FAIRYPROOF

Singularry Wallet

AUDIT REPORT

Version 1.0.0

Serial No. 2025122900012026

Presented by Fairyproof

December 29, 2025

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the Singularry Wallet project.

Audit Start Time:

September 19, 2025

Audit End Time:

December 29, 2025

Audited Code's Github Repository:

https://github.com/singularry/slywallet-contracts

Audited Code's Github Commit Number When Audit Started:

9f47f33c403201e0a111863deef4247e9005cff0

Audited Code's Github Commit Number When Audit Ended:

902a899521a6ea85a8aa9387a7fa8a3106b8117e

Audited Source Files:

The source files audited include all the files as follows:

```
./contracts/slywallet/
├── facets
│   ├── base
│   │   ├── ISLYWalletBase.sol
│   │   └── SLYWalletBaseFacet.sol
│   ├── dca
│   │   ├── ISLYWalletDCAStrategyFacet.sol
│   │   ├── SLYWalletDCAStorage.sol
│   │   └── SLYWalletDCAStrategyFacet.sol
│   ├── sdr
│   │   ├── ISLYWalletDRIPStrategyFacet.sol
│   │   ├── SLYWalletDRIPStorage.sol
│   │   └── SLYWalletDRIPStrategyFacet.sol
│   ├── SLYDiamondCutFacet.sol
│   ├── SLYDiamondLoupeFacet.sol
├── libraries
│   ├── LibPermissions.sol
│   └── LibSLYDiamond.sol
├── SLYWalletDiamond.sol
├── SLYWalletDiamondInit.sol
├── SLYWalletFactory.sol
├── SLYWalletReentrancyGuard.sol
```

```
└── SLYWalletStorage.sol
```

The goal of this audit is to review Singularry's solidity implementation for its Wallet function, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

This audit only applies to the specified code, software or any materials supplied by the Singularry team for specified versions. Whenever the code, software, materials, settings, environment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code Review, Including:

- Project Diagnosis

Understanding the size, scope and functionality of your project's source code based on the specifications, sources, and instructions provided to Fairyproof.

- Manual Code Review

Reading your source code line-by-line to identify potential vulnerabilities.

- Specification Comparison

Determining whether your project's code successfully and efficiently accomplishes or executes its functions according to the specifications, sources, and instructions provided to Fairyproof.

2. Testing and Automated Analysis, Including:

- Test Coverage Analysis

Determining whether the test cases cover your code and how much of your code is exercised or executed when test cases are run.

- Symbolic Execution

Analyzing a program to determine the specific input that causes different parts of a program to execute its functions.

3. Best Practices Review

Reviewing the source code to improve maintainability, security, and control based on the latest established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following source(s) of truth about how the token issuance function should work:

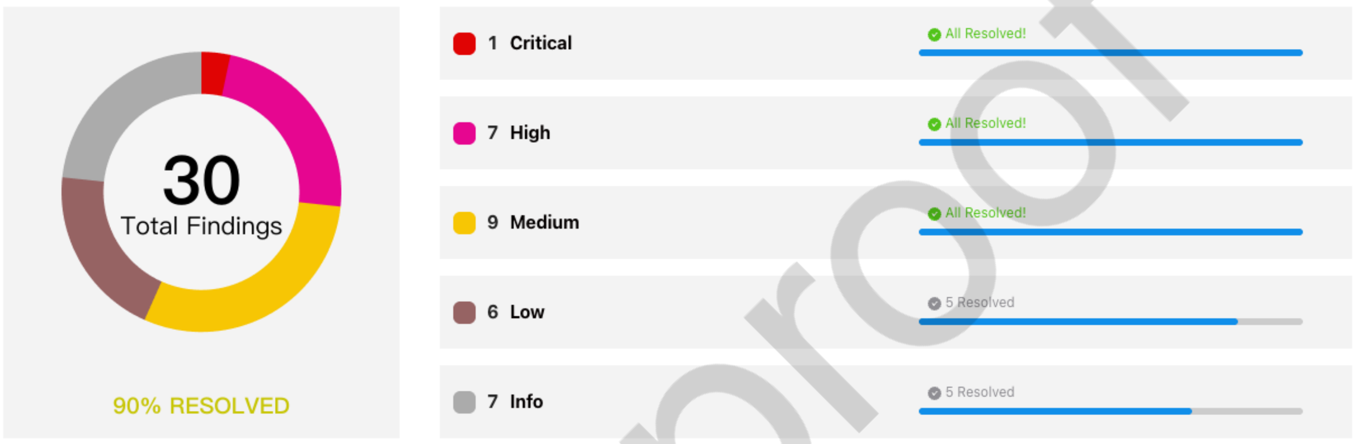
Website:https://singularry.org/

Source Code: https://github.com/singularry/slywallet-contracts

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the Singularry team or reported an issue.

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2025122900012026	Fairyproof Security Team	Sep 19, 2025 - Dec 29, 2025	Low Risk



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, one issue of critical-severity, seven issues of high-severity, nine issues of medium-severity, fix issues of low-severity and seven issues of info-severity were uncovered. The Singularry team fixed one issue of critical, seven issues of high, nine issues of medium, five issues of low and five issues of info, and acknowledged the remaining issues.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to Singularry

Singularry is a platform of creating next-generation financial systems with intelligent, autonomous protocols and seamless cross-chain operations.

The above description is quoted from relevant documents of Singularry.

04. Major functions of audited code

The main subject of this audit is a smart wallet that implements the Diamond protocol. The wallet includes built-in operational features and supports investment strategies involving the Venus and Thena protocols.

05. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Access Control
- Admin Rights
- Arithmetic Precision
- Code Improvement
- Contract Upgrade/Migration
- Delete Trap
- Design Vulnerability
- DoS Attack
- EOA Call Trap
- Fake Deposit
- Function Visibility
- Gas Consumption
- Implementation Vulnerability
- Inappropriate Callback Function
- Injection Attack
- Integer Overflow/Underflow
- IsContract Trap
- Miner's Advantage

- Misc
- Price Manipulation
- Proxy selector clashing
- Pseudo Random Number
- Re-entrancy Attack
- Replay Attack
- Rollback Attack
- Shadow Variable
- Slot Conflict
- Token Issuance
- Tx.origin Authentication
- Uninitialized Storage Pointer

06. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Informational is not an issue or risk but a suggestion for code improvement.

07. Major areas that need attention

Based on the provided source code the Fairyproof team focused on the possible issues and risks related to the following functions or areas.

- Function Implementation

We checked whether or not the functions were correctly implemented.
We found some issues, for more details please refer to [FP-1,FP-5,FP-7,FP-8,FP-9,FP-10,FP-11,FP-12,FP-13,FP-14,FP-16,FP-18,FP-19,FP-21,FP-22,FP-23,FP-24,FP-25,FP-27] in "09. Issue description".

- Access Control

We checked each of the functions that could modify a state, especially those functions that could only be accessed by owner or administrator
We found some issues, for more details please refer to [FP-4,FP-20,FP-26,FP-28] in "09. Issue description".

- Token Issuance & Transfer

We examined token issuance and transfers for situations that could harm the interests of holders.
We didn't find issues or risks in these functions or areas at the time of writing.

- State Update

We checked some key state variables which should only be set at initialization.
We didn't find issues or risks in these functions or areas at the time of writing.

- Asset Security

We checked whether or not all the functions that transfer assets were safely handled.
We found one issue, for more details please refer to [FP-6] in "09. Issue description".

- Miscellaneous

We checked the code for optimization and robustness.
We found some issues, for more details please refer to [FP-2,FP-3,FP-15,FP-17,FP-29,FP-30] in "09. Issue description".

08. List of issues by severity

Index	Title	Issue/Risk	Severity	Status
FP-1	Price limit logic is fundamentally flawed for non-stablecoin funding tokens	Design Vulnerability	Low	Acknowledged
FP-2	Gas optimization for double iteration patterns	Code Improvement	Info	Acknowledged
FP-3	DCAPosition lacks tracking of total tokens received	Code Improvement	Info	Acknowledged
FP-4	Authenticator role has excessive permissions allowing complete asset control bypass	Access Control	Critical	✓ Fixed
FP-5	Reentrancy vulnerability allows single DCA position to be executed multiple times	Code Improvement	High	✓ Fixed
FP-6	Missing slippage protection in DCA strategy exposes users to MEV attacks and value loss	Design Vulnerability	High	✓ Fixed
FP-7	Reentrancy vulnerability allows single DCA position to be executed multiple times	Access Control	High	✓ Fixed
FP-8	Missing slippage protection in DRIP strategy exposes users to MEV attacks and value loss	Code Improvement	High	✓ Fixed
FP-9	DRIP strategy receives wnative token instead of native token without auto-conversion, causing gradual native token depletion	Code Improvement	High	✓ Fixed
FP-10	Failed swaps leave irreversible side effects from token preparation without cleanup mechanism	Code Improvement	High	✓ Fixed
FP-11	Critical facets can be arbitrarily removed, bricking the wallet	Code Improvement	High	✓ Fixed
FP-12	`withdrawERC20` incompatible with non-standard ERC20 tokens like USDT	Code Improvement	Medium	✓ Fixed
FP-13	Zero address facets create irreversible function selector occupation	Code Improvement	Medium	✓ Fixed
FP-14	`removeFunctions` lacks facet address validation allowing unauthorized function removal	Code Improvement	Medium	✓ Fixed
FP-15	Insufficient precision in block time calculation for modern high-speed blockchains	Design Vulnerability	Medium	✓ Fixed
FP-16	Global `thenaDeployer` configuration limits pool type flexibility in DCA/DRIP positions	Design Vulnerability	Medium	✓ Fixed
FP-17	Confusing logic and naming in `_hasAllocation` function requires clarification	Code Improvement	Medium	✓ Fixed
FP-18	DRIP positions with percentage-based trading cannot auto-complete due to geometric convergence leaving dust amounts	Code Improvement	Medium	✓ Fixed
FP-19	`totalFeesCollected` field aggregates fees from different tokens making the value meaningless	Code Improvement	Medium	✓ Fixed
FP-20	`withdrawVToken` can be replaced by `withdrawERC20`	Admin Rights	Medium	✓ Fixed
FP-21	`addKey` function lacks validation for `Role.None`, potentially allowing invalid key additions	Code Improvement	Low	✓ Fixed
FP-22	No pagination mechanism implemented	Design Vulnerability	Low	✓ Fixed
FP-23	No deduplication of owners during wallet creation	Code Improvement	Low	✓ Fixed
FP-24	`replaceFunctions` lacks old facet cleanup logic causing storage bloat	Code Improvement	Low	✓ Fixed
FP-25	Incorrect length validation in `batchDiamondCutWithMultipleInits` causing transaction failures	Code Improvement	Low	✓ Fixed

FP-26	Unnecessary complexity in permission system - redundant `permissions` mapping	Code Improvement	Info	✓ Fixed
FP-27	Unnecessary public visibility of `hasPermission` function in `LibPermissions`	Code Improvement	Info	✓ Fixed
FP-28	Overly complex permission check	Access Control	Info	✓ Fixed
FP-29	`_checkMarketAlignment` function has poor readability due to nested try-catch blocks	Code Improvement	Info	✓ Fixed
FP-30	`sdrGetReadyPositions` function can be optimized to reduce gas consumption by avoiding double iteration	Code Improvement	Info	✓ Fixed

09. Issue descriptions

[FP-1] Price limit logic is fundamentally flawed for non-stablecoin funding tokens

Design Vulnerability Low Acknowledged

Issue/Risk: Design Vulnerability

Description:

The current price checking mechanism always uses `tokenOut` price from the oracle regardless of the `fundingToken` type. This creates meaningless and potentially misleading price limits when the funding token is not a stablecoin, causing DCA strategies to execute at unintended times and undermining the core value proposition of price-controlled DCA.

Problematic Code:

```
// In dcaExecuteTrades function - lines ~275-285
if (position.maxPrice > 0 || position.minPrice > 0) {
    uint256 currentPrice = IPriceOracle(ds.priceOracle).getPrice(position.tokenOut);

    if ((position.maxPrice > 0 && currentPrice > position.maxPrice) ||
        (position.minPrice > 0 && currentPrice < position.minPrice)) {
        // Skip this trade but update next trigger
        position.nextTriggerBlock = block.number + position.triggerInterval;

        emit DCAPriceOutOfRange(...);
        continue;
    }
}
```

Recommendation:

Option 1: Exchange Rate Based Pricing (Most logical)

Update/Status:

The team has skipped the issue.

[FP-2] Gas optimization for double iteration patterns

Code Improvement Info Acknowledged

Issue/Risk: Code Improvement

Description:

Multiple functions in `SLYDiamondLoupeFacet` originally used inefficient double-iteration patterns that have been optimized to single-pass algorithms with assembly optimizations. The commented-out implementations show O(n²) complexity that has been improved to O(n).

Affected Functions:

1. `facets()` - Gets all facets and their selectors
2. `facetFunctionSelectors()` - Gets selectors for a specific facet

Recommendation:

```
function facets() external override view returns (Facet[] memory facets_) {
    LibSLYDiamond.DiamondStorage storage ds = LibSLYDiamond.diamondStorage();

    uint256 numFacets = ds.facetAddresses.length;
    facets_ = new Facet[](numFacets);

    for (uint256 i = 0; i < numFacets; i++) {
        address currentFacet = ds.facetAddresses[i];
        facets_[i].facetAddress = currentFacet;

        // Memory allocation based on length
        bytes4[] memory selectors = new bytes4[](ds.selectors.length);
        uint256 count = 0;

        // Single iteration to collect selectors
        for (uint256 j = 0; j < ds.selectors.length; j++) {
            if (address(bytes20(ds.facets[ds.selectors[j]])) == currentFacet) {
                selectors[count] = ds.selectors[j];
                count++;
            }
        }

        // Assembly optimization: directly modify array length without reallocation
        assembly {
            mstore(selectors, count)  // Modify length field in memory
        }
```

```
            facets_[i].functionSelectors = selectors;
        }
    }

    function facetFunctionSelectors(address _facet) external override view returns (bytes4[]
    memory functionSelectors_) {
        LibSLYDiamond.DiamondStorage storage ds = LibSLYDiamond.diamondStorage();

        uint256 selectorsLength = ds.selectors.length;
        bytes4[] memory selectors = new bytes4[](selectorsLength);
        uint256 count = 0;

        for (uint256 i = 0; i < selectorsLength; i++) {
            bytes4 selector = ds.selectors[i];
            if (address(bytes20(ds.facets[selector])) == _facet) {
                selectors[count] = selector;
                unchecked { count++; }
            }
        }

        assembly {
            mstore(selectors, count)
        }

        return selectors;
    }
```

Update/Status:

The team has skipped the issue.

[FP-3] DCAPosition lacks tracking of total tokens received

Code Improvement Info Acknowledged

Issue/Risk: Code Improvement

Description:

The current `DCAPosition` struct and `dcaGetPosition` function do not track or return the total amount of target tokens (`tokenOut`) that the user has received through DCA trades. This missing information makes it difficult for users to evaluate their DCA strategy performance, calculate average buy prices, or understand the effectiveness of their positions.

While the `DCATradeExecuted` event does emit the `amountOut` for each individual trade:

```
emit DCATradeExecuted(positionId, tradeAmount, amountOut, price);
```

Users can theoretically reconstruct this information by analyzing historical events off-chain, but having this data directly available through on-chain queries would be much more convenient and user-friendly. Direct on-chain access eliminates the need for complex event parsing and provides immediate access to position performance metrics.

Recommendation:

Option 1: Extend DCAPosition struct

```
struct DCAPosition {
    // ... existing fields ...
    uint256 totalTokensReceived;    // ✅ New: Total target tokens received
}
```

Similar Enhancement for DRIP Strategy: The `DRIPPosition` struct in `SLYWalletDRIPStorage.sol` has the same limitation. It tracks `totalBuyTraded` (amount spent) and `totalSellTraded` (amount sold) but doesn't track the tokens received from buy trades or the funding tokens received from sell trades. Consider applying a similar enhancement:

```
struct DRIPPosition {
    // ... existing fields ...
    uint256 totalBuyTokensReceived;    // Target tokens received from buy trades
    uint256 totalSellTokensReceived;   // Funding tokens received from sell trades
}
```

Update/Status:

The team has skipped the issue.

[FP-4] Authenticator role has excessive permissions allowing complete asset control bypass

Access Control Critical ✓ Fixed

Issue/Risk: Access Control

Description:

In the current implementation, the `Authenticator` role is granted `Execute` and `ExecuteBatch` permissions, which effectively grants nearly unlimited control over the wallet. This completely violates the principle of least privilege and the intended role separation.
The `Authenticator` role, which should only validate signatures according to the interface documentation, can currently:

1. **Bypass asset withdrawal restrictions:**

```
// Direct ETH transfer bypassing withdrawETH permission check
wallet.execute(attacker, ethers.parseEther("1"), "0x");

// Direct ERC20 transfer bypassing withdrawERC20 permission check
wallet.execute(tokenAddress, 0,
    token.interface.encodeFunctionData("transfer", [attacker, amount])
);
```

2. Execute arbitrary external contract calls:

```
// Call any external contract with any parameters
wallet.execute(maliciousContract, 0, maliciousData);
```

Impact:

- Complete loss of all wallet assets (ETH and ERC20 tokens)
- Unauthorized external contract interactions

Recommendation:

1. **Remove Execute permissions from Authenticator role** (Optional).
2. **Implement whitelist-based execution control**

Update:

The team removed the Authenticator role.

Status:

The issue is fixed at ae2f77fd47740f9b5ee57f936994e1ed4de90918.

[FP-5] Reentrancy vulnerability allows single DCA position to be executed multiple times

Code Improvement High ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `dcaExecuteTrades` function lacks reentrancy protection, allowing malicious executors to re-enter the function during fee distribution and execute the same DCA position multiple times in a single transaction. This violates the intended "one execution per trigger block" behavior and can cause financial harm to users.

Vulnerable Code:

```
function dcaExecuteTrades(uint256[] calldata _positionIds) external override initialized {
    // ... position validation and checks ...
    // ❌ Fee distribution occurs before state updates - reentrancy trigger point
    uint256 actualTradeAmount = _calculateAndDistributeFees(
        position.fundingToken,
        tradeAmount,
        msg.sender,  // Attacker-controlled executor address
        positionId
    );

    // Token operations and swap execution
    // ...

    // ❌ State updates happen last - vulnerable to reentrancy
    position.amountTraded += tradeAmount;
    position.nextTriggerBlock = block.number + position.triggerInterval;
}

function _distributeFeeAmounts(...) internal {
    if (_fundingToken == ds.wnative) {
        if (_executorFeeAmount > 0) {
            // ❌ Reentrancy trigger: external call to executor
            (bool success,) = _executor.call{value: _executorFeeAmount}("");
            require(success, "Executor fee transfer failed");
        }
    }
}
```

Attack Mechanism:

1. Malicious executor calls `dcaExecuteTrades([positionId])`
2. During fee distribution, executor's contract receives ETH callback
3. In the `receive()` function, executor re-enters `dcaExecuteTrades([positionId])`
4. Position state hasn't been updated yet, so all checks pass again
5. Same position gets executed twice in one transaction

Recommendation:

Add reentrancy protection such as OpenZeppelin's ReentrancyGuard.

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-6] Missing slippage protection in DCA strategy exposes users to MEV attacks and value loss

Design Vulnerability High ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

The `dcaExecuteTrades` function calls `ThenaLib.exactInputSingle` with `minAmountOut` set to 0, providing no slippage protection. This leaves users vulnerable to MEV sandwich attacks, liquidity manipulation, and significant value loss during trade execution.

Vulnerable Code:

```
// In dcaExecuteTrades function
uint256 amountOut = ThenaLib.exactInputSingle(
    ds.thenaRouter,
    position.fundingToken,
    position.tokenOut,
    ds.thenaDeployer,
    actualTradeAmount,
    0,  // ❌ No slippage protection - accepts any output amount
    address(this),
    block.timestamp + 300
);
```

Recommendation:

Add slippage protection configuration at position creation time:

```
struct DCAPosition {
    ...
    uint256 minAmountOut;            // ✅ User-defined minimum output per trade
    bool isActive;
    uint256 createdAt;
}


// Use in execution
function dcaExecuteTrades(uint256[] calldata _positionIds) external {
    // ...existing logic...

    uint256 amountOut = ThenaLib.exactInputSingle(
        ds.thenaRouter,
        position.fundingToken,
        position.tokenOut,
        ds.thenaDeployer,
        actualTradeAmount,
        position.minAmountOut,  // ✅ User-defined slippage protection
        address(this),
        block.timestamp + 300
    );
}
```

Update/Status:

The issue is fixed at 902a899521a6ea85a8aa9387a7fa8a3106b8117e.

[FP-7] Reentrancy vulnerability allows single DCA position to be executed multiple times

Access Control High ✓ Fixed

Issue/Risk: Access Control

Description:

The `sdrExecuteTrades` function lacks reentrancy protection, allowing malicious executors to re-enter the function during fee distribution and execute the same DRIP position multiple times in a single transaction. This violates the intended "one execution per trigger block" behavior and can cause financial harm to users.

DRIP Strategy Reentrancy Attack Flow

Attack Prerequisites

- Wallet balance: 100 BNB
- Position settings: 10 BNB per trade (10% of balance)
- Estimated fees: 1 BNB (10% fee rate)
- Net swap amount: 9 BNB per execution

Phase 1: Initial Trade Execution

```
_executeSwapWithFees(positionId, position, tokenIn, tokenOut, 10 BNB, isSellOrder,
deviation) {

    // Step 1: Calculate fees (no external calls)
    (totalFee=1, platformFee=0.5, executorFee=0.5) = _calculateFees(BNB, 10);

    // Step 2: Prepare tokens for swap
    swapAmount = 10 - 1 = 9 BNB;
    _prepareTokensForSwap(BNB, 10, 9, wnative, router);
    // ✅ IWETH(wnative).deposit{value: 10}();  // Consumes 10 BNB (100 → 90 BNB)
    // ✅ IERC20(wnative).approve(router, 9);   // Approves 9 WBNB

    // Step 3: Execute swap
    // ✅ executeSwapSafely(router, BNB, tokenOut, deployer, 9)
    //    Returns amountOut, consumes 9 WBNB allowance

    // Step 4: Distribute fees (REENTRANCY TRIGGER POINT)
    _distributeFees(BNB, 0.5, 0.5, msg.sender, feeManager);
    // ❌ executor.call{value: 0.5}(""); // Triggers attacker's receive() function

    // Step 5: Update position state (NOT EXECUTED YET)
    // ❌ _updatePositionAfterTrade() // totalBuyTraded += 10, lastExecutionBlock =
current
}
```

Phase 2: Reentrancy Attack

```
// Attacker's malicious contract receive() function
receive() external payable {
    if (!attacking && msg.value > 0) {
        attacking = true;
        // Re-enter the same function
        target.sdrExecuteTrades([positionId]);
        attacking = false;
    }
}

// Reentrancy execution - SAME FUNCTION CALLED AGAIN
_executeSwapWithFees(positionId, position, tokenIn, tokenOut, 10 BNB, isSellOrder,
deviation) {

    // Step 1: Calculate fees (position state unchanged, same calculations)
    (totalFee=1, platformFee=0.5, executorFee=0.5) = _calculateFees(BNB, 10);

    // Step 2: Prepare tokens for swap
    swapAmount = 10 - 1 = 9 BNB;
    _prepareTokensForSwap(BNB, 10, 9, wnative, router);

    // Current wallet state:
    // - Native BNB balance: 90 BNB (after first deposit)
    // - WBNB balance: 1 WBNB (10 deposited - 9 swapped)

    // ✅ IWETH(wnative).deposit{value: 10}();  // SUCCESS: 90 BNB → 80 BNB
    // ✅ IERC20(wnative).approve(router, 9);   // SUCCESS: 1+10=11 WBNB, approve 9

    // Step 3: Execute swap
    // ✅ executeSwapSafely(router, BNB, tokenOut, deployer, 9)
    //    SUCCESS: 11 WBNB > 9 WBNB required

    // Step 4: Distribute fees
    _distributeFees(BNB, 0.5, 0.5, msg.sender, feeManager);
    // ✅ Normal execution (no nested reentrancy)

    // Step 5: Update position state
    // ✅ _updatePositionAfterTrade() // totalBuyTraded += 10 = 10
}

// Return to Phase 1, Step 5
// ✅ _updatePositionAfterTrade() // totalBuyTraded += 10 = 20 (DOUBLE INCREMENT!)
```

Recommendation:

Add reentrancy protection such as OpenZeppelin's ReentrancyGuard.

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-8] Missing slippage protection in DRIP strategy exposes users to MEV attacks and value loss

Code Improvement High ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `executeSwapSafely` function calls `ThenaLib.exactInputSingle` with `minAmountOut` set to 0, providing no slippage protection. This leaves users vulnerable to MEV sandwich attacks, liquidity manipulation, and significant value loss during trade execution.

Vulnerable Code:

```
function executeSwapSafely(
      address router,
      address tokenIn,
      address tokenOut,
      address deployer,
      uint256 amountIn
  ) external returns (uint256 amountOut) {
      require(msg.sender == address(this), "SLYDRIP: Internal function only");

      return ThenaLib.exactInputSingle(
          router, tokenIn, tokenOut, deployer, amountIn, 0, address(this), block.timestamp
+ 300
      );
  }
```

Recommendation:

Add slippage protection configuration at position creation time:

```
// DRIP position struct
  struct DRIPPosition {
      uint256 minAmountOut;          // ✅ User-defined minimum output per trade
      bool isActive;
      uint256 createdAt;
      bool isActive;                          // Whether the position is active
      uint256 totalFeesCollected;             // Total fees collected from this position
      uint256 lastExecutionBlock;             // Last block when trade was executed
(cooldown)
  }

  function executeSwapSafely(
      address router,
      address tokenIn,
      address tokenOut,
      address deployer,
      uint256 amountIn,
      uint256 minAmountOut
  ) external returns (uint256 amountOut) {
      require(msg.sender == address(this), "SLYDRIP: Internal function only");

      return ThenaLib.exactInputSingle(
          router, tokenIn, tokenOut, deployer, amountIn, minAmountOut, address(this),
block.timestamp + 300
      );
  }
```

Update/Status:

The issue is fixed at 902a899521a6ea85a8aa9387a7fa8a3106b8117e.

[FP-9] DRIP strategy receives wnative token instead of native token without auto-conversion, causing gradual native token depletion

Code Improvement High ✓ Fixed

Issue/Risk: Code Improvement

Description:

When DRIP positions execute "sell to native token" trades (e.g., USDT→BNB), the strategy receives WBNB from the DEX but fails to convert it back to native BNB. This creates a one-way conversion pattern where native BNB is consumed for purchases but never replenished, eventually breaking BNB-denominated strategies.

DRIP Strategy Issue:

```
// In executeSwapSafely
function executeSwapSafely(
    address router,
    address tokenIn,
    address tokenOut,  // When this is WNATIVE
    address deployer,
    uint256 amountIn
) external returns (uint256 amountOut) {
    // ❌ Calls exactInputSingle without handling WBNB→BNB conversion
    return ThenaLib.exactInputSingle(
        router, tokenIn, tokenOut, deployer, amountIn, 0,
        address(this),  // Receives WBNB, not native BNB
        block.timestamp + 300
    );
    // ❌ No conversion: WBNB remains as WBNB
}

// But exactInputSingle has no such mechanism for general use:
function exactInputSingle(...) internal returns (uint256 amountOut) {
    // ... router setup ...
    return ISwapRouter(swapRouter).exactInputSingle(params);
    // ❌ No post-processing for WBNB→BNB conversion
}
```

Recommendation:

Option 1: Fix executeSwapSafely in DRIP

```
function executeSwapSafely(
    address router,
    address tokenIn,
    address tokenOut,
    address deployer,
    uint256 amountIn
) external returns (uint256 amountOut) {
    amountOut = ThenaLib.exactInputSingle(
        router, tokenIn, tokenOut, deployer, amountIn, 0,
        address(this), block.timestamp + 300
    );

    // ✅ Auto-convert WBNB to native BNB when needed
    SLYWalletDRIPStorage.Data storage ds = SLYWalletDRIPStorage.diamondStorage();
    if (tokenOut == ds.wnative && amountOut > 0) {
        IWETH(ds.wnative).withdraw(amountOut);
    }

    return amountOut;
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-10] Failed swaps leave irreversible side effects from token preparation without cleanup mechanism

Code Improvement High ✓ Fixed

Issue/Risk: Code Improvement

Description:

When `executeSwapSafely` fails in `_executeSwapWithFees`, the function catches the error but doesn't revert the side effects caused by `_prepareTokensForSwap`. This results in permanent native token conversions and unnecessary token approvals that persist even when trades fail.

Vulnerable Code Flow:

```
function _executeSwapWithFees(...) internal {
    // 1. Calculate fees (no side effects)
    (uint256 totalFee, uint256 platformFee, uint256 executorFee) = _calculateFees(tokenIn,
tradeAmount);
    uint256 swapAmount = tradeAmount - totalFee;

    // 2. ❌ Irreversible side effects occur here
    _prepareTokensForSwap(tokenIn, tradeAmount, swapAmount, ds.wnative, ds.thenaRouter);

    // 3. ❌ If swap fails, side effects are NOT cleaned up
    try this.executeSwapSafely(ds.thenaRouter, tokenIn, tokenOut, ds.thenaDeployer,
swapAmount)
    returns (uint256 amountOut) {
        // Success path...
    } catch Error(string memory reason) {
        // ❌ Emit event and exit - NO CLEANUP
        emit SDRTradeFailed(positionId, position.targetToken, swapAmount, reason);
        return; // Function exits with permanent state changes
    }
}

// Side Effects in _prepareTokensForSwap:
function _prepareTokensForSwap(
    address tokenIn,
    uint256 tradeAmount,    // e.g., 10 BNB
    uint256 swapAmount,     // e.g., 9 BNB (after fees)
    address wnative,
    address router
) internal {
    if (tokenIn == wnative) {
        // ❌ Side Effect 1: Irreversible BNB → WBNB conversion
        IWETH(wnative).deposit{value: tradeAmount}();      // Consumes 10 BNB

        // ❌ Side Effect 2: Token approval
        IERC20(wnative).approve(router, swapAmount);       // Approves 9 WBNB
    } else {
        // ❌ Side Effect 3: Token approval for ERC20
        IERC20(tokenIn).approve(router, swapAmount);       // Approves swapAmount
    }
}
```

Repeated trade failures within a short timeframe will deplete the wallet's native BNB balance through multiple BNB-to-WBNB conversions, potentially causing subsequent orders to fail due to insufficient native token liquidity.

Recommendation:

Option 1: Revert entire transaction on swap failure

```
function _executeSwapWithFees(...) internal {
    (uint256 totalFee, uint256 platformFee, uint256 executorFee) = _calculateFees(tokenIn,
tradeAmount);
    uint256 swapAmount = tradeAmount - totalFee;

    _prepareTokensForSwap(tokenIn, tradeAmount, swapAmount, ds.wnative, ds.thenaRouter);

    // ✅ Remove try-catch, let failures revert the entire transaction
    uint256 amountOut = this.executeSwapSafely(ds.thenaRouter, tokenIn, tokenOut,
ds.thenaDeployer, swapAmount);

    // Success-only code...
    _distributeFees(tokenIn, platformFee, executorFee, msg.sender, feeManager);
    _updatePositionAfterTrade(positionId, position, tradeAmount, totalFee, isSellOrder);
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-11] Critical facets can be arbitrarily removed, bricking the wallet

Code Improvement High ✓ Fixed

Issue/Risk: Code Improvement

Description:

In `LibSLYDiamond` the `removeFunctions()` function is used to remove interfaces from certain facets. However, there is **no validation** to prevent the removal of critical, non-removable interfaces such as those from `SLYDiamondCutFacet` and `SLYWalletBaseFacet`.

If a user accidentally removes function selectors from these critical facets, the **entire wallet becomes permanently unusable** (bricked), as there would be no way to:

1. **Perform further diamond cuts** (if `diamondCut()` is removed from `SLYDiamondCutFacet`)
2. **Manage keys** (if `addKey()`, `removeKey()` are removed from `SLYWalletBaseFacet`)
3. **Execute transactions** (if `execute()`, `executeBatch()` are removed)
4. **Withdraw assets** (if `withdrawETH()`, `withdrawERC20()` are removed)

Recommendation:

Add protected function selectors.

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-12] `withdrawERC20` incompatible with non-standard ERC20 tokens like USDT

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `withdrawERC20` function directly uses `token.transfer()` and checks its return value, which is incompatible with non-standard ERC20 tokens that don't return boolean values or return inconsistent values.

Vulnerable Code:

```
function withdrawERC20(address _token, address _to, uint256 _amount) external {
    // ...
    IERC20 token = IERC20(_token);
    success = token.transfer(_to, _amount);  // ❌ Incompatible with non-standard tokens
    require(success, "SLYWallet: token transfer failed");
    // ...
}
```

Recommendation:

Use OpenZeppelin's `SafeERC20` library which handles all these edge cases:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
using SafeERC20 for IERC20;
// ✅ Use SafeERC20, automatically handles return value issues
token.safeTransfer(_to, _amount);
```

Update/Status:

The issue is fixed at ae2f77fd47740f9b5ee57f936994e1ed4de90918.

[FP-13] Zero address facets create irreversible function selector occupation

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `addFunctions` function in `LibSLYDiamond` allows adding function selectors with zero address as the facet address, but `replaceFunctions` explicitly rejects zero addresses. This asymmetry can lead to permanent occupation of function selectors.
Vulnerable Code:

```
// addFunctions - allows zero address
function addFunctions(address _facetAddress, bytes4[] memory _functionSelectors) internal
{
    ds.facets[selector] = bytes20(_facetAddress);  // Can be address(0)
}

// replaceFunctions - rejects zero address
function replaceFunctions(address _facetAddress, bytes4[] memory _functionSelectors)
internal {
    require(_facetAddress != address(0), "LibSLYDiamond: Add facet can't be address(0)");
}
```

Impact:

- **Permanent function selector occupation**: Specific functions become unimplementable
- **Denial of service**: Targeted functionality may become permanently unavailable
- **Upgrade interference**: Could block specific system upgrades
- **Administrative complications**: May require complex workarounds

Recommendation:

Add zero address validation to `addFunctions` function:

```
function addFunctions(address _facetAddress, bytes4[] memory _functionSelectors) internal
{
    require(_functionSelectors.length > 0, "LibSLYDiamond: No selectors in facet to cut");
    require(_facetAddress != address(0), "LibSLYDiamond: Cannot add functions to zero
address");

    // ... rest of implementation
}
```

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-14] `removeFunctions` lacks facet address validation allowing unauthorized function removal

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

** The `removeFunctions` function does not verify that the provided `_facetAddress` parameter matches the actual facet address of the functions being removed. This allows removal of functions belonging to any facet by specifying an arbitrary `_facetAddress`.

Vulnerable Code:

```
function removeFunctions(address _facetAddress, bytes4[] memory _functionSelectors)
internal {
    for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;
selectorIndex++) {
        bytes4 selector = _functionSelectors[selectorIndex];
        address oldFacetAddress = address(bytes20(ds.facets[selector]));
        require(oldFacetAddress != address(0), "LibSLYDiamond: Function doesn't exist");

        // ❌ Missing validation: oldFacetAddress should equal _facetAddress
        delete ds.facets[selector];
    }
    // Cleanup logic checks _facetAddress, not the actual oldFacetAddress
}
```

Risk Scenario:

```
// Functions currently mapped:
// selector1 -> facetA
// selector2 -> facetB

// Privileged user mistakenly calls with wrong facet address:
removeFunctions(facetC, [selector1, selector2]);

// Result: Functions are removed but cleanup logic is incorrect
```

Impact:

- **Incorrect function removal**: Functions can be removed with wrong facet parameter
- **Cleanup logic bypass**: Old facets may not be properly cleaned up
- **Data inconsistency**: Mismatch between intended and actual operations

Recommendation:

Add validation to ensure `_facetAddress` matches the actual facet of each selector:

```
function removeFunctions(address _facetAddress, bytes4[] memory _functionSelectors)
internal {
    for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;
selectorIndex++) {
        bytes4 selector = _functionSelectors[selectorIndex];
        address oldFacetAddress = address(bytes20(ds.facets[selector]));
        require(oldFacetAddress != address(0), "LibSLYDiamond: Function doesn't exist");
        require(oldFacetAddress == _facetAddress, "LibSLYDiamond: Function belongs to
different facet");

        delete ds.facets[selector];
    }
}
```

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-15] Insufficient precision in block time calculation for modern high-speed blockchains

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

The current implementation of `dcaInitialize()` and `dcaUpdateAverageBlockTime()` functions uses seconds as the unit for `_averageBlockTime`, which lacks sufficient precision for modern high-speed blockchains that have sub-second block times.

Current Implementation:

```
function dcaInitialize(
    // ...
    uint256 _averageBlockTime  // ❌ Expects seconds (e.g., 3 for BNB)
) external override onlyOwnerOrAdmin {
    // ...
    ds.averageBlockTime = _averageBlockTime;

    // ❌ Calculation assumes _averageBlockTime is in seconds
    ds.hourlyBlocks = 3600 / _averageBlockTime;     // 1 hour in seconds / block time
    ds.dailyBlocks = 86400 / _averageBlockTime;     // 24 hours in seconds / block time
    ds.weeklyBlocks = 604800 / _averageBlockTime;   // 7 days in seconds / block time
    ds.monthlyBlocks = 2592000 / _averageBlockTime; // 30 days in seconds / block time
}
```

Real-world Block Times:

- **BNB Chain**: ~750ms (0.75 seconds) per block

Precision Issues:

```
// Current approach for BNB Chain (750ms blocks):
uint256 averageBlockTime = 1; // ❌ Forced to round to 1 second, loses precision

// Calculation results:
hourlyBlocks = 3600 / 1 = 3600 blocks  // ❌ Should be 4800 blocks
dailyBlocks = 86400 / 1 = 86400 blocks // ❌ Should be 115200 blocks
```

Recommendation:

Change `_averageBlockTime` to use milliseconds for better precision:

```
/**
 * @dev Initialize the DCA strategy facet
 * @param _averageBlockTime Average block time in milliseconds (e.g., 750 for BNB, 2100
for Polygon)
 */
function dcaInitialize(
    address _thenaRouter,
    address _thenaDeployer,
    address _wnative,
    address _priceOracle,
    uint256 _averageBlockTime  // ✅ Now in milliseconds
) external override onlyOwnerOrAdmin {
     // ✅ Calculate block intervals with millisecond precision
    ds.hourlyBlocks = (3600 * 1000) / _averageBlockTime;     // 1 hour in ms / block time
in ms
    ds.dailyBlocks = (86400 * 1000) / _averageBlockTime;     // 24 hours in ms / block time
in ms
    ds.weeklyBlocks = (604800 * 1000) / _averageBlockTime;   // 7 days in ms / block time
in ms
    ds.monthlyBlocks = (2592000 * 1000) / _averageBlockTime; // 30 days in ms / block time
in ms

}

// SLYWalletDCAStorage.sol
struct Data {
    // Version
    string version;

    // Configuration params
    uint256 averageBlockTime; // ✅ Average block time in milliseconds (e.g., 750 for BNB,
2100 for Polygon)
    address thenaRouter;      // Address of the Thena Router
    address thenaDeployer;    // Address of the Thena Deployer
    address wnative;          // Wrapped native token address (e.g., WBNB)
    address priceOracle;      // Address of the price oracle
    // Trigger intervals (in blocks)
    uint256 hourlyBlocks;     // ✅ ~4800 for BNB (3600*1000/750), ~1714 for Polygon
(3600*1000/2100)
    uint256 dailyBlocks;      // ✅ ~115200 for BNB (86400*1000/750), ~41142 for Polygon
(86400*1000/2100)
    uint256 weeklyBlocks;     // ✅ ~806400 for BNB (604800*1000/750), ~288000 for Polygon
(604800*1000/2100)
    uint256 monthlyBlocks;    // ✅ ~3456000 for BNB (2592000*1000/750), ~1234285 for
Polygon (2592000*1000/2100)

    // DCA positions
    DCAPosition[] positions;
    // ... rest unchanged
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-16] Global `thenaDeployer` configuration limits pool type flexibility in DCA/DRIP positions

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

The current design stores `thenaDeployer` as a global configuration parameter, which limits users to a single pool type (either standard pools or custom pools) across all DCA positions. This reduces flexibility and prevents optimal pool selection for different token pairs.

Current Limitation:

```
struct Data {
    // ...
    address thenaRouter;      // Global router - appropriate
    address thenaDeployer;    // ❌ Global deployer - too restrictive
    // ...
}

struct DCAPosition {
    // ...
    // ❌ No deployer field - must use global setting
}
```

Algebra DEX Pool Types:

AlgebraFactory: https://basescan.org/address/0xEFCB993e113ea8197C17c6f4959495929Be0B68e#code

Standard Pools:

- `deployer = address(0)` (no permission)

Custom Pools:

- `deployer = custom address` (with creation permissions)

Problems with Current Design:

Limited pool selection: Users cannot mix standard and custom pools

Recommendation:

Move `deployer` configuration to individual DCA positions:

```
struct DCAPosition {
    uint256 minAmountOut;
    address deployer;         // ✅ Pool deployer for this position
    bool isActive;
    uint256 createdAt;
}


struct Data {
    // Version
    string version;

    // Configuration params
    uint256 averageBlockTime;
    address thenaRouter;      // ✅ Keep router global (universal)
    // Remove: address thenaDeployer;  // ❌ Remove global deployer
    address wnative;
    address priceOracle;

    // ... rest unchanged
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-17] Confusing logic and naming in `_hasAllocation` function requires clarification

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `_hasAllocation` function contains complex boolean logic with double negations that is difficult to understand and verify. The variable names don't clearly reflect their actual logical meaning, making code maintenance and auditing challenging.

Problematic Code:

```
function _hasAllocation(SLYWalletDRIPStorage.DRIPPosition storage position) internal view
returns (bool) {
    bool hasBuy = !SLYWalletDRIPStorage.includesBuying(position.triggerMode) ||
        position.totalBuyTraded < position.buyFundingAmount;
    bool hasSell = !SLYWalletDRIPStorage.includesSelling(position.triggerMode) ||
        position.totalSellTraded < position.sellFundingAmount;
    return hasBuy || hasSell;
}
```

Confusion Points:

1. **Counterintuitive variable values:** For a sell-only strategy (RIP_ONLY), `hasBuy` evaluates to `true` even though the strategy doesn't involve buying
2. **Complex boolean logic:** The combination of negations (`!includesBuying`) with OR operators creates non-obvious logic flow
3. **Unclear function purpose:** The function name suggests checking allocation availability, but the logic appears to check something different

Recommendation:

Option 1: Add detailed comments

```
function _hasAllocation(SLYWalletDRIPStorage.DRIPPosition storage position) internal view
returns (bool) {
    // TODO: Add clear comments explaining the logic
    // This function checks if position can continue execution (not blocked by completion)
    // - For directions not included in strategy: always "not blocking" (true)
    // - For directions included in strategy: "not blocking" only if not completed

    bool hasBuy = !SLYWalletDRIPStorage.includesBuying(position.triggerMode) ||
        position.totalBuyTraded < position.buyFundingAmount;
    bool hasSell = !SLYWalletDRIPStorage.includesSelling(position.triggerMode) ||
        position.totalSellTraded < position.sellFundingAmount;
    return hasBuy || hasSell;
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-18] DRIP positions with percentage-based trading cannot auto-complete due to geometric convergence leaving dust amounts

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

When DRIP positions use `percentagePerTrade < 100%`, the geometric convergence of trade amounts eventually results in dust amounts that are too small to trade, preventing positions from auto-completing. Users must manually cancel such positions to recover remaining funds, creating poor user experience and trapping small amounts of capital.

Root Cause: The percentage-based trading creates a geometric sequence where each trade uses a percentage of the *remaining* allocation rather than the original allocation. This mathematically ensures that some amount will always remain, no matter how many trades are executed.

Mathematical Analysis:

```
After N trades: remaining = originalAmount × (1 - P/100)^N
Where P = percentagePerTrade

As N increases, remaining approaches 0 but never reaches 0
Eventually: tradeAmount = remaining × P/100 becomes so small it rounds to 0
```

Code Flow:

```
function _executeTrade(...) internal {
    // remaining becomes very small after many trades
    uint256 remaining = position.buyFundingAmount - position.totalBuyTraded;
    if (availableBalance > remaining) availableBalance = remaining;

    // tradeAmount becomes 0 due to dust amounts
    uint256 tradeAmount = SLYWalletDRIPStorage.calculateTradeAmount(availableBalance,
position.percentagePerTrade);

    if (tradeAmount == 0) {
        emit SDRInsufficientBalance(positionId, tokenIn, 0, availableBalance);
        return; // ❌ Position stuck - cannot auto-complete
    }
}
```

Recommendation:

Option 1: Completion Threshold (Recommended)

```
function _isCompleted(...) internal view returns (bool) {
    uint256 buyThreshold = position.buyFundingAmount / 1000; // 0.1%
    uint256 sellThreshold = position.sellFundingAmount / 1000;

    bool buyDone = !includesBuying(position.triggerMode) ||
        (position.buyFundingAmount - position.totalBuyTraded) <= buyThreshold;
    bool sellDone = !includesSelling(position.triggerMode) ||
        (position.sellFundingAmount - position.totalSellTraded) <= sellThreshold;
    return buyDone && sellDone;
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-19] `totalFeesCollected` field aggregates fees from different tokens making the value meaningless

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `DRIPPosition.totalFeesCollected` field accumulates fee amounts from different token types (funding token and target token) into a single uint256 value. Since DRIP strategies involve bidirectional trading, fees are collected in both tokens, making the aggregated value mathematically meaningless and unusable for analysis.

```
struct DRIPPosition {
    address fundingToken;       // e.g., USDT
    address targetToken;        // e.g., ETH
    // ...
    uint256 totalFeesCollected;    // ❌ Aggregates fees from both tokens
    // ...
}

function _updatePositionAfterTrade(
    uint256 positionId,
    DRIPPosition storage position,
    uint256 tradeAmount,
    uint256 feeAmount,             // Could be USDT or ETH
    bool isSellOrder
) internal {
    // ❌ Directly adds different token amounts
    position.totalFeesCollected += feeAmount;
}
```

Problem Scenarios:

Scenario 1: Mixed token fees

```
// Buy trade: 1000 USDT → ETH, fee = 10 USDT
position.totalFeesCollected += 10;  // = 10

// Sell trade: 0.5 ETH → USDT, fee = 0.005 ETH
position.totalFeesCollected += 0.005;  // = 10.005

// Result: 10.005 (meaningless - not USDT, not ETH, not USD)
```

Recommendation:

Option 1: Separate fee tracking

```
struct DRIPPosition {
    // ...
    uint256 totalFundingTokenFeesCollected;  // Fees paid in funding token
    uint256 totalTargetTokenFeesCollected;   // Fees paid in target token
    // Remove: uint256 totalFeesCollected;
}

function _updatePositionAfterTrade(...) internal {
    if (isSellOrder) {
        // Selling target token, fee deducted from target token
        position.totalTargetTokenFeesCollected += feeAmount;
    } else {
        // Buying with funding token, fee deducted from funding token
        position.totalFundingTokenFeesCollected += feeAmount;
    }
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-20] `withdrawVToken` can be replaced by `withdrawERC20`

Admin Rights Medium ✓ Fixed

Issue/Risk: Admin Rights

Description:

In `SLYWalletVenusFacet.sol`, there exists a `withdrawVToken` function with the designed permission requirement of `onlyAdmin`. However, in the `SLYWalletBaseFacet` contract, the `withdrawERC20`, `execute`, and `executeBatch` functions can all achieve the same functionality. These three functions require `Permission.WithdrawAssets`, `Permission.Execute`, and `Permission.ExecuteBatch` permissions respectively, which are completely different from the permission design of `withdrawVToken`. Users with `Permission.WithdrawAssets` permission can call `withdrawERC20` to withdraw `vToken` without needing to call the `withdrawVToken` function.

Recommendation:

1. Carefully consider permission issues for various operations at the design level.
2. Although this file is not within the audit scope, we still performed an interface browsing. We recommend conducting an internal audit or another third-party audit of this out-of-scope code. For example:
    - Block initialization of this contract in the constructor, since this contract is only a template.
    - The receive function already exists in `SLYWalletDiamond`, so it can be removed from this contract to prevent unnecessary transactions transferring funds to the template contract.
    - Detailed logic checks for Venus protocol interactions.
    - Overall consideration and inspection of whether there are conflicts or overlaps in functionality design between facets.

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-21] `addKey` function lacks validation for `Role.None`, potentially allowing invalid key additions

Code Improvement Low ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `addKey` function in `SLYWalletBaseFacet` only validates the upper bound of the role parameter but doesn't prevent adding keys with `Role.None` .While this doesn't pose security risks, it can lead to data pollution where addresses are stored with meaningless roles, creating inconsistent state and potential confusion.

Current Implementation:

```
function addKey(address _key, Role _role) external override
hasPermission(Permission.AddKey) {
    require(_key != address(0), "SLYWallet: key cannot be zero address");
    require(!keyExists(_key), "SLYWallet: key already exists");
    require(_role < Role.RoleCount, "SLYWallet: Invalid role"); // ❌ Only upper bound
check

    // ... rest of logic
}
**Potential Issues:**
1. **State inconsistency**: Keys stored with `Role.None` have no meaningful permissions
2. **Logic confusion**: `keyExists` behavior depends on implementation details
3. **Unexpected behavior**: Added keys that appear to exist but have no functional purpose
4. **Wasted storage**: Unnecessary storage usage for non-functional keys
```

Recommendation:

```
function addKey(address _key, Role _role) external override
hasPermission(Permission.AddKey) {
    require(_key != address(0), "SLYWallet: key cannot be zero address");
    require(!keyExists(_key), "SLYWallet: key already exists");
    require(_role > Role.None, "SLYWallet: Cannot add key with None role");     // ✅
Lower bound
    require(_role < Role.RoleCount, "SLYWallet: Invalid role");                 // ✅
Upper bound

    // ... rest of logic
}
```

Update/Status:

The issue is fixed at ae2f77fd47740f9b5ee57f936994e1ed4de90918.

[FP-22] No pagination mechanism implemented

Design Vulnerability Low ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

The `getAllWallets` function returns a complete array of all wallet addresses. When the number of wallets becomes large (estimated 10,000-50,000), even as a view function it may fail due to the following reasons:

- Exceeding RPC call gas limits
- Exceeding client response size limits
- Insufficient memory to construct large return arrays

Recommendation:

Additionally implement a function that supports paginated retrieval.

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-23] No deduplication of owners during wallet creation

Code Improvement Low ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `createSLYWalletWithSalt` and `createSLYWallet` functions do not perform deduplication checks on initial owners when creating wallets, allowing duplicate owners to be added.

Recommendation:

1. Adopt a Safe-like approach: sort owners on the frontend and implement deduplication validation in `createSLYWallet` and `createSLYWalletWithSalt`. (Optional)
2. Modify the `_addKey` function in `SLYWalletDiamondInit` to add `require(ds.roleData.roles[_key] == Role.None)` before `ds.roleData.roles[_key] = _role;`

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-24] `replaceFunctions` lacks old facet cleanup logic causing storage bloat

Code Improvement Low ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `replaceFunctions` function does not check and remove old facet addresses from the `facetAddresses` array when they no longer have any function selectors, unlike `removeFunctions` which properly implements this cleanup.

Code Comparison:

```
// removeFunctions - has proper cleanup ✅
if(!facetStillHasSelectors) {
    // Remove facet from facetAddresses array
    for(uint256 i = 0; i < ds.facetAddresses.length; i++) {
        if(ds.facetAddresses[i] == _facetAddress) {
            ds.facetAddresses[i] = ds.facetAddresses[ds.facetAddresses.length - 1];
            ds.facetAddresses.pop();
            break;
        }
    }
}

// replaceFunctions - missing cleanup ❌
// No cleanup logic for old facet addresses
```

Recommendation:

dd old facet cleanup logic to `replaceFunctions` similar to `removeFunctions`.

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-25] Incorrect length validation in `batchDiamondCutWithMultipleInits` causing transaction failures

Code Improvement   Low   ✓ Fixed

Issue/Risk: Code Improvement

Description:

In the `batchDiamondCutWithMultipleInits` function, the calldatas's length validation logic is incorrect. The function checks `calldatas.length == 0 || calldatas.length == facetCuts.length` instead of `calldatas.length == inits.length`, which can lead to mismatched array access and transaction failures.

Problem Scenarios:

Scenario : Empty calldatas array

```
// Input parameters:
// inits = [initContract1, initContract2]     // length = 2, non-zero addresses
// calldatas = []                            // length = 0, empty array

bytes memory calldata_ = calldatas.length > 0 ? calldatas[i] : new bytes(0); // set
calldata to empty


function initializeDiamondCut(address _init, bytes memory _calldata) internal {
    if (_init == address(0)) {
        return;
    }
    // will be revert
    require(_calldata.length > 0, "LibSLYDiamond: _calldata is empty");

    // Execute initialization
    (bool success, bytes memory error) = _init.delegatecall(_calldata);
    if (!success) {
        if (error.length > 0) {
            // Bubble up the error
            assembly {
                let returndata_size := mload(error)
                revert(add(32, error), returndata_size)
            }
        } else {
            revert("LibSLYDiamond: _init function reverted");
        }
    }
}
```

Recommendation:

Fix the length validation to properly check array length consistency:

```
function batchDiamondCutWithMultipleInits(
        FacetCut[] calldata facetCuts,
        address[] calldata inits,
        bytes[] calldata calldatas,
        string calldata description
    ) external {
       LibSLYDiamond.enforceHasDiamondCutPermission();
       require(facetCuts.length > 0, "SLYDiamondCut: empty batch");
       require(
            inits.length == 0 || inits.length == facetCuts.length,
            "SLYDiamondCut: inits length mismatch"
       );
       require(calldatas.length == inits.length,"SLYDiamondCut: calldatas length
mismatch");
       ....
}
```

Update/Status:

The issue is fixed at 779465cd2b3ac0ad787099b5da62e9883d83deda.

[FP-26] Unnecessary complexity in permission system - redundant `permissions` mapping

Code Improvement Info ✓ Fixed

Issue/Risk: Code Improvement

Description:

In the `SLYWalletStorage` contract, the `RoleData` struct contains both `roles` mapping and `permissions` mapping:

```
struct RoleData {
    mapping(address => ISLYWalletBase.Role) roles;
    mapping(address => mapping(ISLYWalletBase.Permission => bool)) permissions;  //
Redundant
    mapping(ISLYWalletBase.Role => address[]) keysByRole;
    mapping(address => uint256) nonces;
}
```

While the `permissions` mapping is used in the codebase (set during role assignment and checked in some functions), it introduces unnecessary complexity because permissions are entirely determined by roles. The current implementation duplicates permission information - once through role assignment and again through individual permission flags.

The current `LibPermissions.hasPermission()` implementation already provides complete permission control based on roles:

- `Owner`: Has all permissions
- `Admin`: Has all permissions except `RemoveKey` and `DiamondCut`
- `Authenticator`: Has `ValidateSignature`, `Execute`, and `ExecuteBatch` permissions

Since permissions are deterministic based on roles, maintaining a separate `permissions` mapping adds complexity without providing additional value.

Impact:

- Unnecessary storage overhead
- Increased deployment and operation costs
- Code complexity and potential maintenance issues
- Risk of inconsistency between role-based and permission-based checks

Recommendation:

1. Remove the `permissions` mapping from `RoleData` struct
2. Remove any references to direct permission setting/getting functions if they exist
3. Rely entirely on `LibPermissions.hasPermission()` for all permission checks
4. Update any initialization code that might set individual permissions

Code Change:

```
struct RoleData {
    mapping(address => ISLYWalletBase.Role) roles;
    // mapping(address => mapping(ISLYWalletBase.Permission => bool)) permissions; //
Remove this line
    mapping(ISLYWalletBase.Role => address[]) keysByRole;
    mapping(address => uint256) nonces;
}

// SLYWalletDiamondInit
function _addKey(
    SLYWalletStorage.Data storage ds,
    address _key,
    ISLYWalletBase.Role _role
) internal {
    // deduplication
    require(ds.roleData.roles[_key] == ISLYWalletBase.Role.None, "Duplicate key")
    // Set role
    ds.roleData.roles[_key] = _role;

    // Add to role array
    ds.roleData.keysByRole[_role].push(_key);

    // ✅ So Simple!
    emit KeyAdded(_key, _role);
}
```

This simplification will make the permission system more maintainable and reduce the risk of inconsistencies between role-based and permission-based access control.

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-27] Unnecessary public visibility of `hasPermission` function in `LibPermissions`

Code Improvement Info ✓ Fixed

Issue/Risk: Code Improvement

Description:

In the `LibPermissions` library, the `hasPermission` function is currently declared as `public`:

```
function hasPermission(address _key, uint8 _permission) public view returns (bool) {
    // ... implementation
}
```

When library functions are declared as `public`, Solidity requires the library to be deployed as a separate contract and linked to consuming contracts. This creates several inefficiencies compared to `internal` functions.

Issues with `public` library functions:

1. **Separate deployment required**: The library must be deployed independently before deploying contracts that use it
2. **External linking complexity**: Consuming contracts must be linked to the deployed library address
3. **Increased gas costs**: Function calls use `DELEGATECALL` which is more expensive than direct inline code
4. **Deployment complexity**: Requires additional deployment steps and dependency management
5. **Runtime overhead**: External calls have additional gas overhead compared to inlined code

Current Impact:

- `LibPermissions` must be deployed separately before any facet deployment
- All facets using `hasPermission` require library linking
- Each call to `hasPermission` incurs `DELEGATECALL` gas overhead
- Increased operational complexity for deployment and upgrades

Recommendation:

Change the visibility of library functions from `public` to `internal`:

```
function hasPermission(address _key, uint8 _permission) internal view returns (bool) {
    SLYWalletStorage.Data storage ds = SLYWalletStorage.diamondStorage();
    ISLYWalletBase.Role role = ds.roleData.roles[_key];
    // ... rest of implementation
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-28] Overly complex permission check

Access Control Info ✓ Fixed

Issue/Risk: Access Control

Description:

In the `LibSLYDiamond` library, the `enforceHasDiamondCutPermission` function implements its own permission checking logic instead of using the standardized `LibPermissions.hasPermission()` function:

```
function enforceHasDiamondCutPermission() internal view {
    // ... initialization checks ...

    // ❌ Custom permission checking logic
    bool hasDiamondCutPermission = false;

    // Owner role has all permissions
    if (walletDs.roleData.roles[msg.sender] == ISLYWalletBase.Role.Owner) {
        hasDiamondCutPermission = true;
    } else {
        // Otherwise check for specific DiamondCut permission
        hasDiamondCutPermission = walletDs.roleData.permissions[msg.sender]
[ISLYWalletBase.Permission.DiamondCut];
    }

    require(hasDiamondCutPermission, "LibSLYDiamond: caller does not have DiamondCut
permission");
}
```

This approach is inconsistent with the rest of the codebase, which uses `LibPermissions.hasPermission()` for permission checks. The custom implementation duplicates the permission logic and creates maintenance overhead.

Recommendation:

Prerequisite: First implement the change from Issue - change `LibPermissions.hasPermission()` visibility from `public` to `internal` to enable proper library usage.

Replace the custom permission checking logic with the standardized `LibPermissions.hasPermission()` function:

```
import "../libraries/LibPermissions.sol";

using LibPermissions for address;

function enforceHasDiamondCutPermission() internal view {
    SLYWalletStorage.Data storage walletDs = SLYWalletStorage.diamondStorage();

    // Only allow diamond cuts during constructor (initial setup)
    if (address(this).code.length == 0) {
        return;
    }

    // If wallet is not initialized, only allow initialization
    if (!walletDs.initialized) {
        return;
    }

    // ✅ Use standardized permission checking
    require(
        msg.sender.hasPermission(uint8(ISLYWalletBase.Permission.DiamondCut)),
        "LibSLYDiamond: caller does not have DiamondCut permission"
    );
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-29] `_checkMarketAlignment` function has poor readability due to nested try-catch blocks

Code Improvement Info ✓ Fixed

Issue/Risk: Code Improvement

Description:

The `_checkMarketAlignment` function uses deeply nested try-catch blocks that significantly impact code readability and maintainability. The function contains triple-nested try-catch structures that make it difficult to follow the logic flow and prone to errors during modifications.

To improve code quality, we can extend the existing `LibSPFStructs.Trend` enum by adding an `UNKNOWN` value to handle call failures, then simplify the function by leveraging this mechanism instead of complex exception handling patterns.

Current Problematic Code:

```
function _checkMarketAlignment(address token) internal view returns (bool) {
        address slyDiamondService = SLYWalletStorage.diamondStorage().slyDiamondService;

        try ISLYPriceFeedFacet(slyDiamondService).spfGetBTCAddress() returns (address
btcAddress) {
            // If no BTC reference configured, can't verify alignment
            if (btcAddress == address(0)) return false;

            // If token IS the market reference, alignment is always true
            if (btcAddress == token) return true;

            // Get token's trend
            try ISLYPriceFeedFacet(slyDiamondService).spfGetTrends(token)
            returns (LibSPFStructs.Trend tokenAbsoluteTrend, LibSPFStructs.Trend) {

                // Get BTC's trend
                try ISLYPriceFeedFacet(slyDiamondService).spfGetTrends(btcAddress)
                returns (LibSPFStructs.Trend btcAbsoluteTrend, LibSPFStructs.Trend) {

                    // Compare token's trend with BTC's trend
                    bool tokenIsBullish = LibSPFStructs.isBullish(tokenAbsoluteTrend);
                    bool btcIsBullish = LibSPFStructs.isBullish(btcAbsoluteTrend);

                    return tokenIsBullish == btcIsBullish;

                } catch {
                    return false; // Can't get BTC trend, can't verify alignment
                }
            } catch {
                return false; // Can't get token trend, can't verify alignment
            }
        } catch {
            return false; // Can't get service, can't verify alignment
        }
    }
```

Recommendation:

Step 1: Extend LibSPFStructs.Trend enum

```
enum Trend {
      STRONG_DIP, // Strong downward trend
      DIP,         // Moderate downward trend
      NEUTRAL,     // Sideways/no clear trend
      RIP,         // Moderate upward trend
      STRONG_RIP,  // Strong upward trend
      UNKNOWN      // unknown, call failed
  }
```

Step 2: Simplified Implementation

```

function _checkMarketAlignment(address token) internal view returns (bool) {
    address slyDiamondService = SLYWalletStorage.diamondStorage().slyDiamondService;
    if (slyDiamondService == address(0)) return false;

    // Step 1: Get BTC address
    address btcAddress = _getBTCAddress(slyDiamondService);
    if (btcAddress == address(0)) return false;

    // Step 2: If token IS BTC, alignment is always true
    if (btcAddress == token) return true;

    // Step 3: Get both trends
    LibSPFStructs.Trend tokenTrend = _getTokenTrend(slyDiamondService, token);
    LibSPFStructs.Trend btcTrend = _getTokenTrend(slyDiamondService, btcAddress);

    // Step 4: Compare trends
    if (tokenTrend == LibSPFStructs.Trend.UNKNOWN || btcTrend ==
LibSPFStructs.Trend.UNKNOWN) {
        return false;
    }

    bool tokenIsBullish = LibSPFStructs.isBullish(tokenTrend);
    bool btcIsBullish = LibSPFStructs.isBullish(btcTrend);

    return tokenIsBullish == btcIsBullish;
}


function _getBTCAddress(address slyDiamondService) internal view returns (address) {
    bytes memory callData = abi.encodeWithSignature("spfGetBTCAddress()");
    (bool success, bytes memory returnData) = slyDiamondService.staticcall(callData);

    if (!success || returnData.length < 32) return address(0);

    return abi.decode(returnData, (address));
}

//
function _getTokenTrend(address slyDiamondService, address token) internal view returns
(LibSPFStructs.Trend) {
    bytes memory callData = abi.encodeWithSignature("spfGetTrends(address)", token);
    (bool success, bytes memory returnData) = slyDiamondService.staticcall(callData);

    if (!success || returnData.length < 64) return LibSPFStructs.Trend.UNKNOWN;

    (LibSPFStructs.Trend absoluteTrend,) = abi.decode(returnData, (LibSPFStructs.Trend,
LibSPFStructs.Trend));
    return absoluteTrend;
}
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

[FP-30] `sdrGetReadyPositions` function can be optimized to reduce gas consumption by avoiding double iteration

Code Improvement Info ✓ Fixed

Issue/Risk: Code Improvement

Description:

The current implementation of `sdrGetReadyPositions` performs two separate loops - one to count ready positions and another to populate the result array. This double iteration is inefficient and consumes unnecessary gas, especially when dealing with large position ranges.

Recommendation:

```
function sdrGetReadyPositions(uint256 _startIdx, uint256 _count) external view override
initialized returns (uint256[] memory readyPositions) {
       SLYWalletDRIPStorage.Data storage ds = SLYWalletDRIPStorage.diamondStorage();

        uint256 endIdx = _startIdx + _count;
        if (endIdx > ds.positions.length) endIdx = ds.positions.length;
        // Initialize with max possible size
        readyPositions = new uint256[](endIdx - _startIdx);
        // Count ready positions
        uint256 readyCount = 0;
        for (uint256 i = _startIdx; i < endIdx; i++) {
            if (_isPositionReady(i)) {
                readyPositions[readyCount] = i;
                readyCount++;
            }
        }

        // Adjust array length to actual count of ready positions using assembly
        assembly {
            mstore(readyPositions, readyCount)
        }
    }
```

Update/Status:

The issue is fixed at a37452f57c47934c60d856e96590694c85b6d591.

10. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

- N/A

11. Appendices

- N/A

11.2 External Functions Check Points

- N/A

FAIRYPROOF

https://medium.com/@FairyproofT

https://twitter.com/FairyproofT

https://www.linkedin.com/company/fairyproof-tech

https://t.me/Fairyproof_tech

Reddit: https://www.reddit.com/user/FairyproofTech