uint256 positionId, int256 newColl, int256 newDebt) external {
    require(amount >= 0, "Amount must be greater than or equal to 0");
    require(amount == uint256(newColl), "New collateral must be equal to amount");
    require(newDebt >= 0, "New debt must be greater than or equal to 0");

    // transfer
    IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
    deposits[msg.sender] += amount;

    // mint SigmaClisBNBSY
    slisBNB.approve(address(sy), amount);
    sy.deposit(address(this), address(slisBNB), amount, 0);

    // deposit sy to fx pool
    sy.approve(address(fxPoolManager), uint256(newColl));
    uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl, newDebt);

    // transfer fxUSD to the user
    uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
    IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
    IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);

    // transfer xBNB to the user
    IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
  }
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

4. Potential denial of service due to fees hardcoded [High]

```
Severity: High                    Likelihood: High                    Impact: Medium
Status: Fixed
```

Description

In the `SigmaController` contract, the `_convertSlisBNBToWBNB()` function facilitates token swaps from `slisBNB` to `wBNB` via `pancakeSwapRouter::exactInputSingle()`. However, the pool fee is hardcoded to `100` (0.01%), without verifying whether the `slisBNB/wBNB` pool uses this fee tier. PancakeSwap V3 supports multiple fee tiers (e.g., `100`, `500`, `3000`, `10000`, corresponding to `0.01%`, `0.05%`, `0.3%`, `1%`). If the pool's actual fee tier differs from the hardcoded value, the swap transaction will fail, causing a revert.

```
    function _convertSlisBNBToWBNB(uint256 slisBNBAmount) internal returns
(uint256 wBNBAmount) {
        slisBNB.forceApprove(address(pancakeSwapRouter), slisBNBAmount);
        IV3SwapRouter.ExactInputSingleParams memory params =
IV3SwapRouter.ExactInputSingleParams(
            address(slisBNB),
            address(wBNB),
            100,
            address(this),
            slisBNBAmount,
            slisBNBAmount, // amountOutMinimum
            0
        );
        wBNBAmount = pancakeSwapRouter.exactInputSingle(params);
    }
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

Feedback

This function has been removed.

2.3 Medium

5. Lack of slippage check [Medium]

```
Severity: Medium                  Likelihood: Medium                  Impact: Medium
Status: Fixed
```

Description

The `SigmaController` contract facilitates token swaps through the `_swap()` function, which interacts with external swap targets (e.g., decentralized exchanges) to convert input tokens to output tokens, such as during collateral conversion in the `_transferInCollAndConvert()` function. However, the `_swap()` function does not implement slippage protection, which is a critical safeguard in decentralized finance (DeFi) protocols. Slippage protection ensures that the amount of output tokens received from a swap meets a minimum threshold, preventing users from receiving significantly fewer tokens than expected due to price volatility or front-running attacks like Miner Extractable Value (MEV). The contract defines constants `MAX_SLIPPAGE` (10%) and `MIN_SLIPPAGE` (0.01%), indicating an intention to handle slippage tolerance, but these are not utilized in the `_swap()` function. Without enforcing a minimum output amount, users are exposed to potential losses if the market price moves unfavorably during the transaction, especially in volatile markets or low-liquidity pools. This issue affects the `deposit()` function, where collateral tokens are swapped to slisBNB, and potentially other operations relying on `_swap()`. The lack of slippage checks could lead to financial losses for users and undermine trust in the protocol.

```
  /// @dev Internal function to do swap.
  /// @param tokenIn The address of input token.
  /// @param tokenOut The address of output token.
  /// @param amountIn The amount of input token.
  /// @param swapTarget The address of target contract used for swap.
  /// @param swapData The calldata passed to target contract.
  /// @return amountOut The amount of output tokens received.
  function _swap(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    address swapTarget,
    bytes memory swapData
  ) internal returns (uint256 amountOut) {
    _onlySupportedSwapTarget(swapTarget);

    if (amountIn == 0) return 0;

    amountOut = _balanceOf(tokenOut, address(this));
    if (tokenIn != address(0)) {
      IERC20(tokenIn).forceApprove(swapTarget, amountIn);
      (bool success, ) = swapTarget.call(swapData);
      // below lines will propagate inner error up
      if (!success) {
        // solhint-disable-next-line no-inline-assembly
        assembly {
          let ptr := mload(0x40)
          let size := returndatasize()
          returndatacopy(ptr, 0, size)
          revert(ptr, size)
        }
      }
    } else {
      (bool success, ) = swapTarget.call{ value: amountIn }(swapData);
      if (!success) {
        // solhint-disable-next-line no-inline-assembly
        assembly {
          let ptr := mload(0x40)
          let size := returndatasize()
          returndatacopy(ptr, 0, size)
          revert(ptr, size)
        }
      }
    }
    amountOut = _balanceOf(tokenOut, address(this)) - amountOut;
  }
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

2.4 Low

6. 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.

```
  function deposit(address _pool, uint256 amount, uint256 positionId, int256 newColl, int256 newDebt) external {
    require(amount >= 0, "Amount must be greater than or equal to 0");
    require(amount == uint256(newColl), "New collateral must be equal to amount");
    require(newDebt >= 0, "New debt must be greater than or equal to 0");

    // transfer
    IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
    deposits[msg.sender] += amount;

    // mint SigmaClisBNBSY
    slisBNB.approve(address(sy), amount);
    sy.deposit(address(this), address(slisBNB), amount, 0);

    // deposit sy to fx pool
    sy.approve(address(fxPoolManager), uint256(newColl));
    uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl, newDebt);

    // transfer fxUSD to the user
    uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
    IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
    IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);

    // transfer xBNB to the user
    IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
  }
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

7. 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`.

8. Lack of address validation [Low]

```
Severity: Low                    Likelihood: Low                    Impact: Low
Status: Fixed
```

Description

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

```
  function deposit(address _pool, uint256 amount, uint256 positionId, int256 newColl, int256 newDebt) external {
    require(amount >= 0, "Amount must be greater than or equal to 0");
    require(amount == uint256(newColl), "New collateral must be equal to amount");
    require(newDebt >= 0, "New debt must be greater than or equal to 0");

    // transfer
    IERC20(slisBNB).transferFrom(msg.sender, address(this), amount);
    deposits[msg.sender] += amount;

    // mint SigmaClisBNBSY
    slisBNB.approve(address(sy), amount);
    sy.deposit(address(this), address(slisBNB), amount, 0);

    // deposit sy to fx pool
    sy.approve(address(fxPoolManager), uint256(newColl));
    uint256 positionId = fxPoolManager.operate(_pool, positionId, newColl, newDebt);

    // transfer fxUSD to the user
    uint256 fxUSDOut = IERC20(IPool(_pool).fxUSD()).balanceOf(address(this));
    IERC20(IPool(_pool).fxUSD()).approve(address(this), fxUSDOut);
    IERC20(IPool(_pool).fxUSD()).transfer(msg.sender, fxUSDOut);

    // transfer xBNB to the user
    IERC721(_pool).transferFrom(address(this), msg.sender, positionId);
  }

  function redeem(address _pool, uint256 amount, uint256 positionId, int256
newColl, int256 newDebt) external {
    // check if the user has enough deposits
    require(deposits[msg.sender] >= amount, "Insufficient deposits");
    require(amount >= 0, "Amount must be greater than or equal to 0");
    require(amount == uint256(-newColl), "New collateral absolute value must be
equal to amount");
    require(newDebt <= 0, "New debt must be less than or equal to 0");

    if (newDebt < 0) {
      // transfer fxUSD to this contract
      IERC20(IPool(_pool).fxUSD()).transferFrom(msg.sender, address(this),
uint256(-newDebt));
      IERC20(IPool(_pool).fxUSD()).approve(address(fxPoolManager), uint256(-
newDebt));
    }

    // transfer xBNB to this contract
    IERC721(_pool).transferFrom(msg.sender, address(this), positionId);

    // withdraw sy from fx pool
    uint256 syBalance = sy.balanceOf(address(this));
    fxPoolManager.operate(_pool, positionId, newColl, newDebt);
    uint256 syDelta = sy.balanceOf(address(this)) - syBalance;

    // transfer fxUSD to the user
    IERC721(_pool).transferFrom(address(this), msg.sender, positionId);

    if (syDelta > 0) {
      // burn sy, redeem slisBNB
      sy.redeem(address(this), syDelta, address(slisBNB), 0, false);

      // transfer slisBNB to user and update deposits
      slisBNB.transfer(msg.sender, syDelta);
    }

    deposits[msg.sender] -= amount;
  }
```

SigmaController.sol

Recommendation

Consider adding zero address validation

9. 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.

10. 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.

2.5 Informational

11. Redundant code removal for self approval [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.

12. 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.

13. 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.

14. 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.

15. 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.

16. 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).

17. Lack of pause check for `listaStakeManager` [Informational]

`Status: Fixed`

Description

In the `SigmaController` contract, there is a lack of pause checks for the `requestWithdraw()` and `claimWithdraw()` functions of `listaStakeManager`. If the contract is in a paused state, this could lead to unnecessary gas consumption.

Recommendation

Revise the code logic accordingly.

18. Potential arbitrary external call [Informational]

`Status: Acknowledged`

Description

The `_swap` function in the `SigmaController` contract allows the execution of arbitrary external calls to a `swapTarget` address via low-level call (i.e., `swapTarget.call(swapData)`). While the function restricts `swapTarget` to addresses listed in the `supportedSwapTargets` whitelist, this mechanism is insufficient to mitigate the risks associated with arbitrary external calls. The following issues could lead to arbitrary theft of unlimited approval user assets.

Recommendation

Strictly scrutinize the whitelists.

19. Redundant state variable removal [Informational]

`Status: Fixed`

Description

The constants `MAX_SLIPPAGE` and `MIN_SLIPPAGE` defined by the `SigmaController` contract are not used efficiently.

Recommendation

Consider removing redundant code.

20. Merging redundant functions [Informational]

`Status: Fixed`

Description

The `registerSwapTarget()` and `unregisterSwapTarget()` functions can be combined into a single function.

```
function registerSwapTarget(address swapTarget) external onlyOwner {
  supportedSwapTargets[swapTarget] = true;
}

function unregisterSwapTarget(address swapTarget) external onlyOwner {
  require(supportedSwapTargets[swapTarget], "Swap target not registered");
  delete supportedSwapTargets[swapTarget];
}
```

SigmaController.sol

Recommendation

Revise the code logic accordingly.

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.