THREE SIGMA

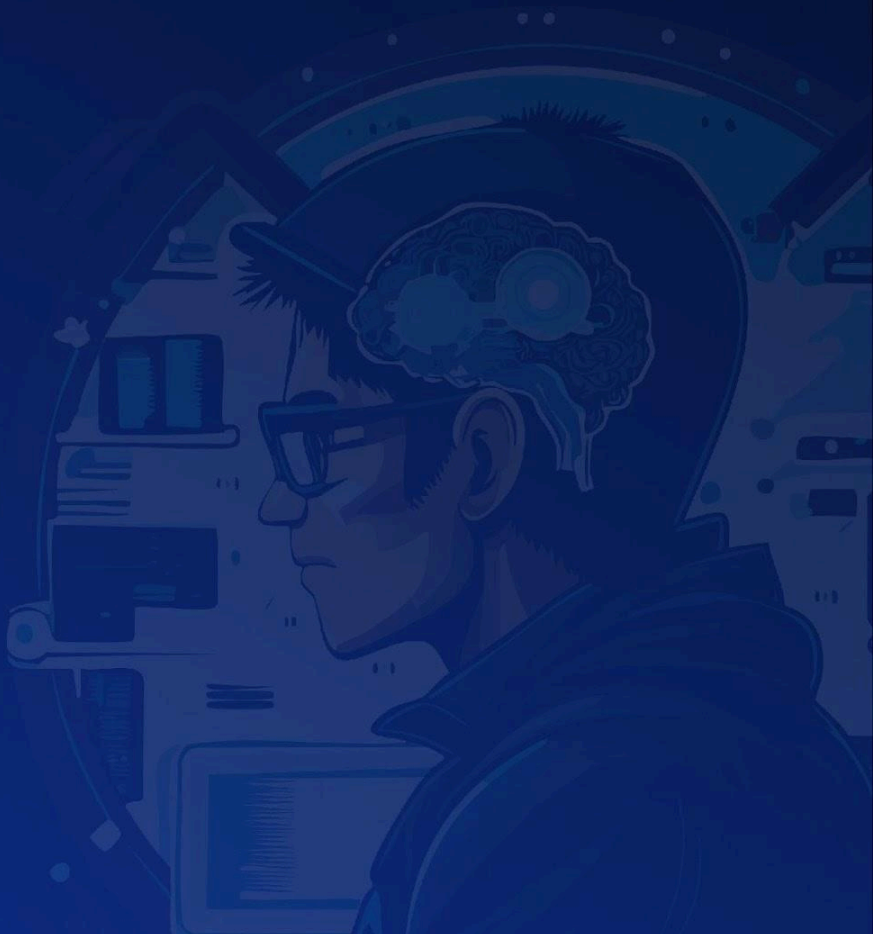
Soneta Borrowing Protocol

Security Review

Disclaimer

Security Review

Soneta Borrowing Protocol



Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

Soneta Borrowing Protocol

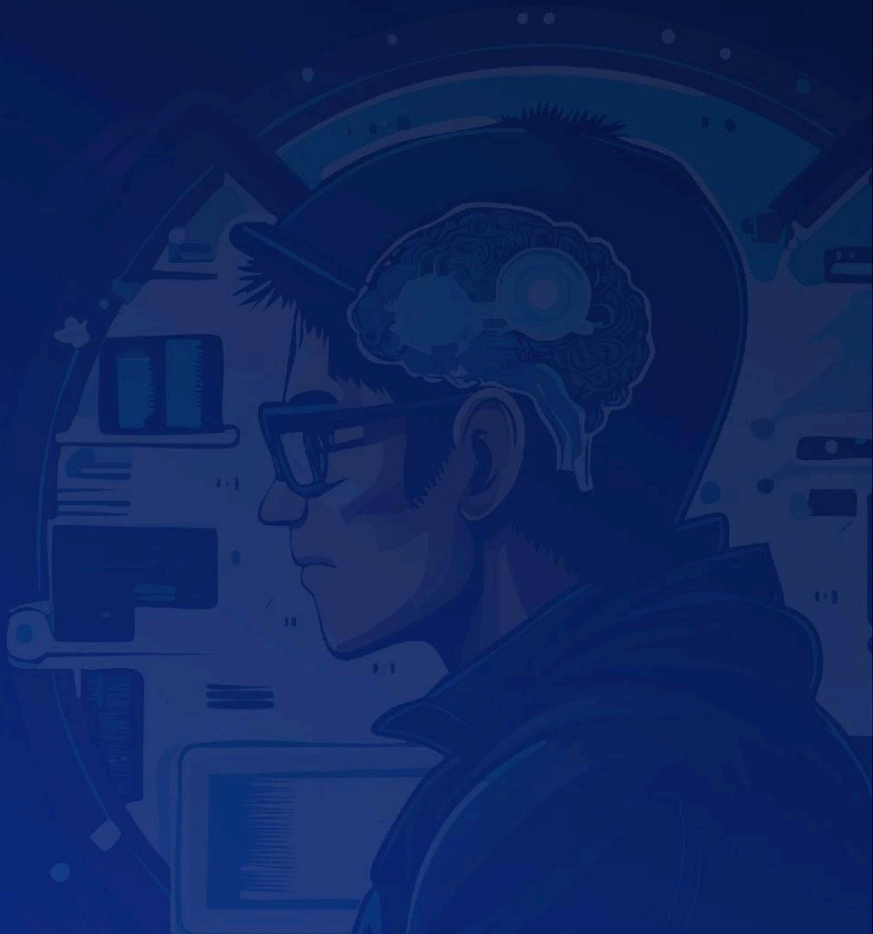
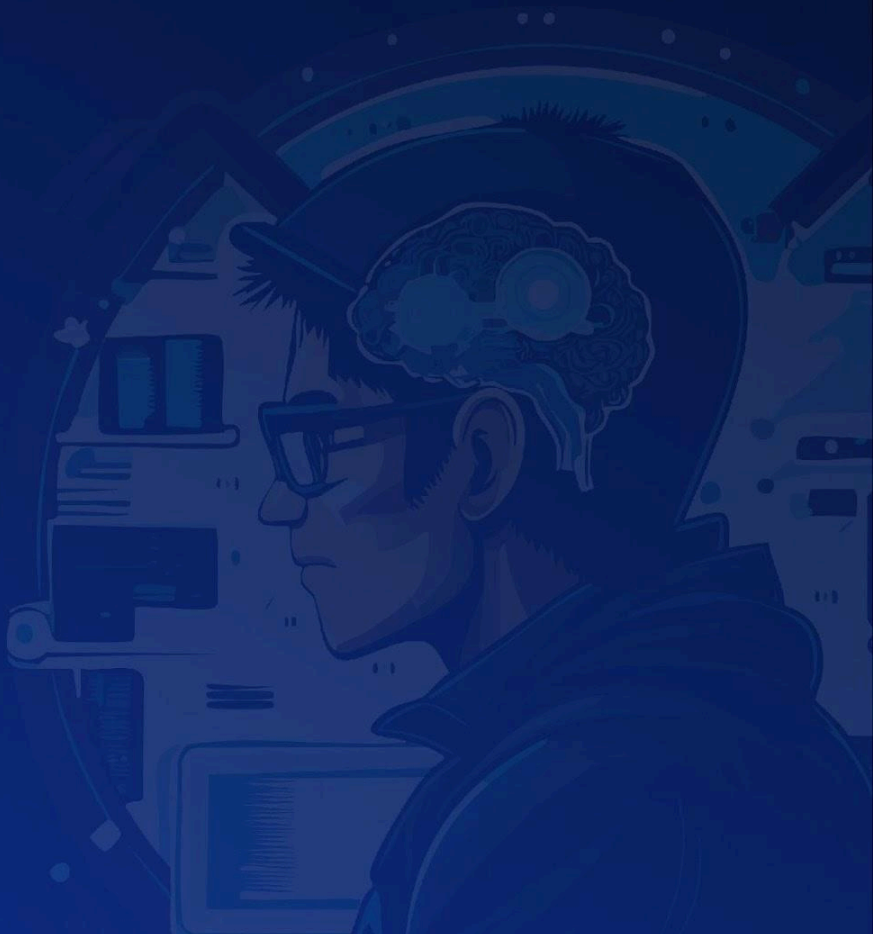


Table of Contents

Summary

Security Review

Soneta Borrowing Protocol



Summary

Three Sigma audited Soneta in a 2.4 person week engagement. The audit was conducted from 19/05/2025 to 22/05/2025.

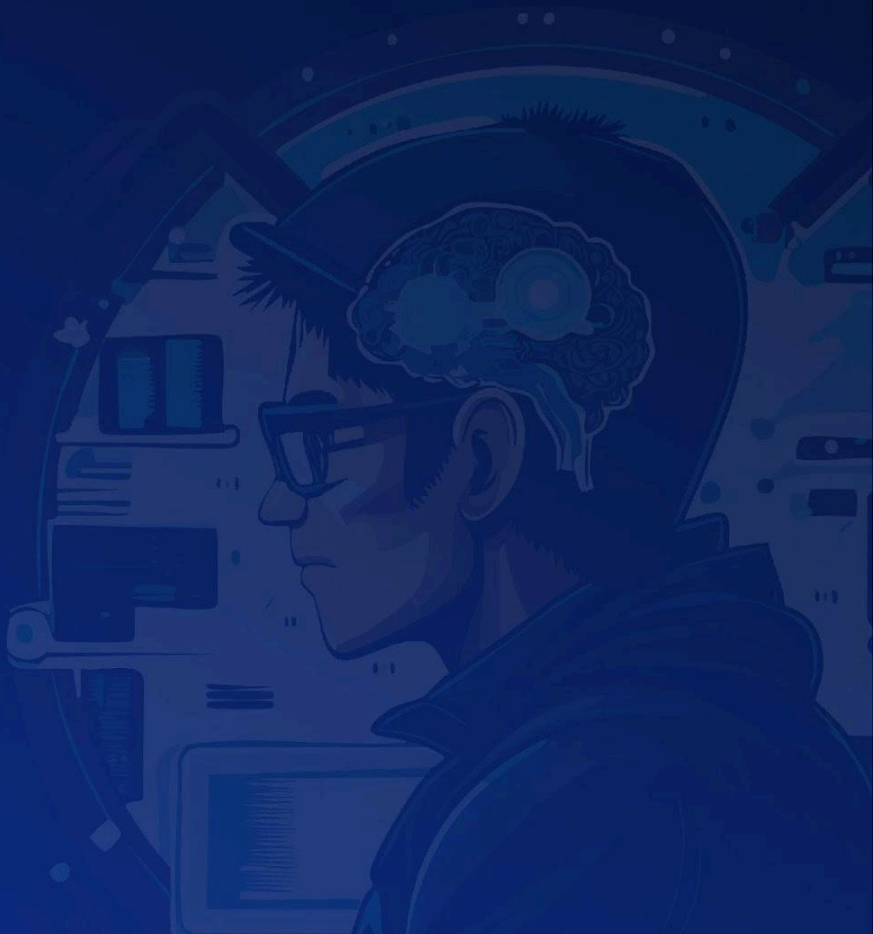
Protocol Description

Soneta is the official fork of Liquity v2 on Sonic. Deposit collateral to mint the protocol's overcollateralized stablecoin, ONE. Customize your interest rate and loan-to-value (LTV) ratio to suit your borrowing needs and risk profile. Stake ONE in the Stability Pool to benefit from a portion of the protocol fees and liquidations. The other portion is allocated via governance to a ONE LP Pool. ONE can be redeemed from the protocol for $1 worth of collateral at any time.

Scope

Security Review

Soneta Borrowing Protocol



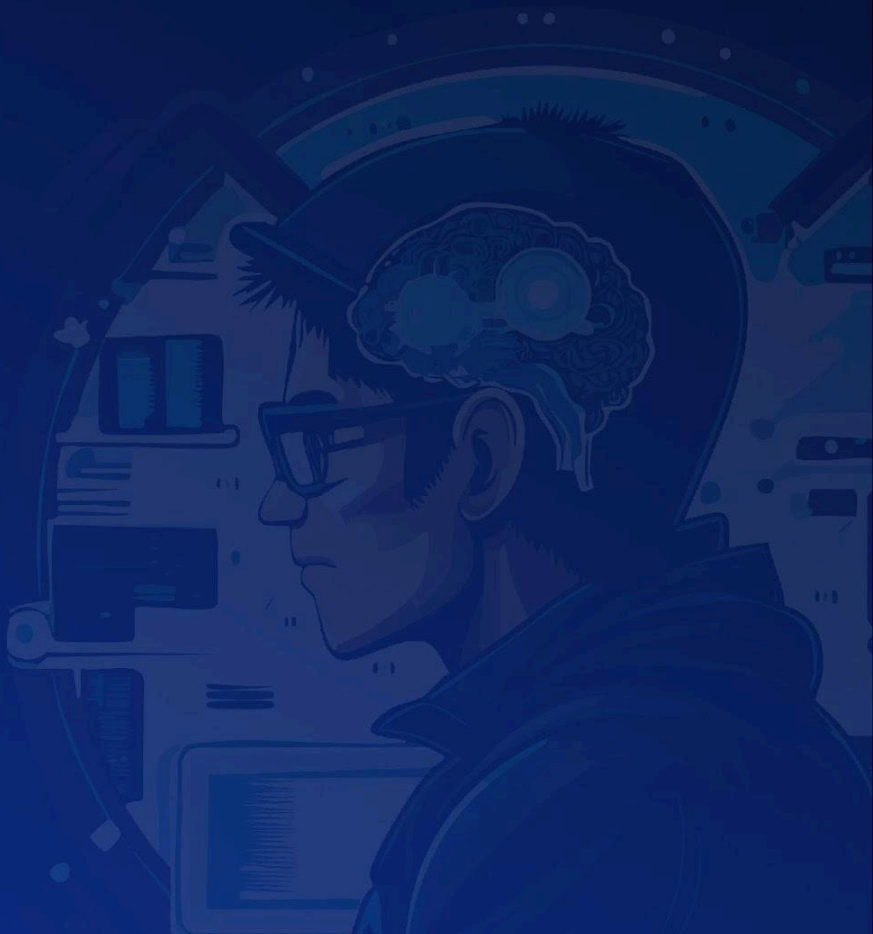
Scope

Filepath	nSLOC
src/Sonata/ChainlinkPriceFeed.sol	390
src/Sonata/ERC20StandardizedWrapper.sol	
src/Sonata/LBTCPriceFeed.sol	
src/Sonata/SingleChainlinkPriceFeed.sol	
src/Sonata/SonataToken.sol	
src/Sonata/WrappedOriginPriceFeed.so	
src/Zappers/Modules/Sonata/HybridCurveShadowExchange.sol	
src/Zappers/Modules/Sonata/HybridCurveShadowExchangeHelpers.sol	
src/Zappers/GasCompZapper.sol	116
src/Zappers/LeftoversSweep.sol	
src/Zappers/LeverageLSTZapper.sol	
Total	**506**

Methodology

Security Review

Soneta Borrowing Protocol



Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

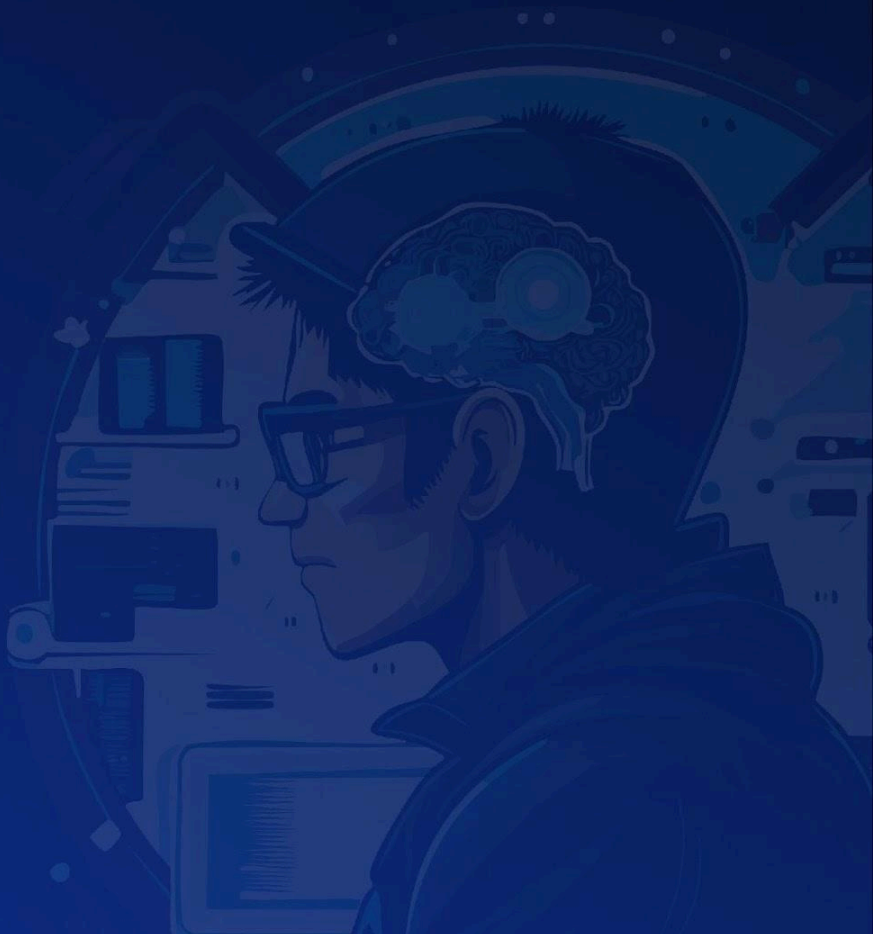
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Soneta Borrowing Protocol



Project Dashboard

Application Summary

Name	Soneta
Repository	https://github.com/SonetaLQTY/sonata
Commit	7c1583e66d3df275b1b6f4edad0d687665ba26d6
Language	Solidity
Platform	Sonic

Engagement Summary

Timeline	19/05/2025 to 22/05/2025
№ of Auditors	3
Review Time	2.4 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	2	2	0
High	5	5	0
Medium	7	7	0
Low	3	3	0
None	7	7	0

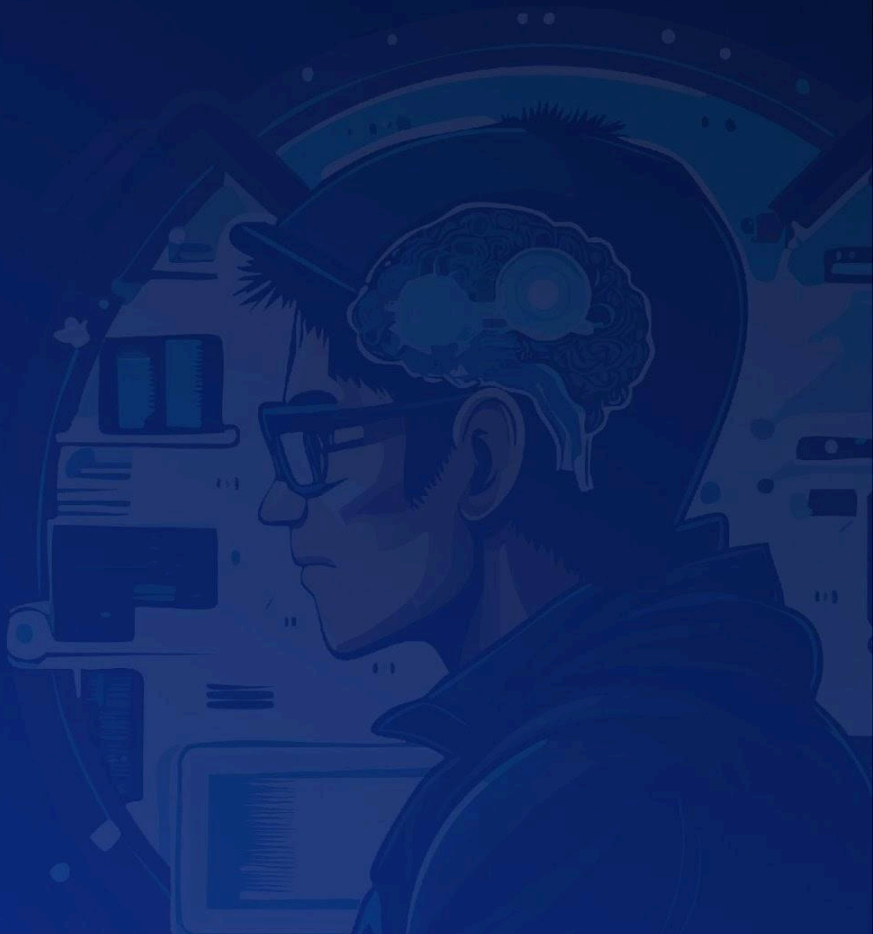
Category Breakdown

Suggestion	7
Documentation	0
Bug	17
Optimization	0
Good Code Practices	0

Risk Section

Security Review

Soneta Borrowing Protocol



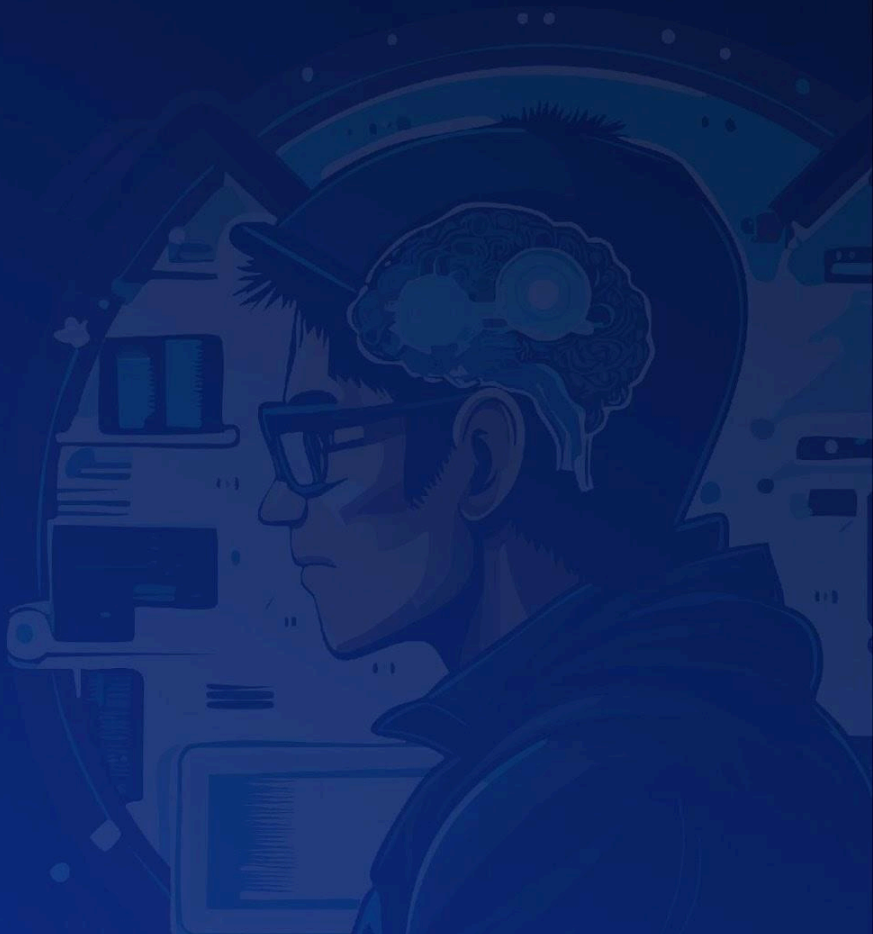
Risk Section

- **Partial Audit Scope**: The audit covers only specified contracts within the repository. Vulnerabilities in contracts outside this scope could introduce risks, potentially impacting overall system security.
- **Administrative Key Management Risks**: The system makes use of administrative keys to manage critical operations. Compromise or misuse of these keys could result in unauthorized actions and financial losses.

Findings

Security Review

Soneta Borrowing Protocol



Findings

3S-Soneta-C01

Collateral Gas Compensation Cap May Disincentivize Liquidations for Low-Value Assets

Id	3S-Soneta-C01
Classification	Critical
Impact	Critical
Likelihood	High
Category	Bug
Status	Addressed in #106a515.

Description:

During liquidations, the protocol rewards the liquidator with two forms of compensation:

* A fixed **ETH_GAS_COMPENSATION** of 0.0375 ETH

* A **COLL_GAS_COMPENSATION**, calculated as the lesser of: 0.5% of the Trove's collateral or 2 units (ether) of the collateral

This compensation structure assumes the collateral is a high-value asset (e.g., ETH on the Ethereum chain), where even a small percentage of the collateral translates to a meaningful reward.

However, in the case of **Sonata**, assets like WrappedOS (WOS) are used as collateral. At the time of writing (May 22, 2025), 1 WOS ≈ 0.15 USD, making the 2-unit collateral cap irrelevant, as liquidators will almost always receive the capped 2 units, which in dollar terms may be negligible.

As a result, liquidators may not be sufficiently incentivized to act.

Liquidations of small or low-value positions may be delayed or missed, increasing the risk of under-collateralized positions remaining in the system.

This degrades the protocol's health, solvency, and responsiveness in volatile market conditions.

Recommendation:

* Increase the fixed **ETH_GAS_COMPENSATION** to better match the Sonic network's needs.

* Removing the 2-unit collateral cap from **COLL_GAS_COMPENSATION** would be the easiest solution, leaving 0.5% of the Trove's collateral as the liquidator's compensation. But as a consequence, it would affect large Trove owners slightly more than the current design with the cap. Ideally, this should be addressed by implementing asset-specific compensation caps that adjust based on asset price.

3S-Soneta-C02

Oracle returns underlying asset price without accounting for LST exchange rate

Id	3S-Soneta-C02
Classification	Critical
Impact	Critical
Likelihood	Medium
Category	Bug
Status	Addressed in #1139464.

Description:

The current price feed used for Liquid Staking Tokens (LSTs) returns the price of the underlying asset (e.g., S) without factoring in the LST-to-underlying exchange rate. This approach is flawed, especially for rebasing tokens, where the exchange rate is not guaranteed to be 1:1.

Consequences of this design flaw include:

Collateral undervaluation:

Users supplying LSTs as collateral will see them undervalued, reducing borrowing power and discouraging protocol usage.

Oracle exploitation risk in depeg events:

If the LST depegs significantly from its underlying due to issues with the staking provider (as seen in past cases like Renzo read more), the oracle will continue reporting the underlying price instead of the market value of the LST.

Attackers could buy depegged LSTs at a discount, deposit them as collateral, borrow a large amount of Bold, and immediately sell it for a profit.

This could cause a sudden and severe peg collapse because the system would allow overborrowing based on incorrect valuations.

Since the system is blind to the LST's true market value, emergency safeguards such as SCR thresholds would not be triggered in time, delaying any meaningful response.

Recommendation:

Adopt a composite price feed for LSTs that reflects their true market value, incorporating both:

* The underlying asset's price (e.g., S/USD)

* The LST-to-underlying exchange rate, preferably from both canonical and market sources

This should be adopted for LSTs such as Wrapped Origin Sonic and Angels Sonic.

3S-Soneta-H01

Improper oracle use for rebasing collateral may enable arbitrage and overborrowing

Id	3S-Soneta-H01
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #5164742.

Description:

The Sonata contract uses **fetchPrice()** (from **SingleChainlinkPriceFeed** and **WrappedOriginPriceFeed**) to determine collateral value during borrowing operations, and **fetchRedemptionPrice()** during redemptions.

While this model is sufficient for non-rebasing assets, it is inadequate for rebasing assets such as liquid staking tokens (e.g., wOS, ST). These tokens diverge from a 1:1 relationship with their underlying asset (e.g., S) and require a more comprehensive valuation model.

The core issue is not just the lack of exchange rate usage, but the lack of full price awareness. Since Sonata interacts deeply with the broader market to maintain its peg (especially through redemptions), its Oracle system must capture both protocol-defined values and real-world pricing signals.

For rebasing collateral, proper pricing should incorporate:

* S/USD price (base asset value)

* Canonical LST rate (protocol-defined or contract-based)

* Market LST rate (sourced from DEXs or aggregators)

Failing to combine canonical and market rates introduces several risks:

If only the canonical rate is used:

* Market price artificially inflated → Users redeem collateral at a discount, sell at a premium → Peg may be pushed upward.

* Market price artificially suppressed → Users borrow against undervalued collateral → Sell token → Peg may be pushed downward.

* Canonical rate artificially inflated → Overborrowing risk → Bold token sell pressure → Peg downward.

* Canonical rate artificially suppressed → Mass redemptions for cheap collateral → Sell for profit → Peg upward.

If only the market rate is used:

* Same risks as Scenario 1, but with inverse consequences based on market dynamics.

In both cases, ignoring one side of the pricing environment blinds the protocol to critical dynamics that maintain peg stability.

Recommendation:

Design an Oracle system that captures global price awareness by incorporating:

1. The base asset price (e.g., ETH/USD or S/USD)

2. The canonical exchange rate for the rebasing asset

3. The market exchange rate from on-chain sources (e.g., DEXs)

Implement logic to compare these rates with a deviation threshold (tunable per asset, higher for low-liquidity tokens):

* For **fetchPrice()** (borrowing): Use the lower of canonical and market exchange rates.

* For **fetchRedemptionPrice()** (redemptions): Use the higher of the two rates if within the threshold; otherwise, fall back to the lower to avoid undervaluing redemptions.

Finally, the protocol should create a detailed Oracle strategy similar to Liquity's oracle design documentation.

3S-Soneta-H02

Misplaced curly braces in GasCompZapper::_transferCollateralInside leads to incorrect collateral transfer logic

Id	3S-Soneta-H02
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #eba22fe.

Description

The **GasCompZapper::_transferCollateralInside** function is responsible for transferring collateral into the zapper contract. This function checks if the collateral token is standardized using the **isSanitized** flag. If the token is not standardized, it uses **SafeERC20** to transfer the collateral directly. However, if the token is standardized, it should convert the collateral amount to token decimals, transfer the underlying token, and then wrap it. Due to misplaced curly braces, the logic for handling standardized tokens is never executed, leading to incorrect collateral transfer when **isSanitized** is true. This results in the zapper contract not receiving the intended wrapped collateral, causing failures in subsequent operations that rely on the correct collateral balance.

Recommendation

To fix the issue, the misplaced curly brace should be moved to correctly encapsulate the logic for handling standardized tokens. The corrected code ensures that the wrapping logic is executed when **isSanitized** is true.

```
function _transferCollateralInside(address _sender, uint256 _amount) internal {
    if (!isSanitized) {
        collToken.safeTransferFrom(_sender, address(this), _amount);
        return;
    }
    ERC20StandardizedWrapper standardizedWrapper =
        ERC20StandardizedWrapper(address(collToken));
    uint256 amountInTokenDecimals =
standardizedWrapper.etherToTokenDecimals(_amount);
```

```
    standardizedWrapper.INPUT_TOKEN().safeTransferFrom(
        _sender, address(this), amountInTokenDecimals
    );
    standardizedWrapper.wrap(_amount);
}
```

3S-Soneta-H03

Unhandled exchange rate call may cause protocol freeze and insolvency

Id	3S-Soneta-H03
Classification	High
Impact	Critical
Likelihood	Low
Category	Bug
Status	Addressed in #bbe4c2a.

Description:

The **_fetchPricePrimary()** function within the WrappedOriginPriceFeed contract includes a direct call to the LST's exchange rate via **WOS_TOKEN.convertToAssets(...)**, but this call is not wrapped in a try/catch block. This omission introduces a serious risk.

ethUsdPrice = WOS_TOKEN.convertToAssets(1e18) * ethUsdPrice / 1e18;

If the **convertToAssets** call reverts due to LST contract issues, external dependencies, or gas-related problems, the entire function will revert, and the following consequences may occur:

* Emergency shutdown is not triggered because the error isn't caught.

* **lastGoodPrice** is not updated, and no fallback mechanism is used.

* All operations that depend on this price feed, such as borrowing, redeeming, or liquidations, will revert and become unusable.

If the reverting behavior persists for a prolonged period or becomes permanent, this could result in:

* Stuck collateral and positions

* Blocked liquidations

* Protocol insolvency, as price-dependent mechanisms cannot function

This risk is particularly dangerous in volatile or adversarial conditions, where quick failover to the last known good price is essential to maintain system integrity.

Recommendation:

* Wrap the LST exchange rate call in a **try/catch** block to handle errors gracefully and trigger fallback logic, for example, using **lastGoodPrice**.

* Implement gas usage checks to detect and avoid out-of-gas errors during oracle resolution.

* Ensure the emergency shutdown mechanism is triggered when the LST feed fails, and that all dependent operations are informed, for example, via return flags or state updates.

3S-Soneta-H04

Incorrect approval call in the GasCompZapper::constructor leads to DoS during wrapping

Id	3S-Soneta-H04
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #31b51ac.

Description

In the **GasCompZapper::constructor**, when the **ERC20StandardizedWrapper** is used inside the **try/catch** block, an unlimited approval of the **INPUT_TOKEN** is made to the **BorrowerOperations** contract.

This is incorrect, as the intention of this approval is to allow **INPUT_TOKEN** wrapping. Therefore, the unlimited approval should be given to the **ERC20StandardizedWrapper** collateral token contract.

Recommendation

To fix the issue, replace **borrowerOperations** with **collToken** in the approval call:

```
    isSanitized = true;
    // Allow Wrapping
    ERC20StandardizedWrapper(address(collToken)).INPUT_TOKEN().approve(
-       address(borrowerOperations), type(uint256).max
+       address(collToken), type(uint256).max
    );
```

3S-Soneta-H05

Incorrect price will be returned from the LBTCPriceFeed

Id	3S-Soneta-H05
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #8a6bd27.

Description

All oracle integrations within **Sonata** are expected to return prices denominated in USD with 18-decimal precision.

In the case of the **LBTCPriceFeed** contract, the hardcoded Chainlink price feed used is **LBTC / BTC**. However, the returned price is never normalized to USD using another Chainlink feed that provides the **BTC / USD** price.

As a result, the **LBTC** price reported in **Sonata** will appear to be around 1 USD or less, since the current **LBTC / BTC** rate is approximately 0.99919598 LBTC per BTC.

In addition to the incorrect **LBTCPriceFeed** implementation, a similar issue exists in the deployment scripts and configuration. For the **LBTC** collateral, the oracle address used is **0xA63b1614D17536C22fDB4c1a58023E35d08Cccef** (**LBTC / BTC**), and it is incorrectly intended to be used with the **WETHPriceFeed** contract:

```
if (string_equals(_branch.branchName, "LBTC")) {
  (priceFeed,) = _tryDeployContractCREATE2(
    "LBTCPriceFeed",
    DEPLOYMENT_SALT,
@>  type(WETHPriceFeed).creationCode,
    abi.encode(
      _branch.oracle,
      _branch.stalenessThresholdHours * 1 hours,
      _borrowerOperationsAddress
    )
  );
}
```

Recommendation

The **LBTCPriceFeed** implementation must return the price in USD. This can be achieved by additionally fetching the **BTC / USD** price and combining it with the **LBTC / BTC** rate. The implementation of **RETHPriceFeed** serves as a good reference for how this should be done.

3S-Soneta-M01

Incomplete debt accounting causes debt cap drift

Id	3S-Soneta-M01
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #1144111.

Description

The OneToken contract enforces a per-branch minting limit via **debtTracker** and **debtCapPerBranch**. The **debtTracker** is incremented or decremented only when the BorrowerOperations contract calls the **mint** or **burn** functions. Once **debtTracker** reaches **debtCapPerBranch**, minting is blocked.

The core Issue is that the accounting is incomplete, since the debt is also reduced through operations such as:

- Liquidations – triggered via the TroveManager

- Redemptions – handled by the CollateralRegistry

These events reduce actual system debt, but they do not update **debtTracker**. As a result, the tracker stays artificially high, blocking minting even though the system has room under its actual debt cap. The only workaround becomes artificially increasing **mintCap**, which eventually decouples it from true system conditions.

Consequences:

- **debtTracker** becomes a misleading measure of real outstanding debt.

- **mintCap** inflation undermines its purpose as a risk control mechanism.

- Fine-grained debt governance becomes impossible over time.

Recommendation

Even if more functions were added to manually update **debtTracker** during liquidations and redemptions, the logic remains fragile and difficult to maintain, since the design relies on tracking debt indirectly and incompletely across multiple contracts.

The recommended Solution is to adopt a different enforcement model, specifically:

- Move the mint cap check into the BorrowerOperations contract.

- Wrap all mint-related functions in a **require** statement that compares the actual debt (**getEntireBranchDebt**) against the **mintCap**.

- Revert the mint if it would cause the cap to be exceeded.

Benefits of the new approach:

- Accurate enforcement of the cap regardless of how debt is reduced.

- Simplified and maintainable logic.

- Greater security and confidence in system-wide debt limits.

3S-Soneta-M02

Incorrect unwrapping of non-wrapped tokens leads to denial of service

Id	3S-Soneta-M02
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #a60d066.

Description

The protocol uses **ERC20StandardizedWrapper** to standardize tokens with non-18 decimals to be used consistently within the protocol. The **GasCompZapper** and **LeverageLSTZapper** contracts handle these tokens and perform wrap/unwrap operations based on the **isSanitized** flag, which indicates whether the collateral token is standardized.

Two instances of the **_tryToUnwrapCollateral** function are called when the received flash loan amount needs to be swapped to ONE tokens. The issue is that the tokens received by the flashloan are already unwrapped. When called on already unwrapped tokens, the function attempts to unwrap them again, which will cause the transaction to revert.

These instances occur in:

1. **GasCompZapper::receiveFlashLoanOnCloseTroveFromCollateral** (line 338)
2. **LeverageLSTZapper::receiveFlashLoanOnLeverDownTrove** (line 213)

In both cases, the function attempts to unwrap tokens that are already in their unwrapped form before swapping them to ONE tokens, causing all operations involving these functions to fail when **isSanitized** is true.

Recommendation

Remove the highlighted **_tryToUnwrapCollateral** calls from the code.

3S-Soneta-M03

Flash loan request with wrapped token instead of unwrapped token leads to transaction failure

Id	3S-Soneta-M03
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #0d186c4.

Description

The **GasCompZapper** and **LeverageLSTZapper** contracts implement functionality that allows users to perform various operations with collateral tokens, including leveraging positions and closing troves. The protocol uses **ERC20StandardizedWrapper** to standardize tokens with non-18 decimals to 18 decimals for use in the protocol.

When a collateral token is wrapped (indicated by **isSanitized = true**), the zappers should use the unwrapped/underlying token when interacting with external services like flash loan providers. However, in several instances, the contracts incorrectly pass the wrapped token (**collToken**) to **flashLoanProvider.makeFlashLoan()** instead of the underlying token.

This issue appears in multiple locations:

1. In **GasCompZapper::closeTroveFromCollateral()**
2. In **LeverageLSTZapper::openLeveragedTroveWithRawETH()**
3. In **LeverageLSTZapper::leverUpTrove()**
4. In **LeverageLSTZapper::leverDownTrove()**

Since Balancer vaults used by the flash loan provider do not have pools for the wrapped tokens, these transactions will revert when **isSanitized** is true, making these features unusable for wrapped tokens.

Recommendation

Check the **isSanitized** flag before making flash loan requests and use the underlying token when the collateral token is wrapped: **ERC20StandardizedWrapper(address(collToken)).INPUT_TOKEN()**.

3S-Soneta-M04

Incorrect decimal scaling in flash loan amount causes flash loan failures with wrapped tokens

Id	3S-Soneta-M04
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #a60d066.

Description

The Sonata protocol includes a mechanism to handle tokens with different decimal precision by wrapping them into standardized 18-decimal tokens using the **ERC20StandardizedWrapper** contract. The **GasCompZapper** and **LeverageLSTZapper** contracts support operations with these wrapped tokens, identified by the **isSanitized** flag set to true.

Similar to issue M-03, when using flash loans for leveraged operations the contracts fail to convert the flash loan amount from 18 decimals to the underlying token's decimal format. The issue occurs in multiple functions across both contracts:

1. In **LeverageLSTZapper::openLeveragedTroveWithRawETH**, the **flashLoanProvider.makeFlashLoan** is called with the wrapped token 18-decimal amount.

2. In **LeverageLSTZapper::leverUpTrove**, the same incorrect scaling occurs.

3. In **LeverageLSTZapper::leverDownTrove**, the issue is repeated.

4. In **GasCompZapper::closeTroveFromCollateral**, the flash loan also uses the wrong decimal scaling.

Additionally, at the beginning of the flash loan callback functions, the received amount (**_effectiveFlashLoanAmount**) needs to be converted back to 18 decimals for **borrowerOperations** operations, but this conversion is missing:

1. In **LeverageLSTZapper::receiveFlashLoanOnOpenLeveragedTrove**

2. In **LeverageLSTZapper::receiveFlashLoanOnLeverUpTrove**

Finally, **_params.flashLoanAmount** needs to be scaled down to 8 decimals before being fed to **exchange.swapFromOne** in the same flash loan callback functions mentioned above:

1. In **LeverageLSTZapper::receiveFlashLoanOnOpenLeveragedTrove**
2. In **LeverageLSTZapper::receiveFlashLoanOnLeverUpTrove**

Note: it's suggested to perform this scaling either inline or on a new local variable, since **_params.flashLoanAmount** needs to stay in 18 decimals for the following call to **_transferCollateralOutside(address(flashLoanProvider), _params.flashLoanAmount);**.

The problem arises because Balancer pools (used by the flash loan provider) contain the underlying tokens, not the wrapped versions. This mismatch in decimal precision will cause flash loans to fail or request incorrect amounts, breaking the leverage functionality for tokens with non-18 decimals.

Recommendation

When interacting with flash loans for wrapped tokens, convert the amount between 18-decimal format and the underlying token's format. This fix should be applied to all instances where flash loans are requested with wrapped tokens and where the received flash loan amount is used in operations requiring 18-decimal precision.

3S-Soneta-M05

Inconsistent token wrapping in flash loan callbacks leads to transaction failures with standardized tokens

Id	3S-Soneta-M05
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #c7362ef.

Description

The **LeverageLSTZapper** and **GasCompZapper** contracts implement mechanisms to handle both regular ERC20 tokens and those wrapped in a standardized wrapper (identified by the **isSanitized** flag). The contracts include flash loan functionality where tokens are borrowed and then used in various operations through callback functions such as **receiveFlashLoanOnOpenLeveragedTrove**.

When the protocol is working with standardized wrapped tokens (**isSanitized == true**), the flash loan provider sends the unwrapped/underlying tokens to the contract. However, the callback functions fail to properly wrap these tokens before using them with other contract calls, particularly with **BorrowerOperations**, which expects the wrapped version when **isSanitized** is true. The issue arises in **LeverageLSTZapper::receiveFlashLoanOnOpenLeveragedTrove** and **LeverageLSTZapper::receiveFlashLoanOnLeverUpTrove**.

Recommendation

Ensure that all flash loan callbacks properly wrap the received tokens before using them with other operations when working with standardized wrapped tokens. Call the **_tryWrapCollateral** function before using the tokens in operations that expect wrapped tokens.

3S-Soneta-M06

Missing decimal precision conversion in token wrapping leads to severe value loss

Id	3S-Soneta-M06
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #cccff10.

Description

The **GasCompZapper::_tryWrapCollateral** function is responsible for wrapping underlying tokens into their standardized wrapped form when working with standardized tokens (**isSanitized** is true). However, the function fails to account for potential decimal precision differences between the underlying token and the standardized wrapper.

```
function _tryWrapCollateral() internal {
    if (!isSanitized) {
        return;
    }
    uint256 amountToken =
ERC20StandardizedWrapper(address(collToken)).INPUT_TOKEN().balanceOf(address(this
));
    ERC20StandardizedWrapper(address(collToken)).wrap(amountToken);
}
```

The function retrieves the balance of the underlying token using **balanceOf()**, which returns the amount in the token's native decimal precision. It then passes this raw amount directly to the **wrap()** function, which expects values in the standardized token's decimal precision (typically 18 decimals).

This mismatch causes severe undervaluation of the wrapped amount. For example, if wrapping 1 WBTC (100,000,000 in 8 decimals) without proper conversion to 18 decimals, the **wrap()** function would interpret this as 0.0000000001 tokens in 18-decimal precision, effectively losing 99.9999999% of the value.

This issue affects all operations that rely on the **_tryWrapCollateral** function when working with tokens that have decimal precision different from the standardized wrapper's precision.

Recommendation

Modify the **_tryWrapCollateral** function to properly convert the token amount from the underlying token's decimal precision to the standardized wrapper's decimal precision:

```
function _tryWrapCollateral() internal {
    if (!isSanitized) {
        return;
    }
    ERC20StandardizedWrapper standardizedWrapper =
ERC20StandardizedWrapper(address(collToken));
    IERC20 inputToken = standardizedWrapper.INPUT_TOKEN();
    uint256 amountToken = inputToken.balanceOf(address(this));
+
+    // Convert from input token's decimals to standardized wrapper's decimals (18)
+    uint8 inputDecimals = ERC20(address(inputToken)).decimals();
+    if (inputDecimals != 18) {
+        amountToken = amountToken * 10**(18 - inputDecimals);
+    }
+
    standardizedWrapper.wrap(amountToken);
}
```

3S-Soneta-M07

Incorrect token approval for exchange when using standardized wrappers leads to transaction failures

Id	3S-Soneta-M07
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #0bf1d35.

Description

The **GasCompZapper** contract is designed to handle both regular ERC20 tokens and standardized wrapped tokens (identified by the **isSanitized** flag). When standardized tokens are used, the contract needs to properly manage both the wrapped token (**collToken**) and its underlying token (**INPUT_TOKEN()**).

In the constructor, the contract sets up approvals for the exchange to facilitate token swaps:

```
constructor(
  IAddressesRegistry _addressesRegistry,
  IFlashLoanProvider _flashLoanProvider,
  IExchange _exchange
) BaseZapper(_addressesRegistry, _flashLoanProvider, _exchange) {
  collToken = _addressesRegistry.collToken();
  try ERC20StandardizedWrapper(address(collToken)).isStandardized() returns (bool
isStandardized) {
    isSanitized = true;
    //Allow Wrapping
    ERC20StandardizedWrapper(address(collToken)).INPUT_TOKEN().approve(
      address(borrowerOperations), type(uint256).max
    );
  } catch (bytes memory) {
    isSanitized = false;
  }
  // ... other code ...
  // Approve Coll to exchange module (for closeTroveFromCollateral)
  if (address(_exchange) != address(0)) {
```

```
    collToken.approve(address(_exchange), type(uint256).max); // The issue is here
  }
}
```

The issue is that when **isSanitized** is **true**, the contract always approves the wrapped token (**collToken**) to the exchange, but during operations like **swapToOne** in exchanges, the contract unwraps the tokens using **_tryToUnwrapCollateral**:

```
function _tryToUnwrapCollateral(uint256 _amount) internal {
    if (!isSanitized) {
        return;
    }
    ERC20StandardizedWrapper(address(collToken)).unwrapTo(address(this), _amount);
}
```

This means the exchange will attempt to use the unwrapped token (**INPUT_TOKEN()**), but it only has approval for the wrapped token (**collToken**), causing transactions to fail due to insufficient allowance.

Recommendation

Modify the constructor to approve the correct token to the exchange based on the **isSanitized** flag:

```
// Approve Coll to exchange module (for closeTroveFromCollateral)
if (address(_exchange) != address(0)) {
-   collToken.approve(address(_exchange), type(uint256).max);
+   if (isSanitized) {
+
ERC20StandardizedWrapper(address(collToken)).INPUT_TOKEN().approve(address(_exc
hange), type(uint256).max);
+   } else {
+     collToken.approve(address(_exchange), type(uint256).max);
+   }
}
```

3S-Soneta-L01

Truncation during wrapping can lead to unexpected reverts in the GasCompZapper and LeverageLSTZapper contracts

Id	3S-Soneta-L01
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #1463adb.

Description

When user-provided collateral amounts are used as input in the **ERC20StandardizedWrapper** wrapping process, there is a possibility that the expected amount will not match the actually wrapped tokens due to truncation of dust amounts within the **ERC20StandardizedWrapper::wrap** function.

```
function wrap(uint256 _etherAmount) external {
@>  uint256 scaleToTokenInputDecimals = _etherAmount / DECIMALS_SCALE;
    INPUT_TOKEN.safeTransferFrom(msg.sender, address(this),
scaleToTokenInputDecimals);
    // @dev We scale it back to avoid input manipulation.
    _mint(msg.sender, scaleToTokenInputDecimals * DECIMALS_SCALE);
}
```

This issue can partially be seen as a user error. However, it can lead to unexpected reverts in the following functions:

* **GasCompZapper::openTroveWithRawETH**
* **GasCompZapper::addColl**
* **GasCompZapper::adjustTrove**
* **GasCompZapper::adjustZombieTrove**
* **LeverageLSTZapper::openLeveragedTroveWithRawETH**

Recommendation

This behavior should either be explicitly acknowledged and prevented on the frontend by ensuring that for **ERC20StandardizedWrapper** tokens, no dust amount is present below the token's decimal precision, or the **_transferCollateralInside** function should return the actual amount of tokens transferred. For example:

```
function addColl(uint256 _troveId, uint256 _amount) external {
  address owner = troveNFT.ownerOf(_troveId);
  _requireSenderIsOwnerOrAddManager(_troveId, owner);
  IBorrowerOperations borrowerOperationsCached = borrowerOperations;
  // Pull coll
- _transferCollateralInside(msg.sender, _amount);
+ _amount = _transferCollateralInside(msg.sender, _amount);
  borrowerOperationsCached.addColl(_troveId, _amount);
}
```

3S-Soneta-L02

Multiple assignments to immutable variable in HybridCurveShadowExchange leads to compilation failures with pre-0.8.21 Solidity versions

Id	3S-Soneta-L02
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #5906dbf.

Description

The **HybridCurveShadowExchange** contract is a contract that facilitates token swaps between ONE tokens and collateral tokens using both Curve and Shadow (Uniswap V3) liquidity. The contract declares **collToken** as an immutable state variable but attempts to assign it twice within the constructor.

```
IERC20 public immutable collToken; // immutable variable
constructor(
    IERC20 _collToken,
    IOneToken _oneToken,
    IERC20 _usdc,
    IWETH _weth,
    // Curve
    ICurveStableswapNGPool _curvePool,
    uint128 _usdcIndex,
    uint128 _oneIndex,
    // UniV3
    ISwapRouter _uniV3Router,
    bytes memory _collateralToOnePath,
    bytes memory _USDCToCollateralPath
) {
    collToken = _collToken; // First assignment
    oneToken = _oneToken;
    USDC = _usdc;
    WETH = _weth;
```

```
    // Curve
    curvePool = _curvePool;
    USDC_INDEX = _usdcIndex;
    ONE_TOKEN_INDEX = _oneIndex;
    // Uniswap
    uniV3Router = _uniV3Router;
    COLLATERAL_TO_USDC_PATH = _collateralToOnePath;
    USDC_TO_COLLATERAL_PATH = _USDCToCollateralPath;
    try
        ERC20StandardizedWrapper(address(collToken)).isStandardized()
    returns (bool) {
        collToken = ERC20StandardizedWrapper(address(collToken)) // Second assignment
            .INPUT_TOKEN();
    } catch {}
}
```

In Solidity versions prior to 0.8.21, immutable variables can only be assigned once during contract construction. Any subsequent assignment, even within the constructor, will result in a compilation error. The contract specifies pragma **^0.8.18**, which allows compatibility with older versions where this issue would manifest.

Solidity 0.8.21 introduced a change allowing multiple assignments to immutable variables within the constructor, with the last assignment being the one retained. This creates an inconsistent behavior depending on which compiler version is used.

Recommendation

To resolve this issue, avoid multiple assignments to the immutable variable by restructuring the code. Use a local variable to determine the final value before the single immutable assignment:

Alternatively, update the pragma to explicitly require a compiler version that supports multiple immutable assignments:

```
- pragma solidity ^0.8.18;
+ pragma solidity ^0.8.21;
```

3S-Soneta-L03

Zapper does not refund intermediate tokens from swap path

Id	3S-Soneta-L03
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #10a7f0d.

Description

Zapper contracts such as **LeverageLSTZapper** and **GasCompZapper** only track and refund the input and output tokens (for example, **oneToken** and **collToken**) during swaps. Intermediate tokens used in multi-hop paths, such as USDC and WS (WETH), are not tracked and may remain stuck in the zapper contract.

This issue arises from a mismatch between how **HybridCurveShadowExchange** and the zapper handle token balance tracking. In the zapper, the **_setInitialTokensBalancesAndReceiver** function only tracks **collToken** and **oneToken**:

```
function _setInitialTokensBalancesAndReceiver(
  IERC20 _collToken,
  IOneToken _oneToken,
  InitialBalances memory _initialBalances,
  address _receiver
) internal view {
  _initialBalances.tokens[0] = _collToken;
  _initialBalances.tokens[1] = _oneToken;
  _setInitialBalancesAndReceiver(_initialBalances, _receiver);
}
```

In contrast, **HybridCurveShadowExchange** correctly tracks all relevant tokens, including intermediates like USDC and wS:

```
function _setHybridExchangeInitialBalances(InitialBalances memory initialBalances)
internal view {
  initialBalances.tokens[0] = oneToken;
```

```
  initialBalances.tokens[1] = USDC;
  initialBalances.tokens[2] = WETH;
  if (address(WETH) != address(collToken)) {
    initialBalances.tokens[3] = collToken;
  }
  _setInitialBalances(initialBalances);
}
```

When **HybridCurveShadowExchange** returns unspent tokens to the zapper, only **oneToken** and **collToken** are forwarded to the user. Any leftover USDC or WS remains stuck in the zapper.

It can only result in dust amounts only, and the **minAmount** condition ensures the user still receives the expected value from the swap.

Recommendation

To improve fund recovery, the zapper contracts can be updated to match the full token tracking used by **HybridCurveShadowExchange** and return all unspent tokens, including intermediates. However, the current logic does not significantly affect user outcomes.

3S-Soneta-N01

Redundant imports in multiple contracts

Id	3S-Soneta-N01
Classification	None
Category	Suggestion
Status	Addressed in #2e1e525.

Description

Several contracts contain unused or redundant import statements:

* **ChainlinkPriceFeed**:

```
import "../Interfaces/IMainnetPriceFeed.sol";
```

* **HybridCurveShadowExchange**:

```
import "openzeppelin-contracts/contracts/utils/math/Math.sol";
```

* **HybridCurveShadowExchangeHelpers**:

```
import "../../../Interfaces/IWETH.sol";
```

Recommendation

Remove the unused import statements from the respective contracts to improve code cleanliness and maintainability.

3S-Soneta-N02

Typos in comments across contracts

Id	3S-Soneta-N02
Classification	None
Category	Suggestion
Status	Addressed in #fdc0972.

Description

Minor typos were found in comments within the following contracts:

* **ChainlinkPriceFeed**:

```
// - primary: Uses the primary price calcuation, which depends on the specific feed
// Typo: "calcuation" → "calculation"
```

* **LeverageLSTZapper**:

```
// Set initial balances to make sure there are not lefovers
// Typo: "lefovers" → "leftovers"
```

Recommendation

Correct the typos to improve code readability and maintain professional documentation standards.

3S-Soneta-N03

Functions can be declared **external** instead of **public** in ERC20StandardizedWrapper

Id	3S-Soneta-N03
Classification	None
Category	Suggestion
Status	Addressed in #c560afa.

Description

The following functions in the **ERC20StandardizedWrapper** contract are declared as **public** but are not used internally. Therefore, they can be marked as **external** to optimize gas usage and improve clarity:

* **etherToTokenDecimals**

* **tokenDecimalsToEther**

* **isStandardized**

Recommendation

Update the function visibility from **public** to **external** for the above functions to follow best practices and reduce contract size and gas costs where applicable.

3S-Soneta-N04

Dust loss due to truncation in **ERC20StandardizedWrapper::_unwrap**

Id	3S-Soneta-N04
Classification	None
Category	Suggestion
Status	Addressed in #1463adb.

Description

The **_unwrap** function in the **ERC20StandardizedWrapper** contract burns the full **_etherAmount** but transfers back only **_etherAmount / DECIMALS_SCALE** of the **INPUT_TOKEN**. This results in loss of dust amounts due to truncation and may lead to unclaimable residual tokens over time.

```
function _unwrap(address _receiver, uint256 _etherAmount) internal {
  _burn(msg.sender, _etherAmount); // burns full amount
  INPUT_TOKEN.safeTransfer(_receiver, _etherAmount / DECIMALS_SCALE); // returns truncated value
}
```

Ideally, the function should burn **_etherAmount / DECIMALS_SCALE * DECIMALS_SCALE** to align with the actual amount returned, preserving any dust in the system rather than burning it.

Recommendation

No immediate fix is required as this is informational. However, it's important to be aware that dust amounts will accumulate and be lost over time. This behavior could be documented or explicitly handled if precise accounting is critical.

3S-Soneta-N05

Unnecessary commented-out code and TODOs can be removed

Id	3S-Soneta-N05
Classification	None
Category	Suggestion
Status	Addressed in #606ce90.

Description

The codebase contains commented-out import statements and incomplete TODO comments that no longer serve a functional purpose and can reduce readability:

* **Redundant commented-out imports**:

```
// import "forge-std/console2.sol";
```

* **Stale TODO comments**:

```
// TODO: pass it as param in functions, so we can reuse the same exchange for different
```

Recommendation

Remove all instances of the commented-out **console2.sol** import and outdated TODO comments to improve code cleanliness and maintainability.

3S-Soneta-N06

Missing **msg.sender** check in **receive()** function of GasCompZapper

Id	3S-Soneta-N06
Classification	None
Category	Suggestion
Status	Addressed in #8280a0d.

Description

The **GasCompZapper** contract defines a **receive()** function to accept ETH transfers:

```
receive() external payable { }
```

However, this function currently lacks a check to ensure that the ETH is only being sent by the expected **WETH** contract. Without this check, users may mistakenly send ETH directly to the contract, resulting in unintended lost funds.

Recommendation

Add a **require** statement in the **receive()** function to restrict ETH transfers to only those originating from the **WETH** contract:

```
receive() external payable {
  require(msg.sender == address(WETH), "Only WETH can send ETH");
}
```

3S-Soneta-N07

Potential approve revert when changing non-zero allowance for collToken

Id	3S-Soneta-N07
Classification	None
Category	Suggestion
Status	Addressed in #1463adb.

Description

In the **HybridCurveShadowExchange::swapToOne** function, **collToken.approve** is called directly to set a new allowance. If **collToken** behaves like USDT or other tokens that require the allowance to be first set to zero before changing to a new non-zero amount, this call can revert.

Recommendation

Use **forceApprove** or follow the allowance reset pattern (set allowance to zero before setting the new amount) to prevent potential reverts when updating allowances.