Guild Audits

MAY 31ST,
2025

SIR PROTOCOL INITIAL AUDIT REPORT

By Guild Audits

TABLE OF CONTENTS

DISCLAIMER

The Guild Audit team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

EXECUTIVE SUMMARY

This section will represent the summary of the whole audit

Commit: a25347f4a5c176f70f2bbbd4e90af00f2d48592a in master branch of https://github.com/SIR-trading/Core

FINDINGS

Index	Title	Severity	Status of Issues
01	[M-01] Signature Malleability in is possible in the permit Function	Medium	Open
02	[L-01] Type Mismatch in OracleInitialized and UniswapOracleProbed Events	Low	Open
03	[L-02] Missing Input Validation In The stake Fucntion	Low	Open
04	[L-03] Incorrect Tax Constraint Enforcement in StartLiquidityMining and ChangeLiquidityMining	Low	Open
05	[L-04] Undocumented Fee on TEA Token Minting	Low	Open
06	[L-05] Discrepancy in Saturation Price Calculation	Low	Open
07	[L-06] Inconsistent Staker Fee Calculation and Documentation in feeAPE	Low	Open
08	[L-07] Unhandled Auction Winner Payment Failures May Result in Fund Loss	Low	Open
09	**[I-01] Missing Error Message in onlyVault Modifier Require Statement**	Informational	Open
10	**[I-02] No Zero-Address Validation in transfer and transferFrom Functions**	Informational	Open
11	**[I-03] Missing Stake-Specific Events in stake and unstake Functions**	Informational	Open
12	**[I-04] Unused OracleAlreadyInitialized Error in Oracle Contract**	Informational	Open
13	**[I-05] Missing Zero-Value Check IN the Claim function**	Informational	Open
14	**[I-06] Missing Reserve Initialization in Vault.sol**	Informational	Open

MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follows:
Manual Review: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
Automated Testing: This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools used are Slither, Echidna and others.
Functional Testing: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to exploit the contracts.

The methodologies for establishing severity issues:

- High Level Severity Issues
- Medium Level Severity Issues
- Low Level Severity Issues
- Informational Level Severity Issues

TYPES OF SEVERITY

Every issue in this report has been assigned a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

These are critical issues that can lead to a significant loss of funds, compromise of the contract's integrity, or core function of the contract not working. Exploitation of such vulnerabilities could result in immediate and catastrophic consequences, such as:

- Complete depletion of funds.
- Permanent denial of service (DoS) to contract functionality.
- Unauthorized access or control over the contract

Medium Severity Issues

These issues pose a significant risk but require certain preconditions or complex setups to exploit. They may not result in immediate financial loss but can degrade contract functionality or pave the way for further exploitation. Exploitation of such vulnerabilities could result in partial denial of service for certain users,leakage of sensitive data or unintended contract behavior under specific circumstances.

Low Severity Issues

These are minor issues that have a negligible impact on the contract or its users. They may affect efficiency, readability, or clarity but do not compromise security or lead to financial loss. Impacts are minor degradation in performance,confusion for developers or users interacting with the contract and low risk of exploitation with limited consequences.

Informational

These are not vulnerabilities in the strict sense but observations or suggestions for improving the contract's code quality, readability, and adherence to best practices.

There is no direct impact on the contract’s functionality or security, it is aimed at improving code standards and developer understanding.

TYPES OF ISSUES

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the audit and have been successfully fixed.

Acknowledged

Vulnerabilities that have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

REPORT OF FINDINGS

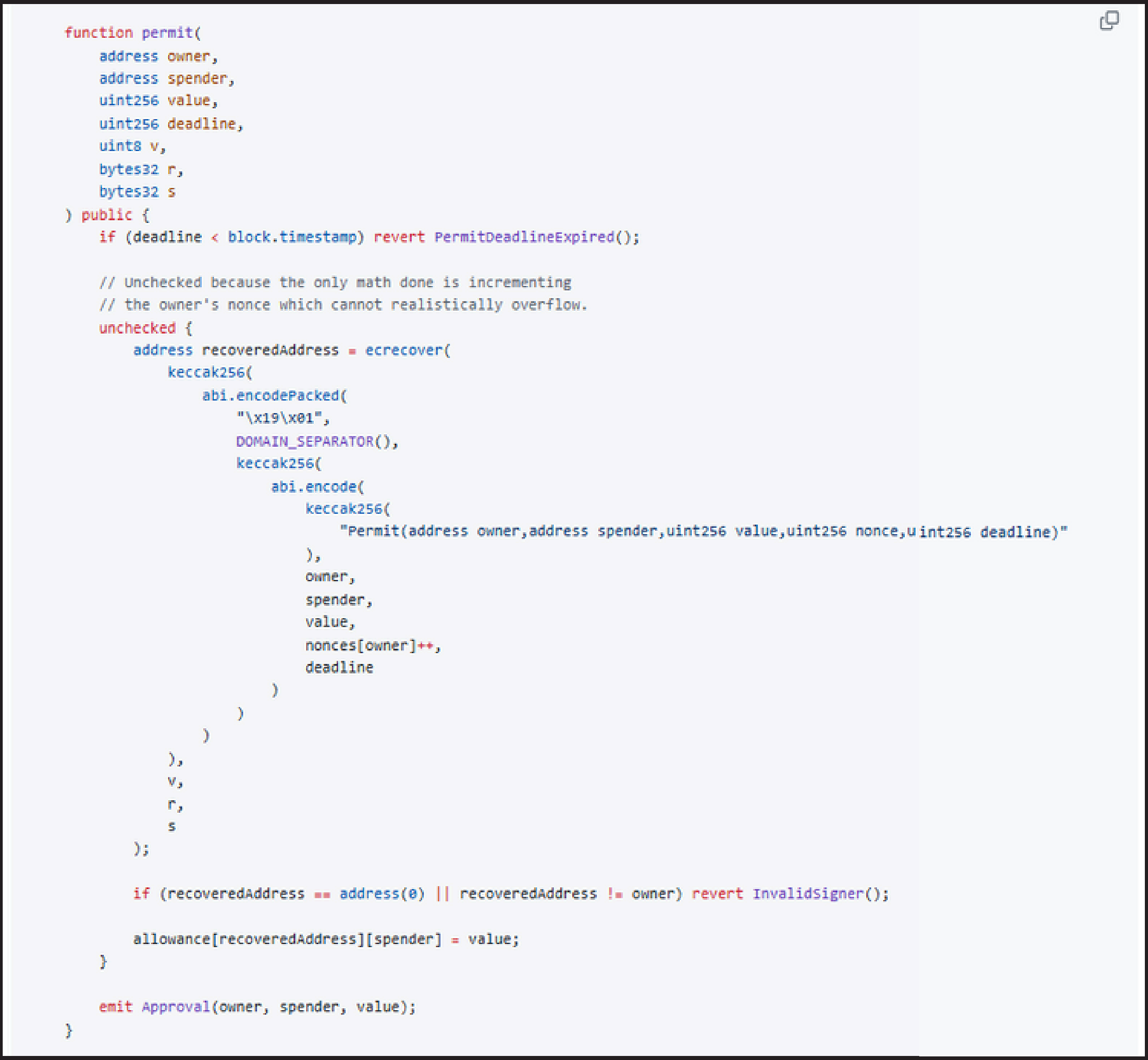
MEDIUM SEVERITY ISSUE

[M-01] Signature Malleability in is possible in the permit Function

- **Description**

The permit function in the Staker.sol contract does not restrict the ECDSA signature's s value to the lower half of the secp256k1 curve's order (s <= N/2, where N is *0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141).*

This enables signature malleability, allowing a valid signature (r, s, v) to be transformed into another valid signature (r, N - s, v') without the private key. An attacker can exploit this by observing a user's permit transaction in the mempool, computing the malleable signature, and submitting it with higher gas fees. If the attacker's transaction is mined first, it consumes the user's nonce (nonces[owner]), incrementing it from n to n+1. The user's transaction then fails as the nonce in their signature (n) no longer matches nonces[owner] (n+1), reverting with *InvalidSigner().*



```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public {
    if (deadline < block.timestamp) revert PermitDeadlineExpired();

    // Unchecked because the only math done is incrementing
    // the owner's nonce which cannot realistically overflow.
    unchecked {
        address recoveredAddress = ecrecover(
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(
                            keccak256(
                                "Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
                            ),
                            owner,
                            spender,
                            value,
                            nonces[owner]++,
                            deadline
                        )
                    )
                )
            ),
            v,
            r,
            s
        );

        if (recoveredAddress == address(0) || recoveredAddress != owner) revert InvalidSigner();

        allowance[recoveredAddress][spender] = value;
    }

    emit Approval(owner, spender, value);
}
```

The lack of a check on the s value allows an attacker to submit a malleable signature, consuming the nonce and causing the user's transaction to revert. This disrupts user transactions and potentially breaking downstream logic dependent on successful permit calls.

Recommendations

Enforce s <= N/2 by adding the following check before the ecrecover call:

if (uint256(s) >
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0)
revert InvalidSigner();

Status: Open

LOW SEVERITY ISSUE

[L-01] Type Mismatch in OracleInitialized and UniswapOracleProbed Events

Summary

A type mismatch exists between the interface (IOracle) and implementation (Oracle) for the avLiquidity parameter in the OracleInitialized and UniswapOracleProbed events. The interface defines avLiquidity as uint160, while the implementation uses uint136.

Vulnerabiity Details

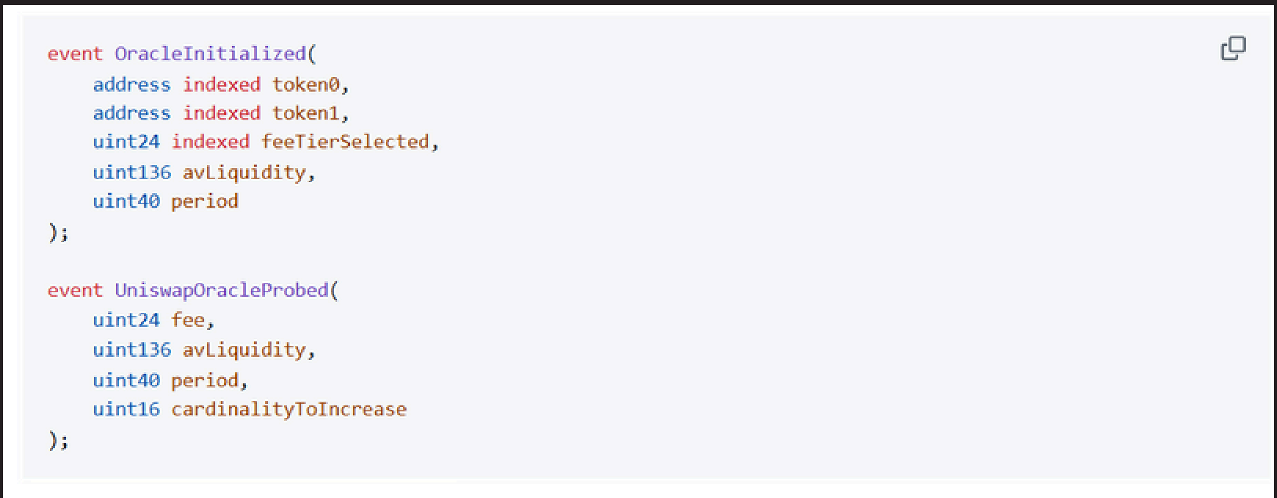
- In the IOracle interface:
  - OracleInitialized event: avLiquidity is uint160.
  - UniswapOracleProbed event: avLiquidity is uint160.
- In the Oracle contract:
  - OracleInitialized event: avLiquidity is uint136.
  - UniswapOracleProbed event: avLiquidity is uint136.
- The mismatch occurs because the implementation uses uint136 for avLiquidity (aligned with the UniswapOracleData struct), while the interface expects uint160.

Impact

- ABI encoding/decoding inconsistencies may occur when interacting with the contract via the interface.
- Off-chain applications parsing events may encounter errors or misinterpret data, expecting uint160 instead of uint136.
- Potential integration issues with tools or contracts relying on the interface's event signature.

Recommended Mitigation

Update the IOracle interface to use uint136 for avLiquidity in both events to match the implementation:



```
event OracleInitialized(
    address indexed token0,
    address indexed token1,
    uint24 indexed feeTierSelected,
    uint136 avLiquidity,
    uint40 period
);

event UniswapOracleProbed(
    uint24 fee,
    uint136 avLiquidity,
    uint40 period,
    uint16 cardinalityToIncrease
);
```

Status: Open

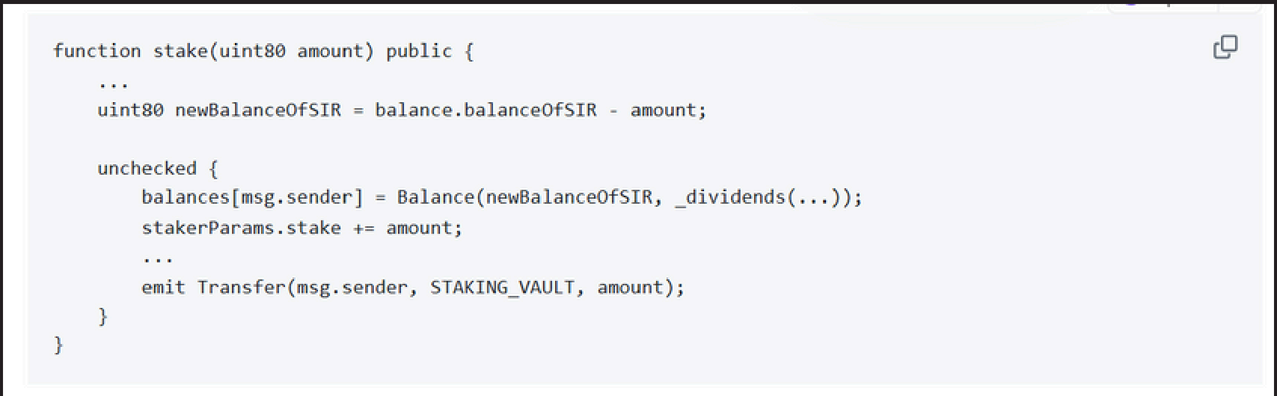
LOW SEVERITY ISSUE

[L-02] Missing Input Validation In The stake Fucntion

Summary

The stake(uint80 amount) function does not validate whether the input amount is greater than zero. As a result, it is possible for users to call the function with a zero value, leading to unnecessary state reads and writes, gas consumption, and emission of meaningless events.

Vulnerabiity Details



```
function stake(uint80 amount) public {
    ...
    uint80 newBalanceOfSIR = balance.balanceOfSIR - amount;

    unchecked {
        balances[msg.sender] = Balance(newBalanceOfSIR, _dividends(...));
        stakerParams.stake += amount;
        ...
        emit Transfer(msg.sender, STAKING_VAULT, amount);
    }
}
```

When amount == 0, the function performs no meaningful state updates but still emits a Transfer event and consumes gas.

Impact

- Users (or bots) can invoke the function with zero-value stakes, causing unnecessary computation and state transitions.
- Emitting Transfer events with a zero amount can pollute the on-chain event history, making it harder to audit and increasing log indexing bloat.
- Repeated zero-value transactions serve no useful purpose and may open the door to low-cost denial-of-service spam under certain conditions.

Recommended Mitigation

Add a validation check to ensure that the staked amount is strictly greater than zero:

require(amount > 0, "Cannot stake zero amount");

This prevents no-op transactions, improves clarity, and reduces unnecessary state changes and event emissions.

Status: Open

LOW SEVERITY ISSUE

[L-03] Incorrect Tax Constraint Enforcement in StartLiquidityMining and ChangeLiquidityMining

Summary

The stake(uint80 amount) function does not validate whether the input amount is greater than zero. As a result, it is possible for users to call the function with a zero value, leading to unnecessary state reads and writes, gas consumption, and emission of meaningless events.

Vulnerabiity Details

SirStructs library specifies that vault taxes must satisfy *Σ_i (tax_i / type(uint8).max)^2 ≤ 0.1^2.* The StartLiquidityMining script sets *newTaxes[0] = 228, newTaxes[1] = 114,* yielding *(228/255)^2 + (114/255)^2 ≈ 0.998 < 0.01*.

However, neither the script nor the updateVaultsIssuances function in ISystemControl programmatically validates this constraint. If the implementation fails to enforce it, adding more vaults or higher taxes could violate the constraint, causing reverts or incorrect reward distributions.

Impact

- Violation of the tax constraint could halt vault updates or misallocate SIR rewards, affecting protocol fairness.

Recommended Mitigation

Add a check in updateVaultsIssuances to enforce the tax constraint. Update scripts to validate taxes before submission.

Status: Open

LOW SEVERITY ISSUE

[L-04] Undocumented Fee on TEA Token Minting

Summary

The protocol documentation, specifically the section detailing the fee structure, focuses on fees generated from the minting and burning of APE tokens, which reward Liquidity Providers ("Gentlemen"). However, this documentation omits that a fee is also levied on Liquidity Providers themselves when they mint TEA tokens (i.e., when providing liquidity). The smart contracts implement this TEA minting fee, with the collected portion contributing to Protocol Owned Liquidity (POL). This discrepancy can lead to a misunderstanding of the complete fee mechanics for Liquidity Providers.

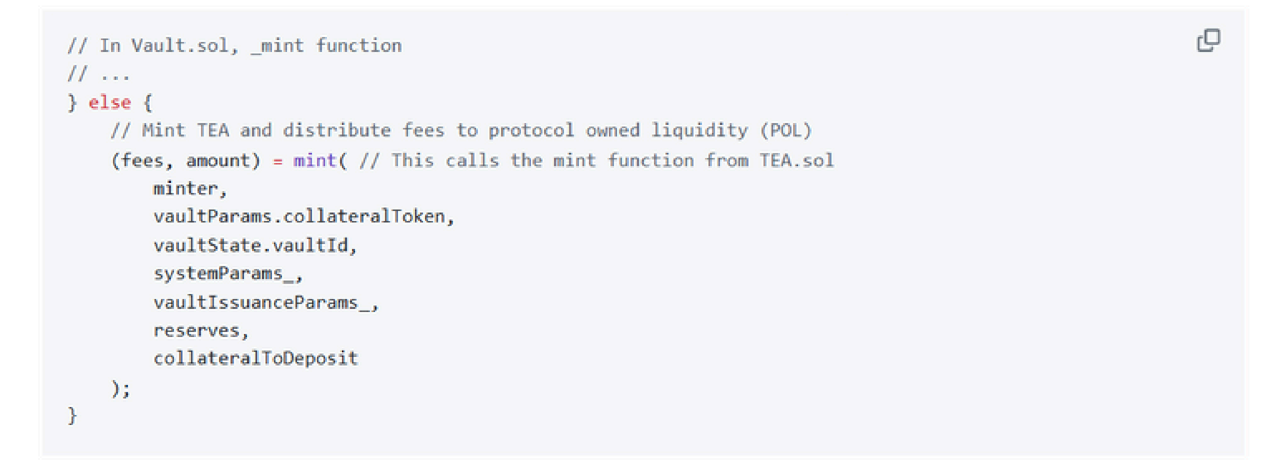
Description of Issue

The provided documentation snippet concerning fees states

Vaults feature a fee system that rewards the gentlemen with significant fees from the minting and burning of APE tokens. These fees vary by vault, increasing with the vault's leverage ratio. Although these fees are substantial, they allow apes to hold APE tokens without incurring any maintenance fees, regardless of the holding period. The fees for minting or burning APE tokens are on par with the costs of holding a margin position for approximately one year, striking a balance between potential returns and upfront costs. This structure aims to benefit liquidity providers and encourage long-term traders, while short-term traders may not see their speculative positions fully materialize, essentially contributing more to the ecosystem's finances through these initial fees.

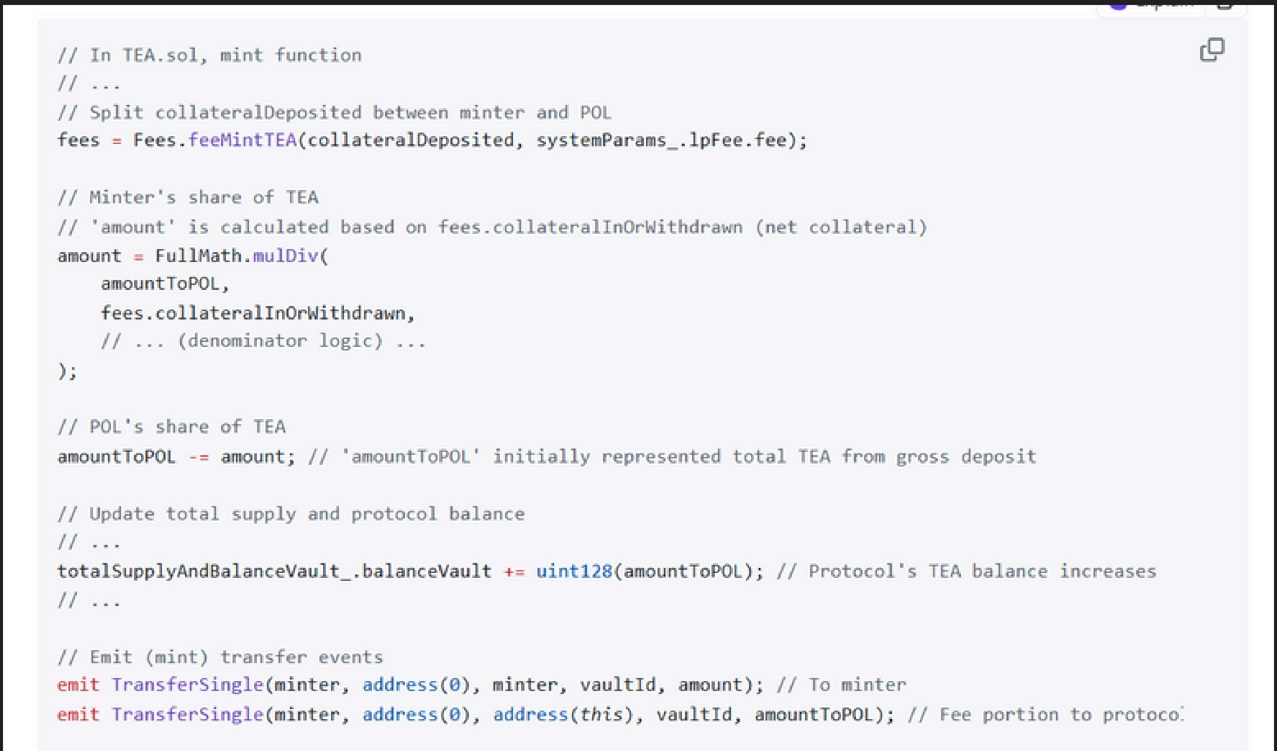
This section exclusively describes fees related to APE token activities and their role in rewarding Liquidity Providers. It does not mention any fees applicable to Liquidity Providers when they mint TEA tokens.
However, an examination of the Vault.sol and TEA.sol contracts reveals the implementation of such a fee:

- Vault.sol - _mint function: When a user mints TEA tokens (isAPE == false), the _mint function calls the mint function inherited from TEA.sol. A comment within this block explicitly notes the distribution of fees to Protocol Owned Liquidity:



```
// In Vault.sol, _mint function
// ...
} else {
    // Mint TEA and distribute fees to protocol owned liquidity (POL)
    (fees, amount) = mint( // This calls the mint function from TEA.sol
        minter,
        vaultParams.collateralToken,
        vaultState.vaultId,
        systemParams_,
        vaultIssuanceParams_,
        reserves,
        collateralToDeposit
    );
}
```

TEA.sol - mint function: This function, responsible for minting TEA tokens, clearly calculates and applies a fee based on systemParams_.lpFee.fee. The portion of TEA tokens corresponding to this fee is then minted to the protocol itself (address(this)), thereby increasing POL.



```
// In TEA.sol, mint function
// ...
// Split collateralDeposited between minter and POL
fees = Fees.feeMintTEA(collateralDeposited, systemParams_.lpFee.fee);

// Minter's share of TEA
// 'amount' is calculated based on fees.collateralInOrWithdrawn (net collateral)
amount = FullMath.mulDiv(
    amountToPOL,
    fees.collateralInOrWithdrawn,
    // ... (denominator logic) ...
);

// POL's share of TEA
amountToPOL -= amount; // 'amountToPOL' initially represented total TEA from gross deposit

// Update total supply and protocol balance
// ...
totalSupplyAndBalanceVault_.balanceVault += uint128(amountToPOL); // Protocol's TEA balance increases
// ...

// Emit (mint) transfer events
emit TransferSingle(minter, address(0), minter, vaultId, amount); // To minter
emit TransferSingle(minter, address(0), address(this), vaultId, amountToPOL); // Fee portion to protoco
```

This implemented fee on TEA minting is not reflected in the user-facing documentation regarding the protocol's fee structure.

Impact

LPs might not be aware that a portion of their deposited collateral is effectively taken as a fee when minting TEA tokens, as the documentation focuses on APE token fees as their reward source. They might expect the amount of TEA tokens received to be directly proportional to their full collateral deposit.

Recommendation

To ensure full transparency and align documentation with the on-chain behavior, it is recommended to update the protocol's fee documentation. The updated documentation should clearly

1. State that a fee is applied when Liquidity Providers (Gentlemen) mint TEA tokens.
2. Explain the basis for this fee calculation (e.g., derived from systemParams_.lpFee.fee).
3. Describe the purpose and destination of this fee, specifically its contribution to Protocol Owned Liquidity (POL), and briefly explain the benefits of POL to the ecosystem.

Status: Open

LOW SEVERITY ISSUE

[L-05] Discrepancy in Saturation Price Calculation

Summary

The _updateVaultState function calculates and stores a compressed representation of a vault's state, including tickPriceSatX42, which defines the boundary between the "Power Zone" (ideal constant leverage) and the "Saturation Zone" (liquidity-constrained). Within the logic for the Saturation Zone, there is a significant discrepancy between the mathematical formula for the saturation price (priceSat) implied by the implemented code and the formula stated in the accompanying code comment. This can lead to the vault operating with an incorrect saturation price threshold, potentially affecting P&L calculations and the transition between operational zones.

Description of Issue

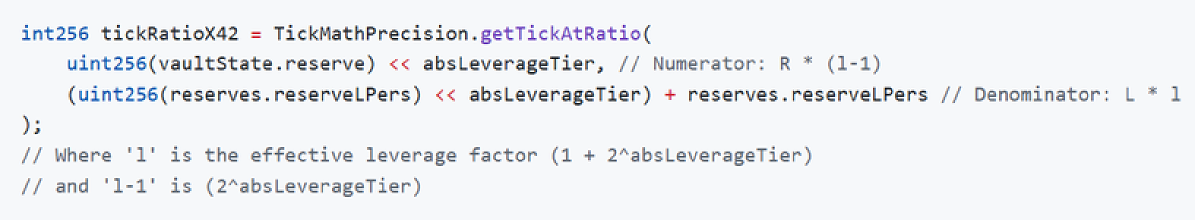
The issue lies in the calculation of tickPriceSatX42 when isPowerZone is false (i.e., the vault is determined to be in the Saturation Zone).

Commented Intention: The code comment for the Saturation Zone states the target formula as:

```
/*
    PRICE IN SATURATION ZONE
    priceSat = r*price*L/R
 */
```

Assuming price is the current price (priceCurrent), this implies *priceSat / priceCurrent = (r * L) / R*. In tick space, this would translate to: *tickSat - tickCurrent = tick((r * L) / R),* then *tickSat = tickCurrent + tick((r * L) / R)*

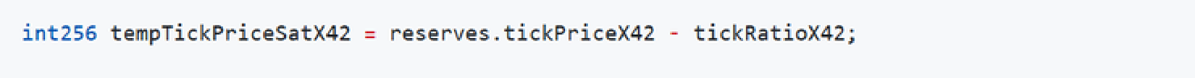
Implemented Logic (for positive leverageTier): The code calculates tickRatioX42 as:



```
int256 tickRatioX42 = TickMathPrecision.getTickAtRatio(
    uint256(vaultState.reserve) << absLeverageTier, // Numerator: R * (l-1)
    (uint256(reserves.reserveLPers) << absLeverageTier) + reserves.reserveLPers // Denominator: L * l
);
// Where 'l' is the effective leverage factor (1 + 2^absLeverageTier)
// and 'l-1' is (2^absLeverageTier)
```

So, *tickRatioX42 = tick((R * (l-1)) / (L * l)).*

Then, tickPriceSatX42 is computed as:



```
int256 tempTickPriceSatX42 = reserves.tickPriceX42 - tickRatioX42;
```

This means *tickSat = tickCurrent - tick((R * (l-1)) / (L * l)).*

Converting the implemented logic back to price terms: priceSat / priceCurrent = 1 / ((R * (l-1)) / (L * l))

priceSat / priceCurrent = (L * l) / (R * (l-1))

So, priceSat = priceCurrent * (L * l) / (R * (l-1)).
The implemented formula priceSat = priceCurrent * (L * l) / (R * (l-1)) does not match the commented formula priceSat = r * price * L / R. For the two to be equivalent, r would need to be equal to l / (l-1). If r is intended to be simply l (the leverage factor), or another distinct system parameter, the implementation is incorrect relative to the comment.

Impact Explanation

If the commented formula (priceSat = r*price*L/R) represents the true intended mathematical model for the saturation price in this zone, then the current implementation is incorrect. This would lead to:

- Incorrect tickPriceSatX42 Storage: The on-chain tickPriceSatX42 will not accurately reflect the intended saturation threshold.
- The point at which the vault's behavior (and thus P&L calculations for LPers and Apes) transitions from the Power Zone to the Saturation Zone (and vice-versa, as determined by comparing the current market price tick with tickPriceSatX42 in the VaultExternal.getReserves function) will be based on this potentially incorrect value.
- Depending on how tickPriceSatX42 influences the distribution of value between LPers and Apes (especially how reserveApes and reserveLPers are calculated in VaultExternal._getReserves based on this stored tickPriceSatX42), an incorrect saturation threshold could lead to unfair or unintended economic outcomes for participants. For example, it might cause the system to enter or exit the "fixed DBT value for LPs" mode at the wrong price points.

Recommendation

1. If the formula in the comment** (priceSat = r*price*L/R) is correct, the Solidity implementation for calculating tickRatioX42 and its subsequent application (addition or subtraction, and the ratio itself) must be revised to accurately reflect this formula.
2. If the current code's derived formula** (priceSat = priceCurrent * (L*l) / (R*(l-1))) is correct and intended, then the comment must be updated to accurately describe the implemented logic. The definition and role of r (if it's different from l/(l-1)) would also need clarification.

Status: **Open**

LOW SEVERITY ISSUE

[L-06] Inconsistent Staker Fee Calculation and Documentation in feeAPE

Summary

The feeAPE function in the Fees.sol library calculates fees paid by users ("Apes") when minting or burning APE tokens. A portion of this fee is intended for SIR stakers, determined by a tax parameter. However, there is a significant discrepancy between the NatSpec documentation, an inline code comment, and the actual implemented formula regarding the percentage of the total fee allocated to stakers. This inconsistency can lead to misconfiguration, incorrect expectations about fee distribution, and reduced clarity of the protocol's economic model.

Description of Issue

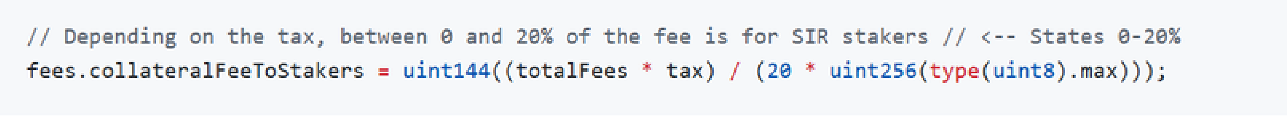
Within the feeAPE function in Fees.sol, the calculation for fees.collateralFeeToStakers exhibits conflicting information:

NatSpec Documentation:

```
/** @notice APES pay a fee to the LPers when they mint/burn APE
   @notice If a non-zero tax is set for the vault, 10% of the fee is sent to SIR stakers // <-- States 10%
   @param tax Tax in basis points charged to the apes for getting SIR // <-- Describes tax as basis points, but it's uint8
 */
```

The NatSpec suggests that if tax is non-zero, a fixed 10% of the total APE fee goes to stakers. It also describes tax as "Tax in basis points," but the parameter type is uint8.

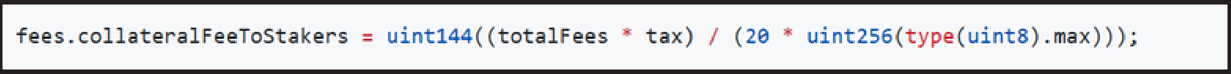
Inline Code Comment:



```
// Depending on the tax, between 0 and 20% of the fee is for SIR stakers // <-- States 0-20%
fees.collateralFeeToStakers = uint144((totalFees * tax) / (20 * uint256(type(uint8).max)));
```

An inline comment directly above the calculation suggests the staker's fee portion can range **between 0 and 20%**.

Actual Implemented Formula:



```
fees.collateralFeeToStakers = uint144((totalFees * tax) / (20 * uint256(type(uint8).max)));
```

Let's analyze this formula:

- tax is a uint8, so type(uint8).max is 255.
- The denominator is 20 * 255 = 5100.
- **The formula is effectively:** fees.collateralFeeToStakers = (totalFees * tax) / 5100.
- If tax = 0, then collateralFeeToStakers = 0.
- If tax = 255 (the maximum value for a uint8), then: collateralFeeToStakers = (totalFees * 255) / (20 * 255) = totalFees / 20 = 0.05 * totalFees.
- This means the implemented formula results in the stakers receiving **between 0% and 5%** of the totalFees, scaled linearly by the tax parameter (0-255).

There is a clear three-way mismatch:

- NatSpec implies a fixed 10% (if tax is active).
- Inline comment suggests a 0-20% range.
- The code implements a 0-5% range, scaled by the uint8 tax parameter. Additionally, the NatSpec description of tax as "basis points" conflicts with its uint8 type and how it's used in the 0-5% scaling formula.

Impact Explanation

This inconsistency has several negative impacts:

- Users, developers, and auditors relying on the NatSpec or inline comments will have an incorrect understanding of the actual fee distribution to stakers.
- The true percentage of APE fees allocated to stakers is a key economic lever for the protocol. The conflicting information obscures the intended design and actual outcome.
- If governance or administrators set the tax parameter for vaults based on the assumption that it corresponds to a 10% rate or a 0-20% range, the actual staker revenue will be significantly different (0-5%), leading to outcomes that don't match intent.

Recommendation

- If the current 0-5% scaled by uint8 tax is the intended logic, then both the NatSpec and the inline comment are incorrect and must be updated. The NatSpec description of tax as "basis points" also needs correction.
- If, for example, a fixed 10% (when tax > 0) is intended, the formula should be changed to fees.collateralFeeToStakers = tax > 0 ? uint144((totalFees * 10) / 100) : 0;.
- If a 0-20% range scaled by uint8 tax is intended, the formula should be fees.collateralFeeToStakers = uint144((totalFees * tax) / (5 * uint256(type(uint8).max)));.

Status: Open

LOW SEVERITY ISSUE

[L-07] Failure to Handle Auction Winner Payment Failure Leads to Loss of Funds

Summary

The collectFeesAndStartAuction function in the Staker.sol contract does not check the success status of the _payAuctionWinner call. If the transfer of the auction lot to the previous winner fails (e.g., due to a problematic token or recipient), the function proceeds as if the payment was successful, causing the previous winner to lose their lot, which is then included in the subsequent auction.

Description of Issue

The Staker.sol contract manages token auctions where users bid WETH to win lots of various ERC20 tokens collected as fees. The process involves settling a previous auction and starting a new one via the collectFeesAndStartAuction function.

The internal function _payAuctionWinner(address token, SirStructs.Auction memory auction, address beneficiary) is designed to transfer the auctioned token amount (auction.bid) to the beneficiary (or auction.bidder if beneficiary is address(0)). Crucially, _payAuctionWinner is designed not to revert on transfer failure. Instead, it uses low-level calls and returns a boolean: true on success, false on failure. Failures can occur for various reasons, including:

- The token contract's transfer function reverts.
- The token contract's transfer function returns false (as per some ERC20 implementations).

- The recipient (beneficiary or auction.bidder) is unable to receive the tokens (e.g., a blacklisted address for tokens like USDT, a contract that reverts on receiving tokens, or insufficient gas provided for the internal call to the token's transfer function if the token's logic is complex).

The collectFeesAndStartAuction(address token) function executes the following relevant steps when settling a previous auction for token:

1. It retrieves the details of the previous auction: SirStructs.Auction memory auction = _auctions[token];.
2. It **resets the state for the current token's auction** in storage, effectively erasing the previous winner's details: _auctions[token] = ... { bidder: address(0), bid: 0, ... };.
3. It updates totalWinningBids based on the previous auction's bid.
4. It calls _distributeDividends().
5. It calls _payAuctionWinner(token, auction, address(0)); to pay the previous winner.
6. **It proceeds without checking the boolean return value of _payAuctionWinner.**
7. It then withdraws new fees for the token from the Vault using vault.withdrawFees(token).

The security guarantee broken is the **fair distribution of auction winnings**. If the call to _payAuctionWinner fails (returns false) for any reason other than auction.bid == 0 (meaning there was a winner to pay, but payment failed), the collectFeesAndStartAuction function incorrectly assumes the payment succeeded. Because the previous auction's state (_auctions[token]) has already been reset, the previous winner loses their claim to the lot. The tokens that failed to transfer remain in the Staker.sol contract and are subsequently included in the lot for the new auction when vault.withdrawFees(token) is called. The winner of the new auction receives the combined lot, effectively taking the previous winner's assets.

This is in contrast to the getAuctionLot function, which correctly checks the return value of _payAuctionWinner and reverts if the payment fails, allowing the winner to potentially try claiming again or for the issue to be addressed.

Impact Explanation

This can lead to a direct and unrecoverable loss of funds for the user who won the previous auction. Their rightful winnings are effectively stolen and given to the winner of the next auction for the same token. This breaks a core promise of the auction mechanism – that the winner receives the lot they bid on.

POC

Click here to view the runnable **POC**

This test demonstrates that Alice's winning lot was not transferred to her and was instead added to the lot of the subsequent auction due to the unchecked return value of _payAuctionWinner.

Recommendation

- The collectFeesAndStartAuction function should check the boolean return value of the _payAuctionWinner call. If the previous auction had a bid (auction.bid > 0) and the payment failed (!paymentSuccessful), the function should revert. This prevents the previous winner's lot from being lost and ensures the auction state remains consistent, potentially allowing the previous winner to claim via getAuctionLot later or enabling manual intervention.

Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-01] Missing Error Message in onlyVault Modifier Require Statement

Description of Issue

- The onlyVault modifier in the APE contract uses a require statement to restrict calls to the vault contract, but it lacks an error message:

```
modifier onlyVault() {
    address vault = _getArgAddress(1);
    require(vault == msg.sender);
    _;
}
```

This modifier is applied to the initialize, mint, and burn functions, ensuring only the vault (stored as an immutable argument) can call them. Without an error message, a revert provides no context about the failure.

Recommendations

Add an error message to the require statement to improve user experience.

```
modifier onlyVault() {
    address vault = _getArgAddress(1);
    require(vault == msg.sender, "APE: caller is not the vault");
    _;
}
```

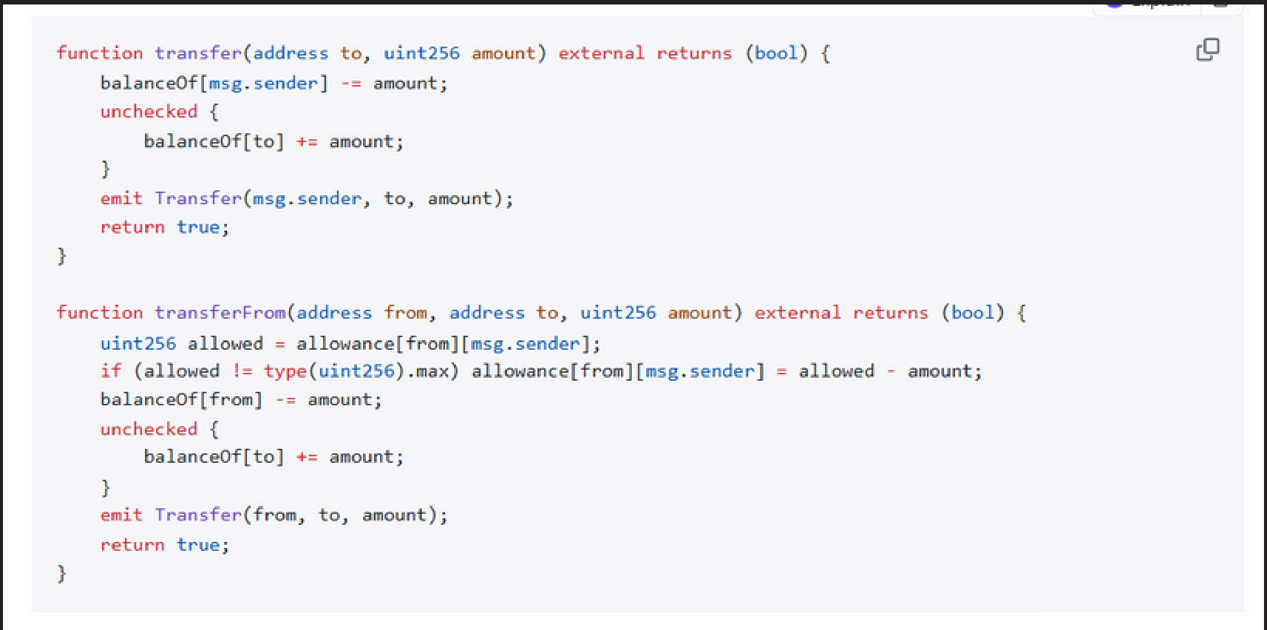
Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-02] No Zero-Address Validation in transfer and transferFrom Functions

Description of Issue

The transfer and transferFrom functions in the APE contract do not validate that the to address is non-zero, allowing tokens to be sent to address(0):



```
function transfer(address to, uint256 amount) external returns (bool) {
    balanceOf[msg.sender] -= amount;
    unchecked {
        balanceOf[to] += amount;
    }
    emit Transfer(msg.sender, to, amount);
    return true;
}

function transferFrom(address from, address to, uint256 amount) external returns (bool) {
    uint256 allowed = allowance[from][msg.sender];
    if (allowed != type(uint256).max) allowance[from][msg.sender] = allowed - amount;
    balanceOf[from] -= amount;
    unchecked {
        balanceOf[to] += amount;
    }
    emit Transfer(from, to, amount);
    return true;
}
```

Transferring tokens to address(0) increases balanceOf[address(0)] without decreasing totalSupply. The totalSupply includes locked tokens, misleading dApps or users expecting it to reflect circulating supply.

Recommendations

Add zero-address validation to both functions to prevent token loss

Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-03] Missing Stake-Specific Events in stake and unstake Functions

Description of Issue

The stake and unstake functions in the Staker contract emit only Transfer events to track staking and unstaking activities, without dedicated events for these actions. This makes it harder for off-chain applications (e.g., indexers, wallets, dashboards) to track staking-specific activities efficiently.

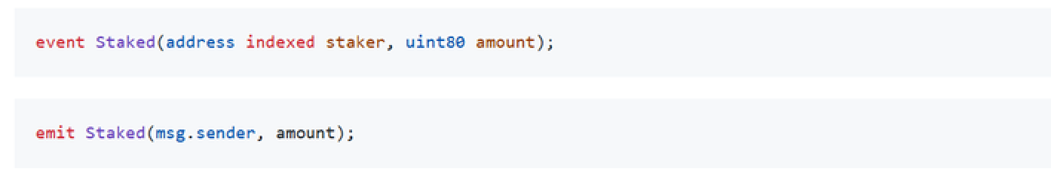
The stake function emits:
emit Transfer(msg.sender, STAKING_VAULT, amount);

The unstake function emits:
emit Transfer(STAKING_VAULT, msg.sender, amount);

While these Transfer events allow tracking via the STAKING_VAULT, they require additional filtering to distinguish staking/unstaking from regular token transfers. This increases complexity for off-chain systems.

Recommendations

Add clear events for staking/unstaking actions:



```
event Staked(address indexed staker, uint80 amount);
```

```
emit Staked(msg.sender, amount);
```

Status: **Open**

INFORMATIONAL SEVERITY ISSUE

[I-04] Unused OracleAlreadyInitialized Error in Oracle Contract

Summary

The OracleAlreadyInitialized error is defined in the Oracle contract and IOracle interface but is not used in the implementation, making it dead code.

Vulnerability Details

- Error declared in Oracle contract and IOracle interface:

error OracleAlreadyInitialized();

- In the initialize function, the contract checks oracleState.initialized but returns early instead of reverting with the error:

if (oracleState.initialized) return;

- No other function uses this error.

Impact

- No functional or security impact. Just dead code

Recommended Mitigation

Remove the unused error from both the Oracle contract and IOracle interface

Status: Open

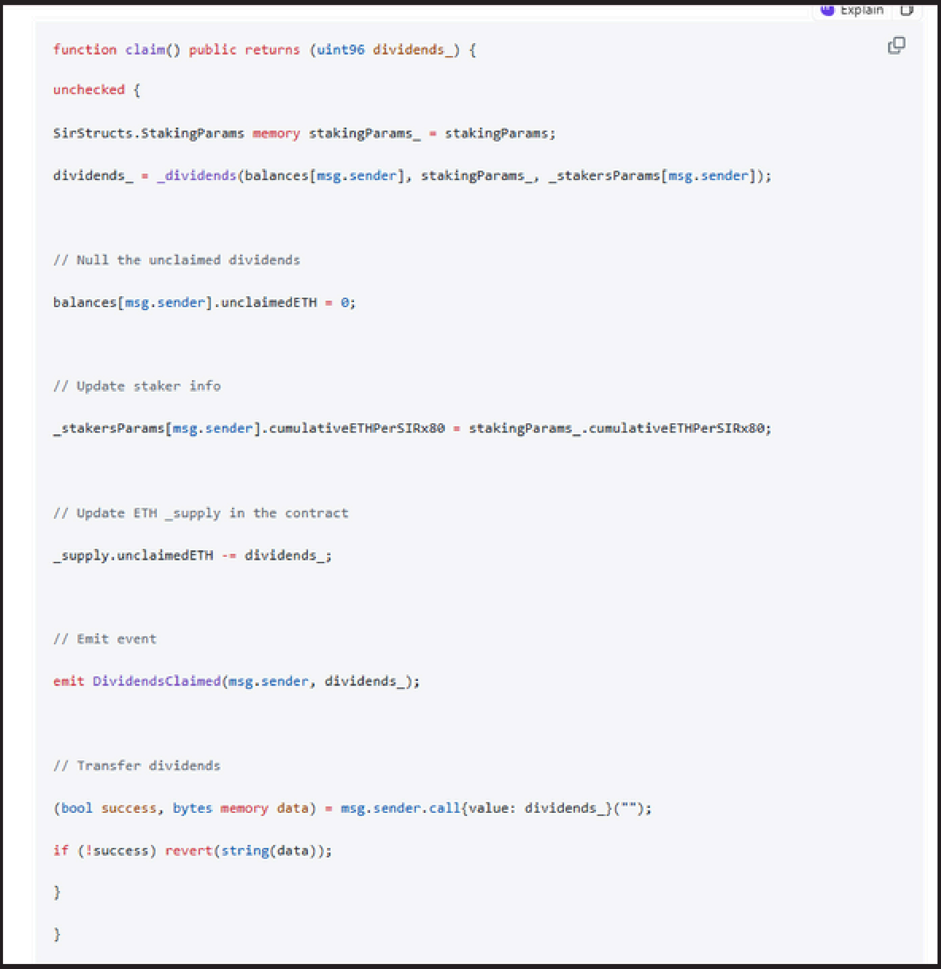
INFORMATIONAL SEVERITY ISSUE

[I-05] Missing Zero-Value Check IN the Claim function

The Claim function does not check if dividends_ > 0 before continuing execution. This may lead to unnecessary state changes and zero-value ETH transfers, wasting gas.

Code Snippet

The function does not check if dividends_ > 0 before continuing execution. This may lead to unnecessary state changes and zero-value ETH transfers, wasting gas.



```
function claim() public returns (uint96 dividends_) {

unchecked {

SirStructs.StakingParams memory stakingParams_ = stakingParams;

dividends_ = _dividends(balances[msg.sender], stakingParams_, _stakersParams[msg.sender]);


// Null the unclaimed dividends

balances[msg.sender].unclaimedETH = 0;


// Update staker info

_stakersParams[msg.sender].cumulativeETHPerSIRx80 = stakingParams_.cumulativeETHPerSIRx80;


// Update ETH _supply in the contract

_supply.unclaimedETH -= dividends_;


// Emit event

emit DividendsClaimed(msg.sender, dividends_);


// Transfer dividends

(bool success, bytes memory data) = msg.sender.call{value: dividends_}("");

if (!success) revert(string(data));

}

}
```

Impact

Gas inefficiency and unnecessary logs/operations.

Recommended Mitigation

Add a check to skip execution if dividends_ is 0.

Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-06] Missing Reserve Initialization in Vault.sol

Description

The Vault.initialize() function in the SIR Protocol fails to explicitly initialize the reserveApes value when creating a new vault. This oversight leaves the reserves.reserveApes field uninitialized and potentially filled with non-zero garbage data from unclean memory. The APE.mint() function assumes that a zero totalSupply implies a safe "first mint" state and uses the passed-in reserves.reserveApes without validation. If this value is unintentionally high due to uninitialized memory, the minting logic will over-mint APE tokens on first deposit, allowing inflation of the token supply far beyond the actual collateral provided.

Impact

This can break the peg or value of APE tokens, and bankrupt vaults.

Recommendation

Explicitly set reserves.reserveApes = 0 and reserves.reserveLPers = 0 during vault initialization.

Status: **Open**