Quantstamp Security Assessment Certificate

February 15th 2023 — Quantstamp Verified

Sturdy (Aura Integration And Leverage)

This audit report was prepared by Quantstamp, the leader in blockchain security.

Executive Summary

Type	Lending Protocol
Auditors	Zeeshan Meghji, Auditing Engineer Andy Lin, Senior Auditing Engineer Ruben Koch, Auditing Engineer II Sina Pilehchiha, Audit Engineer I
Timeline	2022-12-12 through 2022-12-22
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Sturdy Documentation
Documentation Quality	Medium
Test Quality	Low
Source Code	see table below

Repository	Commit
sturdyfi/sturdy	a369675 initial audit

Total Issues	**29** (16 Resolved)
High Risk Issues	**2** (2 Resolved)
Medium Risk Issues	**2** (1 Resolved)
Low Risk Issues	**11** (8 Resolved)
Informational Risk Issues	**8** (1 Resolved)
Undetermined Risk Issues	**6** (4 Resolved)

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
Undetermined	The impact of the issue is uncertain.

Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Fixed	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

The Sturdy Protocol has introduced new functionality, including leveraged yield farming and a new vault that integrates with the Aura protocol. The leveraged yield farming feature lets users increase their collateral holdings within their desired vault by purchasing more collateral with funds borrowed from Sturdy. The Aura vault works very similarly to existing vaults, especially the older Convex vault.

The auditing team found two critical issues with the new leverage contracts. QSP-1 describes a potential exploit that could be used to steal other users' collateral due to insufficient reentrancy protection and input validation. QSP-2 demonstrates the loss of user and protocol value due to broken slippage controls. Other notable findings relate to the incorrect implementation or validation of oracle price feeds (QSP-3, QSP-4, QSP-5). We strongly recommend that the Sturdy team fixes all issues found within the report.

While the Sturdy team has implemented many tests, including integration tests, the quality and coverage of these tests need to be improved. Many important behaviors are not being validated, and several critical scenarios have not been tested. We have provided suggestions for improvement in the "Test Results" section.

The new functionality needs to be sufficiently documented. Sparse documentation makes it difficult for auditors to verify if the contracts function as the team intends them to. The documentation lacks both high-level conceptual explanations and details on the code-level. We highly recommend that the Sturdy team improve architectural, conceptual, and code documentation.

Update: Following the fix review phase, all but two issues have either been fixed, mitigated, or acknowledged. QSP-4 and QSP-11 remain unresolved and should be addressed. QSP-4 describes the possibility of calculating an incorrect collateral token price, the consequences of which could be serious. QSP-11 demonstrates a potential loss of precision in some calculations and should also be resolved.

The code-level documentation has been improved, and most functions and events have been documented according to the NatSpec standard. However, only minimal changes have been made to the test suite, which still needs improvement, as described above. The potential consequences of insufficient testing include unexpected functional bugs.

Update after final fix review: No vulnerabilities within the report remain unresolved following the final fix review. However, a few concerns do remain regarding the calculation of the Balancer LP token price when the prices of ETH and STETH are significantly different (QSP-4).

ID	Description	Severity	Status
QSP-1	Draining Collateral with Reentrancy Attack	High	Fixed
QSP-2	Broken Slippage Controls Will Result in Lost Funds	High	Mitigated
QSP-3	Incorrect Price of `STETH` May Be Used From Chainlink Feed	Medium	Mitigated
QSP-4	Balancer LP Token Price May Be Incorrect	Medium	Acknowledged
QSP-5	`STETH` Price Feed Is a Single Point of Failure	Low	Acknowledged
QSP-6	Strict Inequality Could Cause Validation to Fail	Low	Fixed
QSP-7	Extreme Fee Setup Can Block the Yield Function	Low	Fixed
QSP-8	Unlimited Allowance Given to Vault	Low	Fixed
QSP-9	Hardcoded Slippage May Result in Swap Failure	Low	Fixed
QSP-10	Missing Input Validation	Low	Fixed
QSP-11	Precision Loss Due to Division Before Multiplication	Low	Fixed
QSP-12	Return Values Not Verified	Low	Fixed
QSP-13	Unsafe Casts	Low	Acknowledged
QSP-14	Reentrancy Risks Can Be Mitigated	Low	Acknowledged
QSP-15	Incorrect `_vaultFee` Could Cause `_transferYield()` to Fail	Low	Fixed
QSP-16	Privileged Roles and Ownership	Informational	Acknowledged
QSP-17	Events Not Emitted at Key State Transitions	Informational	Acknowledged
QSP-18	Functions Unnecessarily Marked as `payable`	Informational	Acknowledged
QSP-19	Directly Transferred Tokens Are Swapped	Informational	Acknowledged
QSP-20	Leverage Parameters Are Unconventional or Unintuitive	Informational	Acknowledged
QSP-21	Code Readability Suffers From Misleading Names	Informational	Mitigated
QSP-22	Reliance on External Contracts to Secure Funds	Informational	Acknowledged
QSP-23	Unlocked Pragma	Informational	Acknowledged
QSP-24	`enterPositionWithFlashloan()` Assumes Maximum Slippage	Undetermined	Acknowledged
QSP-25	`vaultYieldInPrice()` Returns Incorrect Value for All Vaults	Undetermined	Fixed
QSP-26	`getYieldAmount()` Returns Incorrect Value for `AuraBalancerLPVault`	Undetermined	Fixed
QSP-27	Allowing More than 10x Leverage	Undetermined	Fixed
QSP-28	Usage of `BALWSTETHWETHOracle` Is Unclear	Undetermined	Fixed
QSP-29	Swap Slippage Is Amplified by Leverage	Undetermined	Acknowledged

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

DISCLAIMER:
If the final commit hash provided by the client contains features that are not within the scope of the audit or an associated fix review, those features are excluded from consideration in this report. Please note that this audit only covers the following contracts:

- contracts/misc/BALWSTETHWETHOracle.sol
- contracts/protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol
- contracts/protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol
- contracts/protocol/leverage/LeverageSwapManager.sol
- contracts/protocol/leverage/GeneralLevSwap.sol

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.9.1

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither on each individual file:
3. `slither ./contracts/protocol/leverage/GeneralLevSwap.sol`
4. `slither ./contracts/protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`
5. `slither ./contracts/protocol/leverage/LeverageSwapManager.sol`
6. `slither ./contracts/misc/BALWSTETHWETHOracle.sol`
7. `slither ./contracts/protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Findings

QSP-1 Draining Collateral with Reentrancy Attack

Severity: *High Risk*

Status: Fixed

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: In this audit, we found an exploit path in which the attacker can drain the collateral of other users. The `GeneralLevSwap` contract does not use the `ReentrancyGuard` (for example, OpenZeppelin's contract: link) nor follows the checks-effects-interactions pattern. Thus, the token transfer calls might trigger reentrancy attacks. However, most of the time, the code only interacts with allow-listed tokens. The tokens are either from the Sturdy platform itself or integrated platforms, such as Aave and Balancer. Those tokens do not have the hook to trigger a reentrancy attack.
Unfortunately, in the `GeneralLevSwap._withdrawWithFlashloan()` function, the line `IERC20(_sAsset).safeTransferFrom(_user, address(this), withdrawalAmount)` can be a valid entry point of a reentrancy attack. The reason is that `_sAsset` is from user input when calling the `GeneralLevSwap.withdrawWithFlashloan()` function. Therefore, the attacker can pass in a malicious `_sAsset` contract address and trigger the attack.
Once triggered, the attacker can call the `IBalancerVault(BALANCER_VAULT).flashLoan()` with arbitrary `userData` (see: balancer's flashLoan code). This allows the Balancer to call the `GeneralLevSwap.receiveFlashLoan()` function with unexpected data. The `_balancerFlashLoanLock == 2` validation on `L131` will also be bypassed since it is inside a reentrancy call. Finally, the transaction calls the `GeneralLevSwap._withdrawWithFlashloan()` function with arbitrary inputs to redeem the aTokens (`_sAsset`) of the victims.

Exploit Scenario:

1. The attacker creates a malicious `_sAsset` contract. It mimics all necessary interfaces but triggers a reentrancy attack inside the `safeTransferFrom()` function.
2. The attacker calls `GeneralLevSwap._withdrawWithFlashloan()` with the `_sAsset` being the malicious contract.
3. During the reentrancy, the malicious contract calls `IBalancerVault(BALANCER_VAULT).flashLoan()` with arbitrary `userData`, allowing hooking back to the `GeneralLevSwap.receiveFlashLoan()` with malicious inputs.
4. The `receiveFlashLoan()` function calls `_withdrawWithFlashloan()` eventually for the victim. It redeems the victim's collateral back to borrow the token.
5. After the reentrancy and the `flashLoan()` calls are done, the stack pops back to the `GeneralLevSwap.withdrawWithFlashloan()`. The code will directly transfer the stolen token to the attacker or supply the tokens back to the vault on behalf of the attacker. The attacker can easily withdraw the tokens even if it supplies back with the standard withdrawal procedure.

Recommendation: We determined that the issue is due to insufficient input validation of the `_sAsset` and a lack of reentrancy protection. Consequently, we suggest the following changes:

1. Tighten the validation for the `_sAssset` of the `GeneralLevSwap.withdrawWithFlashloan()` function. The `_sAsset` must be the aToken for the `GeneralLevSwap.COLLATERAL`. Alternatively, consider removing the `_sAsset` parameter from the function and retrieving the correct address by querying an appropriate function on one of the protocol's contracts.
2. Due to the complexity of the functions, it is hard to follow the checks-effects-interactions pattern strictly. We recommend to add reentrancy guard for the `enterPositionWithFlashloan()`, `withdrawWithFlashloan()`, `zapDeposit()`, and `zapLeverageWithFlashloan()` functions. The above functions can share the same guard. We suggest further guarding the `executeOperation()` and `receiveFlashLoan()` functions. However, they will need a different state variable from the other sets of functions, as these two functions are part of the other function calls.
3. Consider changing the value of `_balancerFlashLoanLock` back to `1` immediately after the `require(_balancerFlashLoanLock == 2, Errors.LS_INVALID_CONFIGURATION);` check to further protect against reentrancy of the `executeOperation()` function.
4. In the `GeneralLevSwap.withdrawWithFlashloan()` function, if it uses the balancer flash loan, add a validation that `require(_balancerFlashLoanLock == 1, ...)` on `L245-249`.

Update: The issue has been fixed through the implementation of all the recommendations.

QSP-2 Broken Slippage Controls Will Result in Lost Funds

Severity: *High Risk*

Status: Mitigated

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: The functions `withdrawWithFlashloan()`, `zapDeposit()`, and `zapLeverageWithFlashloan()` within the `GeneralLevSwap` contract have ineffective slippage mechanisms and permit swaps to be made at very undesirable rates. This leaves users vulnerable to sandwich attacks and front-running. In the worst scenario, it could result in users losing far more collateral tokens than expected when repaying a loan through `GeneralLevSwap`. The slippage control of each function is ineffective in a different way.

1. withdrawWithFlashloan()

While the function does accept a `_slippage` parameter, the only time this parameter is used is when withdrawing collateral from the vault contract: `IGeneralVault(VAULT).withdrawCollateral(COLLATERAL, _amount, _slippage, address(this));`. This will ensure that the amount withdrawn from the vault is within the `_slippage` tolerance of `_amount`. However, this does nothing to protect against severe slippage when swapping from the collateral asset to the borrowing asset and vice versa. In particular, severe slippage may occur in the following two swaps:

- `L328:_swapFrom(_borrowingAsset);`
  . This call swaps all the user's withdrawn collateral asset tokens for borrowing asset tokens. The amount of collateral withdrawn on behalf of the user is the maximum amount a user can withdraw while remaining in a healthy (non-liquidatable position).

  . If the entire debt was repaid on `L292:_repay(_borrowingAsset, _borrowedAmount, _user);`, then all the collateral tokens of the user are swapped. If the slippage is severe enough that the swap only produces enough borrowing asset tokens to repay the flash loan and swap back to the `_requiredAmount`, the user will lose all their collateral.

  . In the current implementation of `AURAWSTETHWETHLevSwap`, we note that the `_swapFrom()` function has a hardcoded slippage of `10%`. While this limits the impact to some extent, 10% is a severe amount of slippage, and the loss suffered by the user could be much greater than `10%` as seen in the exploit scenario.

- `L253:_swapTo(_borrowingAsset, IERC20(_borrowingAsset).balanceOf(**address**(this)));`
  . This call swaps whatever borrowing asset is left after repaying the flash loan into collateral asset tokens. If `_swapFrom()` suffered from severe slippage, then this swap may have negative slippage. However, the damage is already done, as seen in the exploit scenario section.

  . Unlike `_swapFrom()`, the function does not even have the hardcoded slippage control of `10%` within the `AURAWSTETHWETHLevSwap` contract.

2. zapDeposit()

- The `zapDeposit()` function swaps the borrowable asset for the collateral asset. However, no slippage parameters are available for the function, meaning that any

- amount of slippage is possible. This could result in the user's zapped `_principal` losing most of its value.

3. zapLeverageWithFlashloan()

    - The `zapLeverageWithFlashloan()` function does have a `_slippage` parameter. However, this parameter has no impact on the first swap it makes: `L444:uint256 collateralAmount = _swapTo(_zappingAsset, _principal);`. Thus the function suffers from the same problem as `zapDeposit()` above.

Exploit Scenario:

1. Alice has collateral worth `$1000`. For simplicity, we assume a one-to-one collateral ratio. Alice also has a debt worth `$800`. We will also assume a flash loan fee of zero.
2. Alice could withdraw `$200` worth of collateral tokens since she is over-collateralized. We could say her position is worth `$200` at the moment.
3. Alice decides to call `withdrawWithFlashloan()` to repay her full debt and sets the `_requiredAmount` parameter to `$100` worth of collateral tokens.
    1. A flash loan of `$800` is taken to pay off the debt.
    2. Then the `$1000` worth of the collateral asset is withdrawn on behalf of Alice and is swapped with `10%` slippage to `$900` worth of borrowing asset tokens.
    3. The flash loan is repaid, leaving `$100` worth of borrowing asset tokens.
    4. The `$100` worth of borrowing asset is swapped back to `$110` worth of the collateral asset. We have assumed `10%` negative slippage here.
    5. `$100` worth of the collateral asset is sent to Alice, and `$10` worth of the collateral asset is redeposited into the vault on behalf of Alice. Since she has no debt, she could withdraw the `$10` worth of the collateral asset if she wished to. Alice's position plus her withdrawn collateral is worth `$110`.
    6. Alice's position is now worth only `55%` of what it was before.

Recommendation: Implement slippage parameters and checks for all three functions such that the user can control the maximum amount by which their position can be impacted by slippage.

Update: Controls have been implemented to bound the slippage on each swap. However, some functions perform more than one swap, so the net slippage (value lost) of the function call can still be greater than the `_slippage` parameter provided by the user. The user will need internal knowledge of the function to understand the amount of value lost through the leveraged functions. Additionally, while `zapDeposit()` now has a `_slippage` parameter, there is no input validation for `_slippage > 0`.

QSP-3 Incorrect Price of `STETH` May Be Used From Chainlink Feed

Severity: *Medium Risk*

Status: Mitigated

File(s) affected: `misc/BALWSTETHWETHOracle.sol`

Description: The `BALWSTETHWETHOracle.sol` contract retrieves the price of `STETH` by querying a Chainlink price feed: `(, int256 wstETHPrice, , , ) = WSTETH.latestRoundData();`. However, several unchecked assumptions are made about the price, including its staleness, validity, and the number of decimals it has. The consequences of an incorrect price could be severe for the protocol, including the loss of user funds. The following should be done to ensure a valid price:

1. Validate that the `updatedAt` value returned by `WSTETH.latestRoundData()` indicates a timestamp that is not too old.
2. Validate that the price returned by `WSTETH.latestRoundData()` is greater than zero.
3. Rather than assuming that the price has `18` decimals, explicitly check the number of decimals from the price feed through the `decimals()` function.

Recommendation: Implement the validation checks mentioned above and explicitly check the decimals of the price through the price feed contract.

Update: Although most of the recommendations have been implemented, there is still an implicit assumption that the Chainlink feed returns the price with `18` decimals. Rather than making this assumption, the `IChainlinkAggregator.decimals()` function should be used.

QSP-4 Balancer LP Token Price May Be Incorrect

Severity: *Medium Risk*

Status: Acknowledged

File(s) affected: `misc/BALWSTETHWETHOracle.sol`

Description: The approach taken to calculate the Balancer LP Token value involves multiplying the price of the cheaper of the two tokens (`WETH` and `STETH`) by the value returned by `BALWSTETHWETH.getRate()`. This price calculation would be most accurate when the price of `WETH` is the same as `STETH`. However, the `WETH/STETH` price can fluctuate significantly, and the further they diverge, the less accurate the price calculation will be. While the calculation provides a reliable lower bound for the price, it is unclear how different this lower bound will be from the actual price. If the price calculation is too low, it could result in premature liquidation of users who have deposited the Balancer LP token as collateral.

Recommendation:

1. Carefully model just how inaccurate the price calculation will be for likely WETH/STETH price divergence scenarios.
2. Collateral ratios for the Balancer LP token should also be set appropriately.
3. Consider implementing safety mechanisms to prevent liquidations or other activity related to the Balancer LP token once the `WETH/STETH` price diverges too much.
4. Consider finding an alternative pricing calculation that is more resistant to price divergence.

Update: The Sturdy team acknowledged the issue with a message stating that they are using the method in this Chainlink blog to calculate LP token prices. However, the auditing team has several concerns with this:

1. The LP token price calculation demonstrated in the Chainlink blog is for LP tokens of a Curve pool consisting of various stablecoins whose value is equal to each other. However, in the case of the Balancer pool, the tokens are not one-to-one in terms of price. The calculation mentioned in the blog will be less effective if the price of tokens within the pool differs too much.
2. It is important to note that the approach described in the Chainlink blog is for calculating a lower bound of the price rather than the exact price. However, with a script we have provided to the Sturdy team, we have shown that the calculation in the current code fails to provide a lower bound. The result of the calculation is often larger than the actual price of the LP token. This indicates that something is not right with the current calculation.
3. As mentioned before, the Chainlink blog describes a method of calculating the lower bound for the LP token price. It is unclear how much this lower bound would differ from the actual token price. If the lower bound differs too much, it could result in premature liquidations, as described in the issue.

Update after final fix review: The Sturdy team consulted with the Balancer team regarding calculating the LP token price. The Balancer team recommended the formula `Math.min((uint256(stETHPrice), 1e18)` which the Sturdy team has implemented in commit `6098b62f65e4b8504bee06001dd0844a70309aaf`. So far, we have not found anything that contradicts the idea that this formula provides a lower bound for the LP token price, but it is still unclear just how close to the real price this lower bound is.
It is important to note that the Balancer team based their recommendation on the price of ETH being one-to-one with STETH. However, the price of STETH is often different than ETH, and we suspect that the current price calculation will grow more inaccurate with higher differences. We recommend that the Sturdy team consider this possibility and the potential consequences to the protocol.

QSP-5 `STETH` Price Feed Is a Single Point of Failure

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `misc/BALWSTETHWETHOracle.sol`

Description: The `BALWSTETHWETHOracle` contract's primary purpose is to provide a price for the `B-stETH-STABLE` token. However, the correct calculation of the price is highly dependent on having the correct `STETH` price. Currently, the only source of this data is the Chainlink price feed. In the event of any large-scale attack/disruption of the Chainlink network, the protocol could be severely impacted.

Recommendation: Consider adding at least one other robust price feed independent of Chainlink. Furthermore, consider implementing a pause mechanism for when the price feed fails.

Update: The team believes that the Chainlink price should be reliable enough.

QSP-6 Strict Inequality Could Cause Validation to Fail

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: A validation check within `_withdrawWithFlashloan()` incorrectly uses strict inequality, which may cause the function to fail for valid data: `require(withdrawalAmount > _requiredAmount, Errors.LS_SUPPLY_NOT_ALLOWED);`. If the `withdrawalAmount` equals the `_requiredAmount`, the validation check will fail, despite meeting the user's requirements.

Recommendation: Do not use strict inequality for the aforementioned validation check.

Update: The strict inequality has been replaced with a non-strict inequality.

QSP-7 Extreme Fee Setup Can Block the Yield Function

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/vault/GeneralVault.sol`, `protocol/vault/ethereum/AuraBalancerLPVault.sol`, `protocol/libraries/PercentageMath.sol`

Description: In the case where `_vaultFee + _incentiveRatio == PercentageMath.PERCENTAGE_FACTOR` in the `AuraBalancerLPVault` contract, the `AuraBalancerLPVault._transferYield()` function might fail due to overflow. The reason is that the `PercentageMath.percentMul()` function rounds up when the decimal is larger than or equal to `0.5`. Consequently, in the case `_vaultFee + _incentiveRatio == PercentageMath.PERCENTAGE_FACTOR`, the `incentiveAmount` and `treasuryAmount` might be larger than the `yieldAmount`. The transfer can fail unexpectedly,

Exploit Scenario:

1. We assume that the `_vaultFee` represents `30%` and `_incentiveRatio` represents `70%`.
2. When the `yieldAmount` is `95` in the `_transferYield()` function, the `treasuryAmount` will be `29 (95 * 0.3 = 28.5 -> 29)` and the `incentiveAmount` will be `67 (95 * 0.7 = 66.5 -> 67)`.
3. On `L101`, it will send `67` tokens away during the `_sendIncentive()` call, leaving the current contract with a balance of `95-67 = 28`.
4. On `L108`, the line `IERC20(_asset).safeTransfer(_treasuryAddress, treasuryAmount)` will fail as the current address only have `28` tokens left after the incentive transfer, but the code tries to transfer `29` tokens.

Recommendation:

1. In the `AuraBalancerLPVault.setIncentiveRatio()` function, change the validation to `require(_vaultFee + _ratio < PercentageMath.PERCENTAGE_FACTOR, Errors.VT_FEE_TOO_BIG)`. (From <= to <)
2. Override the `GeneralVault.setTreasuryInfo()` in the `AuraBalancerLPVault` contract to add the valuation and then call `super.setTreasuryInfo()`.

Update: The issue has been fixed through the implementation of all the recommendations.

QSP-8 Unlimited Allowance Given to Vault

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: In the constructor of the `GeneralLevSwap` contract, the `_vault` is set to have a maximum allowance of the `COLLATERAL`. While the `GeneralLevSwap` contract is not designed to hold any collateral assets directly, setting an unlimited allowance is a pattern that should generally be avoided.

Recommendation: Remove the unlimited allowance and always set the appropriate allowance before vault interactions that are supposed to sweep off collateral funds.

Update: The infinite approval has been replaced with an approval for only the required amount.

QSP-9 Hardcoded Slippage May Result in Swap Failure

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`

Description: When swapping from the Balancer LP token to the borrowing asset through the `_swapFrom()` function, there is a hard-coded slippage value of `10%` as seen in `L93:uint256 minAmountOut = ((_amountToSwap * fromAssetPrice) / 1e18).percentMul(9000); //10% slippage`. If a user ever wanted to allow more than `10%` slippage due to unusual market conditions, it would not be possible. In these cases, any attempt to swap through `_swapFrom()` would fail.

Recommendation: Remove the hardcoded slippage limit. The slippage limit should be determined by the user based on the losses they are willing to tolerate.

Update: Maximum swap slippage is now determined by the user through a new parameter.

QSP-10 Missing Input Validation

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/leverage/LeverageSwapManager.sol`, `protocol/leverage/GeneralLevSwap.sol`, `protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`

Description: Several functions within the protocol lack sufficient validation of their input parameters. We have listed all missing checks below:

1. `GeneralLevSwap`
   1. `executeOperation()`
      1. Validate that the array parameters have the same length.
      2. Validate that `amounts[0]` is greater than zero.
      3. Validate that `assets[0]` is not the zero address.
   2. `receiveFlashLoan()`
      1. Validate that the array parameters have the same length.
      2. Validate that `amounts[0]` is greater than zero.
      3. Validate that `tokens[0]` is not the zero address.
   3. `_executeOperation()`
      1. Validate that `arg1` is greater than zero.
      2. Validate that `arg2` is not the zero address.
      3. If `isEnterPosition` is `false`, then validate that `arg0` is greater than zero.
      4. If `isEnterPosition` is `false`, then validate that `arg3` is not the zero address.
   4. `enterPositionWithFlashloan()`
      1. Validate that `_borrowingAsset` is not the zero address.
   5. `withdrawWithFlashloan()`
      1. Validate that `_borrowingAsset` is not the zero address.
   6. `zapLeverageWithFlashloan()`
      1. Validate that `_borrowingAsset` is not the zero address.
      2. Validate that `_zappingAsset` is not the zero address.
   7. `zapDeposit()`
      1. Validate that `_zappingAsset` is not the zero address.
2. `LeverageSwapManager`
   1. `initialize()`
      1. Validate that `_provider` is not the zero address.
   2. `registerLevSwapper()`
      1. Validate that `collateral` is not the zero address.
3. `AURAWSTETHWETHLevSwap`
   1. `_swapTo()`
      1. Validate that the `_borrowingAsset` is WETH.
   2. `_swapFrom()`
      1. Validate that the `_borrowingAsset` is WETH.

Recommendation: Implement the recommended validation checks.

Update: All recommended validation checks have been added.

QSP-11 Precision Loss Due to Division Before Multiplication

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: There are two calculations in which precision is lost due to division before multiplication. The first is the calculation for the withdrawal amounts from `` `_withdrawWithFlashloan() `` below:

```
uint256 withdrawalAmountETH = (((totalCollateralETH * currentLiquidationThreshold) /
  PercentageMath.PERCENTAGE_FACTOR -
  totalDebtETH) * PercentageMath.PERCENTAGE_FACTOR) / assetLiquidationThreshold;

uint256 withdrawalAmount = Math.min(
  IERC20(_sAsset).balanceOf(_user),
  (withdrawalAmountETH * (10**DECIMALS)) / _getAssetPrice(COLLATERAL)
);
```

In the calculation for `withdrawalAmountETH` , a number is divided by `PercentageMath.PERCENTAGE_FACTOR` and then multiplied by another number. `withdrawalAmountETH` is also first divided by `assetLiquidationThreshold` and then multiplied by another number.
The second example is the calculation for the amount to borrow with the flash loan in `_leverageWithFlashloan()`:

```
amounts[0] = ((((_principal * _getAssetPrice(COLLATERAL)) / 10**DECIMALS) *
  10**borrowAssetDecimals) / _getAssetPrice(_borrowAsset)).percentMul(_leverage).percentMul(
    PercentageMath.PERCENTAGE_FACTOR + _slippage
  );
```

Above, a number is divided by `10**DECIMALS` and then multiplied by another number.

Recommendation: Multiply before dividing to avoid loss in precision.

Update: The team acknowledged the issue with the following statement:

```
The calculation is based on asset and price decimals. So mathematically, division before multiplication would not result in precision loss.
```

However, the following is an example that shows precision loss within the `_leverageWithFlashloan()` function:

```
(
  (
    (
      (_principal * _getAssetPrice(COLLATERAL)) / 10**DECIMALS
    ) * 10**borrowAssetDecimals
  )
  / _getAssetPrice(_borrowAsset)
)
```

1. Assume that both `DECIMALS` and `borrowAssetDecimals` are `1` for simplicity, Assume that `_principal` is `1`, `_getAssetPrice(COLLATERAL)` is `11`, and `_getAssetPrice(_borrowAsset)` is `11`.
2. The above formula with the division before multiplication will be:

```
(
    (
        (
            (1 * 11) / 10
        ) * 10
    )
    / 11
) = 0
```

However, if we do multiplication beforehand:

```
(
  (
    (
      (_principal * _getAssetPrice(COLLATERAL)) * 10**borrowAssetDecimals
    ) / 10**DECIMALS
  )
  / _getAssetPrice(_borrowAsset)
)
```

This brings us `1` instead of `0`:

```
(
    (
        (
            (1 * 11) * 10
        ) / 10
    )
    / 11
) = 1
```

Update after final fix review: The Sturdy team has fixed the issue by following the recommendations in commit `dc04bd3b9a70a2debd19b7d29f9c017839d75db0`.

QSP-12 Return Values Not Verified

Severity: Low Risk

Status: Fixed

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`, `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Description: Several contracts within the protocol make calls to functions on other contracts with return values that indicate the result of those functions. However, many of these return values are not being validated within the code. We have listed all function calls for which return values are being ignored:

1. `GeneralLevSwap`
   1. `L83:IERC20(COLLATERAL).approve(_vault, type(**uint256**).max);`
   2. `L367:LENDING_POOL.repay(_borrowingAsset, _amount, USE_VARIABLE_DEBT, borrower);`
2. `AuraBalancerLPVault`
   1. `L197:AURA_BOOSTER.deposit(auraPoolId, _amount, true);`
   2. `L202:SturdyInternalAsset(internalAsset).mint(address(this), _amount);`
   3. `` `L236:IConvexBaseRewardPool(baseRewardPool).withdrawAndUnwrap(_amount, false); ``

Recommendation: Validate that the return parameters of the above function calls are the expected values.

Update: The issue has been resolved by verifying the return values of all cases mentioned above.

QSP-13 Unsafe Casts

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `misc/BALWSTETHWETHOracle.sol`, `protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`

Description: There are multiple instances of unsafe casts within the contracts. These could lead to overflow or underflow of values and have unpredictable results. We have listed all instances of unsafe casts below:

1. ` BALWSTETHWETHOracle``
   1. `` `L25:uint256 minValue = Math.min(**uint256**(wstETHPrice), 1e18); ``
   2. `L39:return (true, int256(_get()));`
   3. `` `L45:return int256(_get()); ``
2. AURAWSTETHWETHLevSwap`
   1. `L45:uint256 joinKind = **uint256**(IBalancerVault.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT);`
   2. `L75:uint256 exitKind = **uint256**(IBalancerVault.ExitKind.EXACT_BPT_IN_FOR_ONE_TOKEN_OUT);`

Recommendation: Rather than using direct casts such as `uin256t()` and `int256()`, use the equivalent functions from OpenZeppelin's SafeCast library.

Update: The Sturdy team responded with the following message:

```
Price is always a positive value, so casting is safe.
```

The following is our analysis:

- 1.1: Since the price is positive, the casting from `int256` to `uint256` is safe.
- 1.2 and 1.3: the risk is really low despite being non-zero. To overflow, the price needs to be an extreme value.
- 2.1 and 2.2: The casting from hardcoded `enum` to `uint256` is safe here.

QSP-14 Reentrancy Risks Can Be Mitigated

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Description: To avoid reentrancy risks and reduce the attack surface, it is highly recommended to inherit from OpenZeppelin's `ReentrancyGuard` contract and add the `nonReentrant` modifier to all functions that:

1. Involve asset manipulation.
2. Could be accessed by any external address.
3. Are not expected to be reentered

The following is a non-exhaustive list of functions we recommend adding a `nonReentrant` modifier to:

- `GeneralVault/AuraBalancerLPVault.sol`
  - `depositCollateral()`
  - `depositCollateralFrom()`
  - `withdrawCollateral()`
  - `processYield()`

Recommendation: Add a `nonReentrant` modifier to all suggested functions.

Update: The Sturdy team mentioned that they could not add the `nonReentrant` modifier due to a potential storage collision. However, given the code complexity, the team should be very careful when adding an additional token to the protocol.

QSP-15 Incorrect `_vaultFee` Could Cause `_transferYield()` to Fail

Severity: *Low Risk*

Status: Fixed

File(s) affected: `protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`

Description: The `AuraBalancerLPVault` contract introduces another fee parameter, namely `_incentiveRatio`. It is set by the function `setIncentiveRatio()`, which properly makes sure that the sum of `_vaultFee` and `_incentiveRatio` is less than `100%`. However, while the function `GeneralVault.setTreasuryInfo()`, which sets the treasury address and the `_vaultFee`, checks that the `_vaultFee` is less than or equal to `30%`, it does not perform a sanity check that `_vaultFee` and `_incentiveRatio` (which is only introduced in `AuraBalancerLPVault.sol`) is less than `100%`. So, the contract could be erroneously set up where first the `_incentiveRatio` is set to `71%` via `setIncentiveRatio()`, followed by a setting of the `_vaultFee` of `30%` via `setTreasuryInfo()`, leading to `_incentiveRatio` + `_vaultFee` > `100%`. This would cause the `_transferYield()` function to revert, leading to the yield being stuck in the contract, as more than `100%` of the contract balance is attempted to be transferred.

Recommendation: Overwrite `setTreasuryInfo()` in the `AuraBalancerLPVault` contract to additionally check that the `_vaultFee` summed with `_incentiveFee` will be less than `100%`.

Update: The recommended override and check have been implemented.

QSP-16 Privileged Roles and Ownership

Severity: *Informational*

Status: Acknowledged

File(s) affected: `protocol/leverage/LeverageSwapManager.sol`

Description: The `LeverageSwapManager` possesses a function `registerLevSwapper()` which is callable by the administrator role of the contract. This function allows the admin to change the official implementation of the `*LevSwap` contract for a particular collateral to an entirely different one. They may also disable the leveraged swap functionality for the collateral by calling `registerLevSwapper()` with the zero address for the `swapper` parameter. Users should be made aware of the consequences of this privileged role and its consequences. The team should also warn users in advance if the leveraged swapped functionality is being disabled or changed for a collateral asset.

Recommendation: Update the user-facing documentation so that users are aware of the consequences of the admin role within the `LeverageSwapManager` contract.

Update: The Sturdy team will update their documentation so users are aware of the mentioned privileged roles.

QSP-17 Events Not Emitted at Key State Transitions

Severity: *Informational*

Status: Acknowledged

File(s) affected: `protocol/leverage/LeverageSwapManager.sol`

Description: The `registerLevSwapper()` function represents a critical state change for the protocol, as it changes or disables the leverage swap functionality for a particular collateral asset. The function should emit an event, so this key state transition can be monitored and responded to.

Recommendation: Make `registerLevSwapper()` emit an appropriate event.

Update: The team stated that the function is just used to check the enabled/disabled status of the leverage feature.

QSP-18 Functions Unnecessarily Marked as `payable`

Severity: *Informational*

Status: Acknowledged

File(s) affected: `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Description: The functions `setConfiguration()` and `processExtraYield()` in the `AuraBalancerLPVault` contract are marked as `payable`, yet we were unable to find a reason for why that should be the case. It seems like there would be no use for ETH transferred via these methods and the funds would become stuck.

Recommendation: Unless there is a clear reason for it, remove the `payable` keyword from the mentioned function signatures.

Update: The functions have been marked as `payable` to save on gas costs. We discourage such behavior in order to avoid accidentally locking ETH into the contract.

QSP-19 Directly Transferred Tokens Are Swapped

Severity: *Informational*

Status: Acknowledged

Description: As the `_swapTo()` and `_swapFrom()` functions in the `AURAWSTETHWETHLevSwap` contract return the whole contract balance of Balancer LP Tokens and WETH, respectively, such tokens, which were sent directly to the contract by accident, will be returned to the next user swapping to that asset. So, under such conditions, users will unexpectedly receive more tokens than perhaps anticipated from swaps. This of course does not seem problematic but could lead to unexpected behavior. The inflated amount of collateral would be used to create leverage, possibly resulting in a much larger position than desired. Of course, a user could immediately withdraw again and make a profit, but the user may also be confused by the protocol's behavior.

Recommendation: Clarify if this is desired behavior or not. If not, simply returning the difference in the asset balance before and after adding or removing liquidity to the pool should suffice.

Update: The Sturdy team indicated that this is the desired behavior.

QSP-20 Leverage Parameters Are Unconventional or Unintuitive

Severity: *Informational*

Status: Acknowledged

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: The definition of leverage the protocol uses might differ from a user's expectations. For example, most users would expect when using "2X" leverage that an additional 100% of the provided collateral would be borrowed. Yet in this protocol, a leverage of `2` will result in an additional `200%` being borrowed, which for some users might lead to unexpectedly creating higher leverage than desired. While the used definition differs, the documentation properly describes how the leverage is set and calculated in the protocol.

Recommendation: Confirm that the current leverage definition is desired. If not, adjust it accordingly.

Update: The Sturdy team indicated that this is the desired behavior.

QSP-21 Code Readability Suffers From Misleading Names

Severity: *Informational*

Status: Mitigated

File(s) affected: `protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`

Description: The function signatures from `_swapFrom()` and `_swapTo()` seem to suggest that `_amount` of `_borrowingAsset` is swapped to another asset or being swapped for, respectively. However, in the derived implementation in the `AURAWSTETHWETHLevSwap` contract, the `_borrowingAsset` parameter remains unused in both methods. The `swapTo()` function always swaps WETH to Balancer LP Token by adding them to the pool and the `swapTo()` function swaps the Balancer LP Token to WETH by exiting the pool. The same issue can be spotted in the other contract derived from the `GeneralLevSwap` contract, namely the `ETHSTETHLevSwap` contract.
Also, the implementation of the function `getAvailableStableCoins()` in the `AURAWSTETHWETHLevSwap` contract returns the WETH-address, which of course is not a stablecoin.

Recommendation: In case no future child contracts are planned that make use of the parameter, consider removing the unnecessary `_borrowingAsset` parameter and perhaps improve the naming of the functions.

Update: Some function names have been improved. However, the `_borrowingAsset` parameter remains in the signature of the `_swapFrom()` and `_swapTo()` functions.

QSP-22 Reliance on External Contracts to Secure Funds

Severity: *Informational*

Status: Acknowledged

File(s) affected: `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Description: The platform generates yields by sending the collateral tokens stored by borrowers to other protocols such as Aura. If any of these yield-generating protocols got hacked, the protocol users would also lose their funds.

Recommendation: Make this warning explicit to the protocol's users by adding this risk statement to the protocol's public-facing documentation.

Update: The Sturdy team will add a risk statement to the protocol's documentation.

QSP-23 Unlocked Pragma

Severity: *Informational*

Status: Acknowledged

File(s) affected: `misc/BALWSTETHWETHOracle.sol`, `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`, `protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol`, `protocol/leverage/LeverageSwapManager.sol`, `protocol/leverage/GeneralLevSwap.sol`

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity ^0.8.0`. The `^` before the version number implies an unlocked pragma, meaning that the compiler will use the specified version and above, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend removing the `^` to lock the file onto a specific Solidity version.

Update: The Sturdy team indicated that they always use the same Solidity version to compile contracts. We still recommend fixing the version as added protection.

QSP-24 `enterPositionWithFlashloan()` Assumes Maximum Slippage

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: The function `enterPositionWithFlashloan()` assumes the maximum amount of slippage when calculating the amount to borrow for the flash loan. This can be seen in the code below from the helper function `_leverageWithFlashloan()`

```
amounts[0] = ((((_principal * _getAssetPrice(COLLATERAL)) / 10**DECIMALS) *
  10**borrowAssetDecimals) / _getAssetPrice(_borrowAsset)).percentMul(_leverage).percentMul(
    PercentageMath.PERCENTAGE_FACTOR + _slippage
  );
```

This borrowed amount is then completely swapped to collateral tokens and deposited on behalf of the user as shown in the code below from `_enterPositionWithFlashloan()`:

```
//swap borrowing asset to collateral
uint256 collateralAmount = _swapTo(_borrowingAsset, _borrowedAmount);
require(collateralAmount >= _minAmount, Errors.LS_SUPPLY_FAILED);

//deposit collateral
_supply(collateralAmount, _user);

//borrow borrowing asset
_borrow(_borrowingAsset, _borrowedAmount + _fee, _user);
```

A user may expect that they will only deposit `_principal + _principal*_leverage` amount of collateral if the swap slippage is zero. They may be surprised to see that they have a more leveraged position than they expected. The `zapLeverageWithFlashloan()` suffers from the same issue.

Recommendation: If this is the desired implementation, users should be made aware of the consequences of the `_slippage` parameter through clear user-facing documentation.

Update: The Sturdy team indicated that this is the desired behavior, and they will add documentation to make this clearer to users.

QSP-25 `vaultYieldInPrice()` Returns Incorrect Value for All Vaults

Severity: *Undetermined*

Status: Fixed

File(s) affected: `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Description: The `GeneralVault.vaultYieldInPrice()` function always returns `0` and is not overridden by any vault contracts. As such, this function always returns the incorrect value. This can be seen in the function definition below:

```
function vaultYieldInPrice() external view virtual returns (uint256) {
  return 0;
}
```

Recommendation: The `GeneralVault.vaultYieldInPrice()` function should either be removed or implemented to always return the correct value.

Update: The `vaultYieldInPrice()` function has been removed.

QSP-26 `getYieldAmount()` Returns Incorrect Value for `AuraBalancerLPVault`

Severity: *Undetermined*

Status: Fixed

File(s) affected: `protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol`

Description: The `AuraBalancerLPVault.getYieldAmount()` function should return the amount of pending yield for the vault. However, it is currently returning the result of the `GeneralVault._getYieldAmount()` function, which returns the difference between the `ATokenForCollateral` supply and the amount of underlying token held within the

`ATokenForCollateral` contract. This calculation does not apply to the `AuraBalancerLPVault`, which retrieves yield based on how much the AURA pool awards it.

Recommendation: Reimplement `AuraBalancerLPVault.getYieldAmount()` to return the correct pending yield.

Update: The `getYieldAmount()` function has been removed.

QSP-27 Allowing More than 10x Leverage

Severity: Undetermined

Status: Fixed

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: In the `GeneralLevSwap` contract, the `zapLeverageWithFlashloan()` and `enterPositionWithFlashloan()` functions does not validate the `_leverage` input to be less than or equal to nine. If the user calls the functions with `_leverage > 9`, it will have the effect of having a leverage `> 10x` of the user's principle. This breaks the spec from the Sturdy documentation, which is states that "DeFi's best yield farmers use Sturdy to access up to 10x leverage to farm with".
Note that due to the liquidation threshold, the leverage is likely to fail as one cannot borrow such that their position is liquidatable.

Recommendation: We recommend adding validation to the `_leverage` input on both the `zapLeverageWithFlashloan()` and the `enterPositionWithFlashloan()` function.

Update: Validation has been added so that leverage cannot exceed `10x`.

QSP-28 Usage of `BALWSTETHWETHOracle` Is Unclear

Severity: Undetermined

Status: Fixed

File(s) affected: `misc/BALWSTETHWETHOracle.sol`

Description: It is unclear how exactly `BALWSTETHWETHOracle.sol` will be integrated with the remainder of the system. The `AURAWSTETHWETHLevSwap` contract inherits from the `GeneralLevSwap` contract, which internally uses `SturdyOracle` contract as its `ORACLE`. The `SturdyOracle` contract seems to be designed to make direct calls to chainlink oracles, but their function signatures differ compared to the `BALWSTETHWETHOracle` contract (which inherits from an `IOracle` interface, unused in the `SturyOracle` contract). Therefore, the `BALWSTETHWETHOracle` contract could not be integrated as another `assetSource` of the `SturdyOracle` contract, resulting in non-trivial setup on how it would actually be used in the protocol.

Recommendation: Currently, it seems like the `SturdyOracle.getAssetPrice()#L111` is misusing the interface in the line `int256 price = IChainlinkAggregator(source).latestAnswer()` while the `source` is the `BALWSTETHWETHOracle` and not directly a chainlink oracle. If this is the case, please change the line from `IChainlinkAggregator` to `IOracle` instead. Otherwise, clarify how the `BALWSTETHWETHOracle` contract would be integrated with the rest of the protocol. If it is intended to be used within `SturdyOracle`, its function signatures would require adjustment.

Update: The interface `IChainlinkAggregator` was replaced with `IOracle`, which is implemented by `BALWSTETHWETHOracle`.

QSP-29 Swap Slippage Is Amplified by Leverage

Severity: Undetermined

Status: Acknowledged

File(s) affected: `protocol/leverage/GeneralLevSwap.sol`

Description: The exploit scenario described in QSP-2 demonstrates that the effect of swap slippage can be greatly amplified. In that scenario, the swap slippage was only `10%,` and yet the user lost `45%` of their position's value. The amplification of slippage is primarily caused by the need to pay back the exact amount of borrowed tokens (plus any flash loan fees) to the flash loan provider. The flash loan provider absorbs no fraction of the loss of value due to slippage, so the entire cost is absorbed by the user. It follows that using higher leverage, and thus a higher borrow amount, leads to a higher amplification of the swap slippage rate.

Recommendation: Unfortunately, the amplification of slippage seems to be an unavoidable consequence of the contract's design. As such, we recommend the following actions to mitigate the issue:

1. Implement effective slippage controls such that the user can easily specify how much value they are willing to lose.
2. Consider setting a limit on the amount of leverage that can be used on the contract level.
3. Inform users about the effect of leverage on the amplification of slippage.

Update: The Sturdy team acknowledged that the amplification of slippage is unavoidable with the current design. They will update their documentation accordingly so that users are aware of this.

Automated Analyses

Slither

Most issues found by Slither were false positives; the rest have been incorporated into the report.

Code Documentation

1. All `public` and `external` functions, as well as events, should be documented according to the NatSpec standard. (**Update:** Fixed)
2. The readability and auditability of the codebase could be improved with more in-line comments, especially for complex calculations.
3. Add a comment in the `AuraBalancerLPVault._depositToYieldPool()` function to state that it needs to approve the `lendingPoolAddress` to deposit in the `GeneralVaul._deposit()` function. The purpose of the approval is unclear when reading the `_depositToYieldPool()` function itself. (**Update:** Fixed)
4. The code comment for the `_principal` input parameter of the `GeneralLevSwap.zapLeverageWithFlashloan()` function is slightly misleading. It states to be the "amount of stablecoin". The `AURAWSTETHWETHLevSwap` contract, a subclass of `GeneralLevSwap`, can only use `WETH` as the `zappingAsset` because the `require(ENABLED_BORROWING_ASSET[_zappingAsset], ...)` check is in place. `WETH` is not a stablecoin. We recommend rephrasing the documentation to reflect the real situation. (**Update:** Fixed)

5. Fix the typo "addrss" in `L43:GeneralLevSwap.sol`. (**Update:** Fixed)
6. In `L252:GeneralLevSwap.sol`, the text "remained stable coin" should be changed to "remaining stable coins". (**Update:** Fixed)

Adherence to Best Practices

1. Explicitly declare the visibility of all state variables, including:
    1. `GenLevSwap.ENABLED_BORROWING_ASSET` (**Update:** Fixed)
    2. `LeverageSwapManager._levSwappers` (**Update:** Fixed)
2. Replace usage of the `safeApprove()` function as it is considered to be deprecated by OpenZeppelin.
3. Consider making `GeneralLevSwap` into an `abstract` contract, as it is unusable on its own. Furthermore, consider removing the function bodies of `_swapFrom()` and `_swapTo()`. This way, every contract that inherits from `GeneralLevSwap` must override those functions. (**Update:** Fixed)
4. Within the function, `GeneralLevSwap._executeOperation()`, consider renaming the variables `arg0`, `arg1`, `arg2`, and `arg3` to meaningful names. Another option would be to use a struct that can be encoded and decoded to clarify the message's data structure. (**Update:** Fixed)
5. The `_interalToken` variable in the `AuraBalancerLPVault` should be renamed to `_internalToken`. (**Update:** Fixed)
6. The function `GeneralLeveSwap._calcBorrowableAmount()` is unused and can be removed. (**Update:** Fixed)
7. The `_remove` of the `GeneralLevSwap._remove()` function is unused and can be removed. (**Update:** Fixed)
8. `BALANCER_VAULT` is declared in `GeneralLevSwap` and shadowed by the `BALANCER_VAULT` definition in `AURAWSTETHWETHLevSwap`. Define `BALANCER_VAULT` in only one place. (**Update:** Fixed)
9. In `GeneralLevSwap._leverageWithFlashloan()`, consider moving `L472-474` under the `if (_flashLoanType == FlashLoanType.AAVE) {}` block on `L487` as it is only related to Aave flash loan. Similar to the `GeneralLevSwap.withdrawWithFlashloan()` function on `L220-222`. (**Update:** Fixed)
10. Functions can be marked as `external` if not called internally in a contract. These functions can be marked as `external`:
    1. `BALWSTETHWETHOracle.peek()` (**Update:** Fixed)
    2. `BALWSTETHWETHOracle.get()` (**Update:** Fixed)
    3. `LeverageSwapManager.registerLevSwapper()` (**Update:** Fixed)
    4. `LeverageSwapManager.getLevSwapper()` (**Update:** Fixed)
11. `AURAWSTETHWETHLevSwap._getMinAmount()` uses a hardcoded slippage value. Consider adding it as a constant variable `SLIPPAGE` within the contract. (**Update:** Fixed)
12. The explicit activation of ABI coder V2 via `pragma abicoder v2` is no longer necessary since Solidity version 0.8, as since then, it is activated by default. (**Update:** Fixed)

Test Results

Test Suite Results

The tests were run by running the following commands in sequence:

1. `FORK=main yarn hardhat node`
2. `yarn sturdy_eth:evm:fork:mainnet:migration`
3. `yarn test:eth`

All `157` tests passed. However, some test files were commented out, such as `test-suites/test-eth/liquidator-with-flashloan.spec.ts`. The test suite includes integration tests based on a local mainnet fork and does not just rely on unit tests.

However, the tests leave a lot to be desired. There are some critical scenarios that are not tested such as the slippage constraints of the leveraged swap functionality. There are some functions which always return incorrect values and haven't been tested, such as `AuraBalancerLPVault.vaultYieldInPrice()`. Furthermore, there are many missing assertions. For example, the emission of several events such as `AuraBalancerLPVault.SetParameters` is not asserted. We recommend taking a comprehensive approach to testing where all functionality is verified and all details are checked.

Also, due to the lack of a code coverage script, we cannot determine which parts of the contracts' code have not been tested. We highly recommend that the Sturdy team implements a code coverage script to determine whether all the appropriate code has been tested.

Update: The test suite was run once again after fix review phase. All `157` tests passed as before. However, only minimal changes to tests have been made and we still recommend following our suggestions above to improve the test suite.

```
yarn run v1.22.19
warning package.json: License should be a valid SPDX license expression
$ FORK=main SKIP_DEPLOY=true TS_NODE_TRANSPILE_ONLY=1 hardhat test ./test-suites/test-eth/*.spec.ts --network localhost
Creating Typechain artifacts in directory types for target ethers-v5
Successfully generated Typechain artifacts!

- Enviroment
  - Fork Mode activated at network:  main
  - Provider URL: eth-mainnet.alchemyapi.io
  - Network : localhost
Duplicate definition of RewardsClaimed (RewardsClaimed(address,address,uint256), RewardsClaimed(address,address,address,uint256))

***************
Setup and snapshot finished
***************

  AddressesProviderRegistry
     ✓ Checks the addresses provider is added to the registry (21000 gas)
     ✓ tries to register an addresses provider with id 0 (21000 gas)
     ✓ Registers a new mock addresses provider (100796 gas)
     ✓ Removes the mock addresses provider (105292 gas)
     ✓ Tries to remove a unregistered addressesProvider (25496 gas)
     ✓ Tries to remove a unregistered addressesProvider (25496 gas)
     ✓ Tries to add an already added addressesProvider with a different id. Should overwrite the previous id (62875 gas)

  AToken: Modifiers
     ✓ Tries to invoke mint not being the LendingPool (21000 gas)
     ✓ Tries to invoke burn not being the LendingPool (21000 gas)
```

```
    ✓ Tries to invoke transferOnLiquidation not being the LendingPool (21000 gas)
    ✓ Tries to invoke transferUnderlyingTo not being the LendingPool (21000 gas)

  AToken: Permit
    ✓ Checks the domain separator (21000 gas)
    ✓ Get aWETH for tests (369385 gas)
    ✓ Reverts submitting a permit with 0 expiration (250739 gas)
    ✓ Submits a permit with maximum expiration length (335891 gas)
    ✓ Cancels the previous permit (131220 gas)
    ✓ Tries to submit a permit with invalid nonce (46068 gas)
    ✓ Tries to submit a permit with invalid expiration (previous to the current block) (46068 gas)
    ✓ Tries to submit a permit with invalid signature (46068 gas)
    ✓ Tries to submit a permit with invalid owner (46068 gas)

  AToken: Transfer
    ✓ User 0 deposits 2 WETH, transfers to user 1 (523772 gas)
    ✓ User 1 tries to transfer a small amount of WETH back to user 0 (303440 gas)

  WSTETHWETH Deleverage with Flashloan
    leavePosition - full amount:
      ✓ WETH as borrowing asset (4196789 gas)

  WSTETHWETH Deleverage with Flashloan
    leavePosition - partial amount:
enterPosition HealthFactor:  1198996022912859593
leavePosition 10% HealthFactor:  1198613146959092897
leavePosition 20% HealthFactor:  1197701907555347048
leavePosition 30% HealthFactor:  1194813678753610210
leavePosition 40% HealthFactor:  1000000018472386603
      ✓ WETH as borrowing asset (10396252 gas)

  WSTETHWETH Leverage Swap
    configuration
      ✓ WETH should be available for borrowing. (21000 gas)
    enterPosition(): Prerequisite checker
      ✓ should be reverted if try to use zero amount (21000 gas)
      ✓ should be reverted if try to use invalid stable coin (21000 gas)
      ✓ should be reverted when collateral is not enough (21000 gas)
    enterPosition():
Expected Leverage:  4.6
Current Leverage:  4.701710620665211523 71
      ✓ WETH as borrowing asset (2191635 gas)
Expected Leverage:  4.6
Current Leverage:  4.701661095579918541 62
      ✓ WETH as borrowing asset (3780256 gas)
Expected Leverage:  4.6
Current Leverage:  4.701611622221035143 27
      ✓ WETH as borrowing asset (3672264 gas)
    repay():
      ✓ WETH (1952521 gas)
    liquidation:
      ✓ WETH (1565973 gas)

  AuraWSTETHWETHVault - Deposit & Withdraw
    ✓ should be reverted if try to use an invalid token as collateral (21000 gas)
    ✓ should be reverted if try to use any of coin other than WSTETH-WETH as collateral (21000 gas)
    ✓ deposit WSTETH-WETH for collateral (966552 gas)
    ✓ transferring aAURAWSTETH_WETH should be success after deposit BAL_WSTETH_WETH_LP (1369254 gas)
    ✓ withdraw from collateral should be failed if user has not enough balance (521893 gas)
    ✓ withdraw from collateral (1264073 gas)

  AuraWSTETHWETHVault - Process Yield
    ✓ send yield to YieldManager (1418536 gas)

  WSTETHWETH Zap Deposit
    configuration
      ✓ WETH should be available for borrowing. (21000 gas)
    zapDeposit(): Prerequisite checker
      ✓ should be reverted if try to use zero amount (21000 gas)
      ✓ should be reverted if try to use invalid stable coin (21000 gas)
      ✓ should be reverted when collateral is not enough (21000 gas)
    zapDeposit():
      ✓ zap into LP vault with WETH (1277898 gas)

  WSTETHWETH Zap Leverage with Flashloan
    zapLeverageWithFlashloan(): Prerequisite checker
      ✓ should be reverted if try to use zero amount (21000 gas)
      ✓ should be reverted if try to use invalid stable coin (21000 gas)
      ✓ should be reverted when collateral is not enough (21000 gas)
    zapLeverageWithFlashloan():
Expected Leverage:  4.6
Current Leverage:  4.786691226619242055
      ✓ WETH as borrowing asset (2342512 gas)

  Deposit ETH_STETH_LP as collateral and other as for pool liquidity supplier
=================================================
Supplier 0x8401Eb5ff34cc943f096A32EF3d5113FEbE8D4Eb deposited: 10 WETH
totalDebtETH:  0
availableBorrowsETH:  0
currentLiquidationThreshold:  0



Borrower 0x306469457266CBBe7c0505e8Aad358622235e768 deposited: 10 ETH_STETH_LP
totalDebtETH:  0
availableBorrowsETH:  9.197040200120388135
currentLiquidationThreshold:  0.0000000000000093



Borrower 0x306469457266CBBe7c0505e8Aad358622235e768 borrowed: 9000000000000000000 WETH
totalDebtETH:  9
availableBorrowsETH:  0.197040200120388135
currentLiquidationThreshold:  0.0000000000000093



    ✓ User1 deposits WETH, User deposits ETH_STETH_LP as collateral and borrows WETH (2154573 gas)

  LendingPoolConfigurator
    ✓ Reverts trying to set an invalid reserve factor (21000 gas)
    ✓ Deactivates the WETH reserve (106889 gas)
    ✓ Rectivates the WETH reserve (144408 gas)
    ✓ Check the onlySturdyAdmin on deactivateReserve  (58519 gas)
    ✓ Check the onlySturdyAdmin on activateReserve  (58519 gas)
    ✓ Freezes the WETH reserve (117038 gas)
    ✓ Unfreezes the WETH reserve (117047 gas)
    ✓ Check the onlySturdyAdmin on freezeReserve  (58528 gas)
    ✓ Check the onlySturdyAdmin on unfreezeReserve  (58528 gas)
    ✓ Deactivates the WETH reserve for borrowing (117013 gas)
    ✓ Activates the WETH reserve for borrowing (117846 gas)
    ✓ Check the onlySturdyAdmin on disableBorrowingOnReserve  (59361 gas)
    ✓ Check the onlySturdyAdmin on enableBorrowingOnReserve  (59361 gas)
    ✓ Deactivates the WETH reserve as collateral (144533 gas)
    ✓ Activates the WETH reserve as collateral (170344 gas)
    ✓ Check the onlySturdyAdmin on configureReserveAsCollateral  (85172 gas)
    ✓ Disable stable borrow rate on the WETH reserve (143722 gas)
    ✓ Enables stable borrow rate on the WETH reserve (117002 gas)
    ✓ Check the onlySturdyAdmin on disableReserveStableRate (58452 gas)
    ✓ Check the onlySturdyAdmin on enableReserveStableRate (58452 gas)
    ✓ Changes the reserve factor of WETH (117508 gas)
    ✓ Check the onlyLendingPoolManager on setReserveFactor (59056 gas)
    ✓ Reverts when trying to disable the WETH reserve with liquidity on it (412277 gas)
```

```
  ETHSTETH Deleverage with Flashloan
    leavePosition - full amount:
       ✓ WETH as borrowing asset (4431628 gas)

  ETHSTETH Deleverage with Flashloan
    leavePosition - partial amount:
enterPosition HealthFactor:  1174143446469452409
leavePosition 10% HealthFactor:  1173966798168725059
leavePosition 20% HealthFactor:  1173621056571453307
leavePosition 30% HealthFactor:  1172618236611495772
leavePosition 40% HealthFactor:  1000000001264127333
       ✓ WETH as borrowing asset (11284782 gas)

  ETHSTETH Leverage Swap
    configuration
       ✓ WETH should be available for borrowing. (21000 gas)
    enterPosition(): Prerequisite checker
       ✓ should be reverted if try to use zero amount (21000 gas)
       ✓ should be reverted if try to use invalid stable coin (21000 gas)
       ✓ should be reverted when collateral is not enough (21000 gas)
    enterPosition():
Expected Leverage:  4.6
Current Leverage:  4.600326064422937625 65
       ✓ WETH as borrowing asset (2305685 gas)
Expected Leverage:  4.6
Current Leverage:  4.600319515228523129 27
       ✓ WETH as borrowing asset (4039115 gas)
Expected Leverage:  4.6
Current Leverage:  4.600312966964623204 12
       ✓ WETH as borrowing asset (3943458 gas)
    repay():
       ✓ WETH (2085203 gas)
    liquidation:
       ✓ WETH (1928348 gas)

  ConvexETHSTETHVault - Deposit & Withdraw
     ✓ should be reverted if try to use an invalid token as collateral (21000 gas)
     ✓ should be reverted if try to use any of coin other than ETH-STETH as collateral (21000 gas)
     ✓ deposit ETH-STETH for collateral (1350542 gas)
     ✓ transferring aCVXETH_STETH should be success after deposit ETH_STETH_LP (1762939 gas)
     ✓ withdraw from collateral should be failed if user has not enough balance (530264 gas)
     ✓ withdraw from collateral (1692200 gas)

  ConvexETHSTETHVault - Process Yield
     ✓ send yield to YieldManager (1900863 gas)

  AddressesProviderRegistry
     ✓ Checks the addresses provider is added to the registry (21000 gas)

  LendingPoolAddressesProvider
     ✓ Test the accessibility of the LendingPoolAddressesProvider (49674 gas)
*** LendingPool ***

Network: localhost
tx: 0x0a3b6791bf3459f68ab53af98fca1e2ac0b38c8ee3bab79ae4539a4c49235394
contract address: 0x45652Da975f14612CD42C24CDEC18f7f4886Ab01
deployer address: 0xb4124cEB3451635DAcedd11767f004d8a28c6eE7
gas price: 61946393506
gas used: 4793570

******

     ✓ Tests adding  a proxied address with `setAddressAsProxy()` (5405710 gas)
     ✓ Tests adding a non proxied address with `setAddress()` (631894 gas)

  Pausable Pool
     ✓ User 0 deposits 2 WETH. Configurator pauses pool. Transfers to user 1 reverts. Configurator unpauses the network and next transfer succees (694332 gas)
     ✓ Deposit (400693 gas)
     ✓ Withdraw (492580 gas)
     ✓ Borrow (174694 gas)
     ✓ Repay (174694 gas)

  Interest rate strategy tests
*** DefaultReserveInterestRateStrategy ***

Network: localhost
tx: 0x43dbf3ca264514edb204b3eed96cc86ce5ead55396625205001cc9fc77552935
contract address: 0xf8C7407850bdc63632D32c260D331C50af18782f
deployer address: 0xb4124cEB3451635DAcedd11767f004d8a28c6eE7
gas price: 61946393506
gas used: 669603

******

     ✓ Checks rates at 0% utilization rate, empty reserve (669603 gas)
     ✓ Checks rates at 80% utilization rate (669603 gas)
     ✓ Checks rates at 100% utilization rate (669603 gas)
     ✓ Checks rates at 100% utilization rate, 50% stable debt and 50% variable debt, with a 10% avg stable rate (669603 gas)

  ReEntrancy Oracle Test - if check is disabled, re-entrancy attack would be success
     ✓ Curve ETH_STETH LP Token Price (2364048 gas)

  ReEntrancy Oracle Test - if check is enabled, re-entrancy attack would be failed
     ✓ Curve ETH_STETH LP Token Price (601820 gas)

  Deposit ETH_STETH_LP as collateral and other as for pool liquidity supplier
===============================================
Supplier 0x8401Eb5ff34cc943f096A32EF3d5113FEbE8D4Eb deposited: 10 WETH
totalDebtETH:  0
availableBorrowsETH:  0
currentLiquidationThreshold:  0



Borrower 0x306469457266CBBe7c0505e8Aad358622235e768 deposited: 10 stETH_STETH_LP
totalDebtETH:  0
availableBorrowsETH:  9.197040200120388135
currentLiquidationThreshold:  0.0000000000000093



Borrower 0x306469457266CBBe7c0505e8Aad358622235e768 borrowed: 9000000000000000000 WETH
totalDebtETH:  9
availableBorrowsETH:  0.197040200120388135
currentLiquidationThreshold:  0.0000000000000093



     ✓ User1 deposits WETH, User deposits ETH_STETH_LP as collateral and borrows WETH (2502687 gas)

  Variable debt token tests
     ✓ Tries to invoke mint not being the LendingPool (21000 gas)
     ✓ Tries to invoke burn not being the LendingPool (21000 gas)

  VariableYieldDistribution: configuration
     ✓ Only EmissionManager can register an asset (21000 gas)
     ✓ Should be reverted if the vault address is invalid (21000 gas)
     ✓ Should be reverted if the asset is already configured (21000 gas)

  VariableYieldDistribution: Scenario #1
     ✓ Register ETHSTETH vault (21000 gas)
     ✓ Borrower provides some ETHSTETH token (1355378 gas)
     ✓ After some time, borrower can see his claimable rewards (1782960 gas)
```

```
    ✓ ClaimRewards: should be failed when use invalid address as an receiver address (545485 gas)
    ✓ ClaimRewards: borrower can get rewards (758798 gas)

  VariableYieldDistribution: Senario #2
    ✓ User1 deposits 2 ETHSTETH (1355378 gas)
    ✓ After one day, User2 deposits 4 ETHSTETH (3518378 gas)
    ✓ After one day pass again, the rewards amount should be the same for both. (2184000 gas)

  VariableYieldDistribution: Scenario #3
    ✓ User1 deposits 4 ETHSTETH (1355378 gas)
    ✓ After one day, User1 withdraws 4 ETHSTETH (3444916 gas)
    ✓ User1 can see his claimable rewards. (2726314 gas)
    ✓ On the same day, User2 deposits 4 ETHSTETH (1844338 gas)
    ✓ After one day pass again, the rewards amount should be the same for both. (1674153 gas)

  VariableYieldDistribution: Scenario #4
    ✓ User1 deposits 1 ETHSTETH (1355378 gas)
    ✓ The next day, User2 deposits 5 ETHSTETH (3518378 gas)
    ✓ The second day, User2 withdraws 5 ETHSTETH (3383009 gas)
    ✓ The same day, someone executes processYield. (1721394 gas)
    ✓ The 3rd day, User1 deposits 1 ETHSTETH (1829460 gas)
    ✓ The 4th day, the rewards amount should be the same for both. (1666863 gas)

  VariableYieldDistribution: Scenario #5
    ✓ User1 deposits 1 ETHSTETH (1355378 gas)
    ✓ The next day, execute processYield and User1 takes the half of his rewards (2020973 gas)
    ✓ The 2nd day, User1 queries his claimalbe rewards, and deposits again 2 (2414383 gas)
    ✓ The 3rd day, User1 withdraw 1 token, and User2 deposits 5 (4541342 gas)
    ✓ The 4th day, someone execute processYield (1618620 gas)
    ✓ User1 takes all available rewards. (604416 gas)

  VariableYieldDistribution: Scenario #6
    ✓ User1 deposits 1 ETH_STETH (1355378 gas)
    ✓ The next day, execute processYield and User1 takes the half of his rewards (2020973 gas)
    ✓ The 2nd day, User1 queries his claimalbe rewards, and deposits again 2 (2414383 gas)
    ✓ The 3rd day, User1 withdraw 1 token, and User2 deposits 5 (4541342 gas)
    ✓ The 4th day, someone execute processYield (1618620 gas)
    ✓ User1 takes all available rewards. (604416 gas)

  Withdraw WETH
=================================================
Supplier 0x8401Eb5ff34cc943f096A32EF3d5113FEbE8D4Eb deposited: 2 WETH
totalDebtETH:  0
availableBorrowsETH:  0
currentLiquidationThreshold:  0



Supplier 0x8401Eb5ff34cc943f096A32EF3d5113FEbE8D4Eb withdraw: 2000000000000000000 WETH
totalDebtETH:  0
availableBorrowsETH:  0
currentLiquidationThreshold:  0



    ✓ User1 deposits WETH and then withdraw WETH (583487 gas)

  Yield Manger: configuration
    ✓ Registered reward asset count should be 4 (21000 gas)
    ✓ CRV should be a reward asset. (21000 gas)
    ✓ CVX should be a reward asset. (21000 gas)
    ✓ BAL should be a reward asset. (21000 gas)
    ✓ AURA should be a reward asset. (21000 gas)
    ✓ Should be WETH as an exchange token (21000 gas)
    ✓ Should be failed when set invalid address as an exchange token (21000 gas)

  Yield Manager: simulate yield in vaults
    ✓ Convex ETHSTETH vault (1900863 gas)
    ✓ Aura WSTETHWETH vault (2437861 gas)

  Yield Manger: distribute yield
    ✓ Should be failed when use swap path including invalid tokens (4146460 gas)
APR:  1.2066162805643588
    ✓ Distribute yield CRV,CVX (4873887 gas)
APR:  232.4372831909282
    ✓ Distribute yield BAL (3810308 gas)
APR:  1526.0826690743145
    ✓ Distribute yield AURA (3672361 gas)
```

Solc version: 0.8.10		Optimizer enabled: true		Runs: 200	Block limit: 6718946 gas	
Methods						
Contract	Method	Min	Max	Avg	# calls	eur (avg)
AToken	permit	46068	85152	57235	7	-
AuraBalancerLPVault	depositCollateral	706761	2184000	1359077	46	-
AuraBalancerLPVault	processYield	116745	545485	441766	29	-
AuraBalancerLPVault	withdrawCollateral	742180	2207441	1386113	8	-
AURAWSTETHWETHLevSwap	enterPositionWithFlashloan	1579544	1785810	1700723	16	-
AURAWSTETHWETHLevSwap	withdrawWithFlashloan	1774801	2413986	2090297	10	-
AURAWSTETHWETHLevSwap	zapDeposit	-	-	1159252	1	-
AURAWSTETHWETHLevSwap	zapLeverageWithFlashloan	-	-	1821858	1	-
ERC20	approve	26188	51451	43099	81	-
ERC20	transfer	29694	530264	86807	92	-
LendingPool	borrow	-	-	445974	2	-
LendingPool	deposit	205610	255539	240291	25	-
LendingPool	liquidationCall	1131829	1489231	1310530	2	-
LendingPool	repay	275283	302026	282621	5	-
LendingPool	withdraw	-	-	209266	1	-
LendingPoolAddressesProvider	setAddress	-	-	48428	1	-
LendingPoolAddressesProvider	setAddressAsProxy	-	-	583466	2	-
LendingPoolAddressesProvider	transferOwnership	-	-	28674	2	-
LendingPoolAddressesProviderRegistry	registerAddressesProvider	37379	79796	65657	3	-
LendingPoolAddressesProviderRegistry	unregisterAddressesProvider	-	-	25496	4	-
LendingPoolConfigurator	activateReserve	-	-	58519	4	-
LendingPoolConfigurator	configureReserveAsCollateral	61167	85172	78313	7	-
LendingPoolConfigurator	deactivateReserve	-	-	85889	2	-
LendingPoolConfigurator	disableBorrowingOnReserve	-	-	58485	2	-

LendingPoolConfigurator	disableReserveStableRate	-	-	58550	2	-
LendingPoolConfigurator	enableBorrowingOnReserve	-	-	59361	4	-
LendingPoolConfigurator	enableReserveStableRate	-	-	58452	4	-
LendingPoolConfigurator	freezeReserve	-	-	58519	2	-
LendingPoolConfigurator	setPoolPause	50928	72838	59355	13	-
LendingPoolConfigurator	setReserveFactor	-	-	59056	3	-
LendingPoolConfigurator	unfreezeReserve	-	-	58528	4	-
ReEntrancyTest	enableCheck	-	-	43298	1	-
ReEntrancyTest	test	-	-	1805526	1	-
StableDebtToken	approveDelegation	-	-	53623	11	-
VariableYieldDistribution	claimRewards	164825	238013	213574	7	-
YieldManager	distributeYield	363958	494027	421077	5	-
Deployments					% of limit	
DefaultReserveInterestRateStrategy		-	-	669603	10 %	-
LendingPool		-	-	4793570	71.3 %	-
ReEntrancyTest		-	-	363004	5.4 %	-

157 passing (6m)

✓ Done in 384.64s.

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

```
a02bfa22973e7125a15d95a2d3e88b8c6f5bf2284735f437042ab97a8adc5546 ./contracts/protocol/vault/ethereum/AuraVault/AuraBalancerLPVault.sol
c851535af6deaddb010c597703b66c2c619702b2db9ec244224687e53050745e ./contracts/misc/BALWSTETHWETHOracle.sol
c56bf661c92309127d7a17ccb8969b1b69eb409cf9bfcee120ed086173134712 ./contracts/protocol/leverage/LeverageSwapManager.sol
b4839ca9fa23467ee3fc89a375bd8b581136ae2290ec15fe1269520db5d7f63e ./contracts/protocol/leverage/GeneralLevSwap.sol
decd932fd190cd7cf8d2a7941eeac9702d2063761b0ca8e6dc5b9bc0d09a0d50 ./contracts/protocol/leverage/ethereum-eth/AURAWSTETHWETHLevSwap.sol
```

Tests

```
7aedaea66da8b06d17dfe7216a28e85c1e1277cc82b39cb166db28d86bc310a7 ./test-eth/re-entrancy-oracle.spec.ts
bdd81f21f5925004cc94667faee960f86976108132994cb953b89884a7a9c246 ./test-eth/convex-eth-steth-deleverage-with-flashloan.spec.ts
928c3ab14d508060707ad961505dc142987f385757ef66331974536289f8a35e ./test-eth/aura-wsteth-weth-zap.spec.ts
1eeef5314a71d5947bac7cb0529f13d1ffb533e09d86cdd697047dcfd7250bd1 ./test-eth/atoken-transfer.spec.ts
31105b3a9c9f8372597a7007081d820aee486a5e3baa48184cb46817c3ccd3a6 ./test-eth/__setup.deploy.spec.ts
c600a421676e95588d2612cfeb94ea3081712b33fc737385bfb34a2eb38e66fc ./test-eth/atoken-modifiers.spec.ts
2e3db0646b1847b30fa3fcfa418f70aa3f66099badc19d2af5a360b89441ffdf ./test-eth/configurator.spec.ts
32c0c6dca3d2ba3b84fe7dce06d7d71b124e0c28ca16cbf1608b486b37681176 ./test-eth/rate-strategy.spec.ts
eddfe1ce7e10a3d84006d35ac3e4658846aec450e588ad283afcd1da907f3d03 ./test-eth/d.deploy.spec.ts
a0473470b409b033f507a974fb0e42ecd774dc56cc42843b9bad0dd87411abc8 ./test-eth/lending-pool-addresses-provider.spec.ts
b16043c47da3231b2cd9859fa7d55144ac66abb95c90d5cd35ed14b55c9f5ab6 ./test-eth/borrow.spec.ts
55fdfeafedc2475b1920396a868113e2972f446576d23cf5527fe5c2910b9b40 ./test-eth/aura-wsteth-weth-deleverage-with-flashloan.spec.ts
9204fcaf62f853387fa0949f6a30180dcd99281001c6d5b6bac45f3cc1e21b88 ./test-eth/atoken-permit.spec.ts
be3f2903fc2f68315b4dc8f85e8439eb1b43c9ed2ea9b5de0484c226248ccf1b ./test-eth/aura-wsteth-weth-leverage-with-flashloan.spec.ts
e8713d542e8682cbe9f8ebf37b8f8494e3c8286512895d6c9e6cb743cd37284a ./test-eth/liquidator-with-flashloan.spec.ts
66168a2d1d656dcf5142764b1beaeb44d1df65d6ff15cca83c80de2a69502fde ./test-eth/yield-manager.spec.ts
056849914b75910060e6266c42ac3bc0434ea172ab54fedf1c9409333c08975f ./test-eth/pausable-functions.spec.ts
2cec420b72f6003d734fc4ba8928a2b09910b97e8d924736c6f8fccc02055c51 ./test-eth/addresses-provider-registry.spec.ts
9014bc51732170b11bc812db68a76d1b9bcca67f07834917010099f27a911b2f ./test-eth/repay.spec.ts
12a3d5c9d4ac6c37c3519a9e57488d236865e77943498e696724fd5b2183a63d ./test-eth/aura-wsteth-weth-vault.spec.ts
86e442a5306a0630cb0d1001e66b49ab3d52a1532c6bedc23e8e0ac796d3b57d ./test-eth/variable-debt-token.spec.ts
45164922febb05d9e485467aef99c6c16bf5eb5e69dd250c01768ed0cca576aa ./test-eth/convex-eth-steth-vault.spec.ts
df9b985d77ae6cc97280a671445129f9883816425d47ef0d9e2bdfa6d2a39051 ./test-eth/withdraw.spec.ts
e1c72d738569db53698497f076e87749fe0b40127cc65350d679137155f99606 ./test-eth/variable-yield-distribution.spec.ts
2e1f4b56373f31d29d28cd43375eab52733e99364b0c363e64f62672ad7a30cb ./test-eth/convex-eth-steth-leverage-with-flashloan.spec.ts
6f5017715931333ab9b0bcb8925dba68c13c1a1cc9ce5b66048f19d6a79a9446 ./test-eth/helpers/almost-equal.ts
af2e2e344fd299ca6c8913117cb160f779cc3b61d5d068082cfe38b27963c935 ./test-eth/helpers/mint.ts
2b1a3000072b0430ff3a315815891093f17af9a5d848e4015954a07ed5e91cfb ./test-eth/helpers/make-suite.ts
c4751c67ac9e6010d85a9b0353a1d29a74222358e35f23f49377316d3977fe58 ./test-eth/helpers/utils/helpers.ts
65dca8b3a5c4edf170780fc0fcb019e55eb1bdd46671090b3420e4e359e48e14 ./test-eth/helpers/utils/math.ts
566d463794e3c622311f33a5bcd8ec094fffe9bf78e4d93cfe2e8c5cb6a7511d ./test-eth/helpers/utils/calculations.ts
eeec12f7622fdf2eb02b7b1eb6f8e2d727264774d4f0bdd44a5551177023b602 ./test-eth/helpers/utils/interfaces/index.ts
```

Changelog

- 2022-12-22 - Initial report
- 2023-01-24 - final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Aave, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites&aspo; owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that your access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.