SOLIDIFIED

Audit Report for Kresko Labs Pte. Ltd - January 12, 2024

Summary

Audit Report prepared by Solidified covering the Kresko Synthetic Asset Protocol.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on November 10, 2023, and the results are presented here.

Audited Files

The source code has been supplied in a public GitHub repository:

https://github.com/kreskohq/kresko-protocol

Commit number: `cbbc4c365e544a765a74554b9857a3ec0cf58252`

Fixes received at commit: `Dfe6593bfa638c4225ee587cc04d8797377954aa`

Additional received fixes at commits:

- `50d058f769180ab85f30866f7ebf0f2749e341de`
- `Afd6cfb72e1d2b5c8b4f39b36b0a935ee56937db`
- `fe69fca2f3a17b1886e90236dd3ad0ae8628b688`

File list:

```
├── src/contracts/core
│   ├── common/*
│   ├── kiss/*
│   ├── kresko-asset/*
│   ├── scdp/*
│   └── vault/*
```

Intended Behavior

The contracts implement a non-custodial, capital-efficient synthetic asset protocol that runs on the EVM. It facilitates creating and managing securely collateralized synthetic assets using smart contracts written in Solidity.

Specifically, users can mint the internal KISS stablecoin token by depositing various collateral assets into the KISS vault as well as deposit collateral into the shared collateralized debt position (SCDP) that have their value combined, enabling users to swap KISS tokens with synthetic assets referred to as Kresko Assets using the backing of the shared CDP. Moreover, users can participate in liquidations in case of an unhealthy SCDP.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits, and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium-High	In several places, token amounts are potentially rebased and thus require normalization. Any such oversight might lead to unintended consequences.
Code readability and clarity	High	The code is easy to read and, in general, very clear. Many naming conventions, code patterns, and other best practices were implemented.
Level of Documentation	Medium	Almost all functions are well documented with NatSpec and inline comments, which greatly enhance the overall understanding of the codebase. Additionally, the Gitbook documentation provides an in-depth explanation of how the protocol is intended to work. However, the recent changes that are part of this audit have

		not been sufficiently documented. Thus, certain assumptions, such as using multiple collateral tokens for the SCDP, had to be made.
Test Coverage	Medium-High	Due to the inability to determine the test coverage with the hardhat-coverage plugin, as well as having tests fragmented across Hardhat and Foundry, it was not possible to verify the overall test coverage. Nevertheless, based on our subjective assessment, we consider the test coverage to be medium-high. However, we recommend extending the test suite with more granular and detailed tests to ensure edge cases are covered.

Issues Found

Solidified found that the Kresko Labs contracts contain 2 critical issues, 7 major issues, 17 minor issues, and 15 informational notes.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Inability to fully repay and liquidate debt due to the incorrect comparison of rebased and normalized token amounts in the repaySCDP and liquidateSCDP functions	Critical	Resolved
2	Wrapping and unwrapping Kresko asset tokens results in insufficient backing	Critical	Resolved
3	Liquidations can leave SCDP below the minimum debt value	Major	Resolved
4	Inability to swap assets with the SCDPSwapFacet.swapSCDP function in case of insufficient liquidity of the fee token	Major	Resolved
5	Users are double charged fees in the Vault redeem function when there is insufficient balance in the contract	Major	Resolved
6	SCDP depositors are unable to withdraw their fair share of the remaining collateral if their deposited collateral token's liquidity is fully seized during liquidation	Major	Resolved
7	SCDP depositors of non-fee collateral tokens are not able to accumulate fees	Major	Resolved
8	KreskoAssetAnchor shares can be increased by supplying Kresko assets	Major	Resolved
9	Vault shares can be manipulated after emptying	Major	Resolved

	the underlying assets		
10	KISS token value can be manipulated shortly after deployment	Minor	Acknowledged
11	KreskoAsset implementation contract can be initialized	Minor	Acknowledged
12	totalCover value is incorrectly calculated in the SDebtIndex library	Minor	Resolved
13	The gate modifier contains a wrong balance check for Phase 2 of Quest For Kresk NFT	Minor	Resolved
14	Unbounded loops over arrays	Minor	Acknowledged
15	Prices.rawPrice always returns the primary oracle's price, even if it is stale and the secondary oracle's price is not	Minor	Resolved
16	Prices.handleSequencerDown should not return any price if the L2 sequencer is down	Minor	Resolved
17	Vault's maxRedeem, maxWithdraw, and maxDeposit functions do not correctly include vault fees	Minor	Resolved
18	Missing access control for Kresko.setAnchorToken when the anchor token is not set	Minor	Acknowledged
19	SCDPSwapFacet.previewSwapSCDP may not match swapSCDP amounts due to the different implementation of _swapFeeAssetOut	Minor	Resolved
20	Validations.validateSCDPKrAsset does not validate that the sum of swap fees is lower than 50%	Minor	Resolved
21	Actions.burnSCDP does not validate if the market is open	Minor	Acknowledged
22	Missing upgradeable initializers	Minor	Acknowledged

23	Unused return value of low-level delegatecall function	Minor	Resolved
24	Vault token spending allowance is incorrectly adjusted, resulting in less than allowed token spending	Minor	Resolved
25	Incorrect calculation of the seizeAmount value in the SCDPFacet.getMaxLiqValueSCDP helper function due to misplaced arguments	Minor	Resolved
26	Centralization risk	Minor	Acknowledged
27	Naming conventions	Note	Partially Resolved
28	Usage of low-level transfer function	Note	Acknowledged
29	KreskoAsset allowances get out of sync with rebases	Note	Acknowledged
30	Inconsistent revert behavior when a user-provided parameter is greater than internal storage variables	Note	Acknowledged
31	The protocol returns 0 for the collateralization ratio if the user has no debt	Note	Resolved
32	ERC-165 interfaceId's are not self-evident	Note	Acknowledged
33	Vault.maxMint should assume the agent has infinite assets	Note	Acknowledged
34	SafetyCouncilFacet.safetyStateSet is not reset to false if all assets return their pause state to false	Note	Acknowledged
35	Validations.validateCollateralArgs reverts with underflow if there are no deposited collateral assets	Note	Acknowledged
36	Potential unsafe casting of toWad if _amount is lower than zero	Note	Resolved

37	Vault.setGovernance does not implement a 2-step process for transferring the governance	Note	Resolved
38	Errors.id and Errors.symbol perform external calls to possibly untrusted addresses	Note	Acknowledged
39	Inconsistencies in Vault adaptation of ERC-4626	Note	Acknowledged
40	Inconsistent check for minimum collateral ratio and liquidation threshold	Note	Resolved
41	Possible zero address of state variables	Note	Acknowledged

Critical Issues

1. Inability to fully repay and liquidate debt due to the incorrect comparison of rebased and normalized token amounts in the SCDP's repaySCDP and liquidateSCDP functions

The `repaySCDP` and `liquidateSCDP` functions in the `SCDPFacet` contract check if the specified `_repayAmount` does not exceed the debt of the repaid asset in lines `68-75` and `142-144`, respectively. The asset's debt, stored in `SCDPAssetData.debt`, is the normalized value in case the asset is a rebasing asset. However, the compared `_repayAmount` is not normalized and is potentially rebased, resulting in an incorrect comparison. Consequently, the debt of a rebasing asset can not be fully repaid, leading to the accumulation of bad protocol debt.

Similarly, the check in line `83` compares the potentially rebased `seizedAmount` amount with the normalized `seizeAssetData.swapDeposits`.

Recommendation
We recommend using the normalized amounts for the comparisons.

Status
Resolved

2. Wrapping and unwrapping Kresko asset tokens results in insufficient backing

Users can wrap selected tokens by depositing the Kresko assets' underlying token via the `wrap` function in the `KreskoAsset` contract and, in return, receive Kresko asset tokens of equal value. As Kresko assets are potentially rebasing, for example, to adjust for stock splits, an anchor

token (`KreskoAssetAnchor`) is used to represent the normalized amount of possibly rebased underlying Kresko asset tokens.

However, due to minting the Kresko asset tokens in line `260` of the `wrap` function before the corresponding anchor tokens are minted in line `262`, the calculation in the `convertToShares` function to determine the number of anchor tokens to mint uses the already increased Kresko asset token supply. Consequently, fewer anchor tokens are minted.

In addition, the `unwrap` function burns Kresko asset tokens prior to burning the corresponding anchor tokens, resulting in receiving more underlying tokens. As a result, wrapping and unwrapping Kresko asset tokens allows draining the underlying tokens, thus causing the Kresko asset to be insufficiently backed.

Similarly, depositing and wrapping native tokens via the `receive` function mints Kresko asset tokens before the anchor tokens.

This issue has been disclosed by the Kresko team to Solidified during the audit.

Mitigation
The chosen solution to resolve the vulnerability is calling `_mint` in the `KreskoAsset.wrap` and `KreskoAsset.receive` functions after minting the anchor tokens via the `IKreskoAssetAnchor.wrap` function. Similarly, in the `KreskoAsset.unwrap` function, asset tokens are burned after calling the anchor token's `IKreskoAssetAnchor.unwrap` function.

Status
Resolved

Major Issues

3. Liquidations can leave SCDP below the minimum debt value

In `src/contracts/core/scdp/facets/SCDPFacet.sol:149`, the liquidation repayment amount in USD is bound to the minimum debt value or the maximum liquidatable value in USD. The issue is that the `liquidateSCDP` function never checks if the position will be left below the minimum debt value after this repayment. The function simply deducts the burned SCDP repayment amount from the SCDP asset debt and transfers the seized collateral to the liquidator.

Proof of Concept
Suppose the maximum liquidatable value is $10, the minimum debt value is $10, and the repayment amount is $9. Since the repayment amount is lower than the maximum liquidatable value, the SCDP will be left with $1, below the minimum debt value.

Recommendation
We recommend validating that the SCDP will not end up below the minimum debt value after the liquidation and require the liquidator to pay all or nothing in this case.

Status
Resolved

4. Inability to swap assets with the SCDPSwapFacet.swapSCDP function in case of insufficient liquidity of the fee token

In `SCDPSwapFacet`, the `swapSCDP` function uses `SSwap.handleAssetsOut`, which can return an `amountOut` value greater than the contract's asset balance since this return value is not the actual sum transferred to `_assetsTo`. This is done because when there is no "swap" owned collateral, some debt will be issued to the receiver, and the receiver will receive fewer assets

than the function initially calculated. In all cases, the `amountOut` return value is used to be transferred to the receiver and to calculate fees.

The problem is that when `_assetOutAddr` is the `feeAsset`, the `amountOut` may possibly be greater than the current liquidity. As a result, the final `protocolFeeTaken` value can be greater than the contract's total balance of the fee asset. In this scenario, the transaction will revert.

Recommendation
We recommend updating the `SCDPSwapFacet._paySwapFees` function to transfer at most the contract balance for the `feeRecipient`. While this mitigates the denial of service in this scenario, this means the protocol might not receive its full share of fees in all cases. Preferably, consider refactoring the SCDP design to charge fees in the swapped token.

Status
Resolved

Client response
Fixed as suggested without refactoring to use input asset as the fee.

5. Users are double charged fees in the Vault `redeem` function when there is insufficient balance in the contract

---

The `redeem` function in the `Vault` contract uses the contract's balance to repay the user when the amount of assets to be returned plus the fee is greater than the balance. To determine the new amount of shares to burn (`sharesIn`) and the new amount of fees to charge (`assetFee`), the balance is passed to the `previewWithdraw` function. This function uses the `handleWithdrawFee` function to compute the `assetsOut` and the `assetFee`. The `assetsOut` figure is then used to compute the `sharesIn` value.

However, the `assetsOut` figure already includes the fee when outputted from the `handleWithdraw` function when it is then used to compute `sharesIn`. This means the `sharesIn` value also includes the fee when computed. This results in the user being double charged fees

both in line `151`, where the `assetFee` is deducted from the number of assets to be returned to the user, and in line `158`, where `sharesIn` is burned from the user by the contract.

Recommendation
We recommend deducting the fee from `assetsOut` when computing `sharesIn`. This way, the number of shares burned will be equivalent to the number of assets returned to the user, which would already have the fee deducted.

Status
Resolved

6. SCDP depositors are unable to withdraw their fair share of the remaining collateral if their deposited collateral token's liquidity is fully seized during liquidation

---

The shared collateralized debt position (SCDP) allows users to deposit various collateral assets that have their value aggregated. Users can perform swaps, i.e., swapping kresko assets with KISS, using the backing from this shared collateral value, and paying fees to the depositors.

If the SCDP is liquidateable due to the collateral value being lower than the minimum collateralization ratio, liquidation occurs, and the collateral can be seized by the liquidator in return for the repaid debt. The `handleSeizeSCDP` function in the `SDeposits` library, called internally by the `liquidateSCDP` function, handles seizing a specific and specified collateral token from the shared pool. Collateral is preferably seized from the swap deposits, i.e., funds received as part of a swap, and if the swap deposits are insufficient, the collateral is seized from the shared vault deposits.

To adjust everyone's deposits and to reflect the seized collateral, the `liquidityIndexSCDP` index is used to scale the deposit amounts by the same ratio in `src/contracts/core/scdp/funcs/SDeposits.sol:122-124`. The index is set to zero if all of the collateral token's liquidity is seized.

However, if `liquidityIndexSCDP` is set to zero, it will result in a division by zero error in line `36` of the `handleDepositSCDP` function, preventing any new deposits for this collateral token. Depositors of this collateral token are now unable to withdraw funds, which is expected as the collateral was fully seized. Yet, as depositors are only able to withdraw funds for the collateral token they have initially deposited, this will also prevent withdrawing funds for other collateral tokens, e.g., swap deposits.

Recommendation
We recommend implementing a mechanism to allow depositors to withdraw their share of the remaining collateral.

Status
Resolved

7. SCDP depositors of non-fee collateral tokens are not able to accumulate fees

---

Swap fees are accumulated and distributed in the configured fee asset stored in `SCDPState.feeAsset`. Consequently, depositors who deposited collateral in the shared CDP are unable to withdraw their share of the accumulated fees if their deposited collateral token is not the configured fee asset token.

Recommendation
We recommend distributing fees to the depositors relative to their deposit value, regardless of the used collateral token.

Status
Resolved

Client response
This is temporarily resolved by restricting deposits to only either the fee asset or an asset that has previously accumulated fees.

8. KreskoAssetAnchor shares can be increased by supplying Kresko assets

The `KreskoAssetAnchor` contract is meant to be an ERC-4626 vault that handles the pro-rata representation of kresko assets in order to allow easier integration for the rebasing `KreskoAsset` with other protocols.

Kresko assets are minted via the `issue` and burned via the `destroy` functions of the `KreskoAssetAnchor` contract. However, the `KreskoAssetAnchor` contract also exposes the `mint` function from the inherited `ERC4626Upgradeable` contract. This allows supplying kresko assets in return for newly minted anchor tokens. This leads to a broken invariant between the `KreskoAssetAnchor` anchor contract and the `KreskoAsset` contract as the anchor token supply increases while the `KreskoAsset` supply remains unchanged.

Recommendation
We recommend overriding the `mint` function in the `KreskoAssetAnchor` contract and reverting the transaction when called.

Status
Resolved

9. Vault shares can be manipulated after emptying the underlying assets

In `Vault.sol`, the functions `previewDeposit`, `previewMint`, `previewRedeem`, and `previewWithdraw`, which are used in the `deposit`, `mint`, `redeem`, and `withdraw` functionality, all reset the vault token supply or the total underlying assets to `1e18` during calculations whenever either one of those reach zero.

This is problematic when either assets or shares are not *simultaneously* zero, as this logic effectively "resets" one of these parameters while the other remains unchanged.

This leads to the following two scenarios:

1. `totalSupply() == 0 && totalAssets() > 0`, leading to `tSupply == 1e18 && tAssets == totalAssets()`
2. `totalSupply() > 0 && totalAssets() == 0`, leading to `tSupply == totalSupply() && tAssets == 1e18`

To achieve scenario 1, the vault is emptied, and new assets are subsequently deposited to receive more shares than what is deserved:

1. A user exits the vault with all existing shares (KISS) to a deposit asset with less than 18 decimals
2. Due to rounding down when calculating the corresponding asset token amount, the vault gets left with 1 wei of assets but 0 shares
3. Thereafter, a subsequent asset deposit will receive `1e18` more shares (i.e., KISS) than intended

Scenario 2 is caused in the situation where the vault has a small shares supply and an insignificant amount of underlying assets. Specifically, the `totalAssets` function internally rounds down the dollar value of the underlying assets, thus potentially returning zero while the `totalSupply` function returns a non-zero amount of shares. As a result, subsequently depositing assets results in receiving less shares than intended.

This issue has been disclosed by the Kresko team to Solidified during the audit.

Recommendation

The chosen solution to resolve the vulnerability is resetting *both* shares and assets whenever either is zero. In addition, we recommend pre-minting a number of vault shares (KISS) after the deployment of the vault.

Status

Resolved

Minor Issues

10. KISS token value can be manipulated shortly after deployment

Kresko's KISS token is an internal stablecoin with a target value of 1 USD. KISS tokens are minted against the collateral value of the corresponding `Vault`'s collateral tokens (e.g., USDC, USDT, etc.).

However, immediately after the deployment of the `Vault` contract and before the initial deposit, the exchange rate can be significantly manipulated by directly transferring `1 wei` of a collateral token to the `Vault` contract. Subsequently calling the `deposit` function for the initial deposit will result in a very high number of share tokens, caused by the `tAssets` denominator being set to `1 wei` in line `237` of the `previewDeposit` function. Consequently, KISS tokens are worth significantly less than the anticipated value of 1 USD.

While this does not cause any noticeable issues due to the protocol not assuming a static value of 1 USD, it may cause unexpected behavior or not yet considered edge cases.

Recommendation
We recommend keeping track of the deposited collateral balance instead of directly querying the contract's collateral token balances with the `VAssets.getDepositValueWad` function. Alternatively, consider minting KISS tokens against deposited collateral as part of the deployment process.

Status
Acknowledged

Client response
We acknowledge this, and we will be minting vKISS as part of the development process.

11. KreskoAsset implementation contract can be initialized

The `KreskoAsset` contract is an upgradeable contract consisting of an implementation contract and a proxy contract. The state of the `KreskoAsset` contract is stored in the proxy contract, while the implementation contract contains the logic.

However, besides initializing the `KreskoAsset` proxy contract by calling the `initialize` function, the implementation contract can also be initialized by directly calling the `initialize` function, granting the caller the admin role.

While the `KreskoAsset` contract does not make use of `delegatecall`, which would allow the contract owner to destroy the implementation contract and thus affect the proxy contract as well, it is considered a best practice to prevent the initialization of the implementation contract.

Recommendation
We recommend calling the `_disableInitializers` function in the constructor of the `KreskoAsset` contract. This issue is also found in the KISS contract.

Status
Acknowledged

Client response
We acknowledge this, and we will be uncommenting the `_disableInitializers` for production deployments.

12. totalCover value is incorrectly calculated in the SDebtIndex library

The `cover` function in the `SDebtIndex` library pulls in assets from the caller to cover the debt of the SCDP. The total cover amount is stored in the `totalCover` variable of the `SDIState` struct and calculated by the `valueToSDI` function.

This `valueToSDI` function is assumed to return the value in WAD (18 decimals) precision based on the actively used `totalCoverAmount` function in the `SDebtIndex` library that returns the value in WAD precision.

However, the current implementation of the `valueToSDI` function results in a value with 36 decimals. Given that the `valueIn` parameter is in WAD precision (due to being the result of the `wadUSD` function call in line 31), `oracleDecimals` is 8, and the `SDIPrice` function returns a value in 8 decimals, calculating `(WAD * 1e8 * WAD) / 1e8` results in a value with 36 decimals.

As this `totalCover` value is not actively used and is currently only used for informational purposes, it does not affect the protocol, and we consider this a minor issue.

Recommendation
We recommend updating the `valueToSDI` function to return the value in WAD precision.

Status
Resolved

13. The `gate` modifier contains a wrong balance check for Phase 2 of Quest For Kresk NFT

In `src/contracts/core/common/Modifiers.sol:190`, the `gate` modifier validates if the message sender owns different Quest For Kresk NFTs for each gated phase. The issue is that the check for phase 2 will pass if the user has "either phase 1 or phase 2" NFTs, while intuitively, it should require that the user has "both phase 1 and phase 2" NFTs.

This issue's validity depends on the gating mechanism's expected behavior.

Recommendation
We recommend validating if the current check is intended or updating the `gate` modifier otherwise.

Status
Resolved

14. Unbounded loops over arrays

Many instances of the codebase contain unbounded loops over collateral and Kresko asset arrays, such as in `src/contracts/core/scdp/funcs/SGlobal.sol:69` and `121`.

Since users can freely deposit any number of supported collaterals and mint any number of supported Kresko assets, they can cause a denial of service on the protocol if these arrays are allowed to grow too much, as these operations may revert due to block gas limits. As a result, it may be impossible to perform critical operations such as repaying debt, withdrawing collateral, liquidating accounts, and others.

Recommendation
We recommend monitoring protocols' operations and supported collateral and Kresko assets so that no users can cause a Denial of Service due to unbounded loops. In addition, we recommend considering refactoring how positions are calculated so that it is not necessary to perform unbounded iterations.

Status
Acknowledged

Client response
We acknowledge this design choice and will apply necessary limitations to chains where the block gas limit combined with the asset count could, in theory, brick a user account.

15. `Prices.rawPrice` always returns the primary oracle's price, even if it is stale and the secondary oracle's price is not

In `src/contracts/core/common/funcs/Prices.sol:192`, the `rawPrice` function always returns the price from the primary oracle, even if it is invalid (stale) and the secondary oracle is not. Since this is used in `Validations.validateRawAssetPrice`, which is used by `AssetConfigurationFacet`, some admin operations may revert if the primary oracle is not working properly, even if the second oracle is.

Recommendation
We recommend returning both prices from the `rawPrice` function and performing the necessary checks in `validateRawAssetPrice`.

Status
Resolved

Client response
This is intended since we use the `rawPrice` only for push-oracles, mainly to avoid having to inject the Redstone payload in certain places. Refactor was applied to rename the function to better represent its usage

16. `Prices.handleSequencerDown` should not return any price if the L2 sequencer is down

---

In `src/contracts/core/common/funcs/Prices.sol:96`, the `handleSequencerDown` function returns a secondary oracle price if the L2 sequencer is down. The issue is that if a sequencer becomes unavailable, it is impossible to access read/write APIs that consumers are using, and applications on the L2 network will be down for most users without interacting directly through the L1 optimistic rollup contracts ("deposit" transactions).

This means the L2 sequencer being down does not have to do with ChainLink Price Feeds specifically being down but rather with L2 operations being unable to go through without being submitted via an L1 "deposit". As such, secondary oracle prices may also be outdated in this scenario. In addition, the `Prices.handleSequencerDown` behavior is inconsistent with `VAssets.price`, which simply reverts if the sequencer is down.

Recommendation
We recommend reverting operations in case the L2 sequencer is down, as they might mean stale prices.

Status
Resolved

17. Vault's maxRedeem, maxWithdraw, and maxDeposit functions do not correctly include vault fees

---

In `src/contracts/core/vault/Vault.sol:300`, the `maxRedeem` function, which is used by the `maxWithdraw` function, returns the maximum redeemable amount of shares for a user. The issue is that the behavior is inconsistent depending on whether the vault has enough assets for that specific `assetAddr` requested by the user or not.

If the vault does not have enough assets for that specific asset address, the returned value will be subject to the value returned by `previewWithdraw`, which includes asset fees, while if the vault does have enough assets, the return value will not include fees.

In addition, `maxDeposit` does not include any fees, while `maxMint` does.

Recommendation
We recommend amending the `maxRedeem`, `maxWithdraw`, and `maxDeposit` functions to be inclusive of fees as per EIP-4626.

Status
Resolved

18. Missing access control for Kresko.setAnchorToken when the anchor token is not set

---

In `KreskoAsset.sol`, the `setAnchorToken` function does not have the `onlyRole(Role.ADMIN)` modifier to validate that the anchor token can be set only by the admin. Since this function is not called during the contract initialization, it is possible that an attacker front-runs the admin transaction and sets the anchor token of the deployed contract first.

Recommendation

We recommend adding appropriate access controls to the `setAnchorToken` function.

Status

Acknowledged

Client response

This is fine as we deploy/initialize KreskoAsset + KreskoAssetAnchor + the proxies in a single transaction. All addresses are salted, so we know them in the initializers, allowing us to skip the access control check safely.

19. SCDPSwapFacet.previewSwapSCDP may not match swapSCDP amounts due to the different implementation of _swapFeeAssetOut

---

In `SCDPSwapFacet.sol`, the `previewSwapSCDP` function previews the `amountOut` that a swap would receive from submitting the `_amountIn` value as a parameter. The issue is that if the swap is targeting the `scdp().feeAsset`, the fees are not taken from `amountIn`, but rather from `amountOut`. As a result, the preview function might return an incorrect result.

Recommendation

We recommend updating the `previewSwapSCDP` function to correctly handle the case where `_assetOutAddr` is `scdp().feeAsset` to take fees on the output, not on the input.

Status
Resolved

20. `Validations.validateSCDPKrAsset` does not validate that the sum of swap fees is lower than 50%

In `Validations.sol`, the `validateSCDPKrAsset` function uses `validateFees` to check if the sum of swap fees is lower than 100%. The issue is that `Types.sol` specifies that the sum of swap fees must be lower than 50%, which is not checked anywhere.

Recommendation
We recommend updating the `valiadteSCDPKrAsset` function to correctly validate that the sum of swap fees is lower than 50%.

Status
Resolved

Client response
Resolved by removing mentions to the arbitrary 50% limit.

21. `Actions.burnSCDP` does not validate if the market is open

In `Actions.sol`, the `burnSCDP` function does not verify if the market is open, while the `mintSCDP` function does. As a result, users are still able to perform operations that reduce the total system debt while they cannot mint more assets.

Recommendation
We recommend not allowing users to perform any market operations if the status is closed.

Status
Acknowledged

Client response
This is intended and in line with the minter side (and liquidations), where we only restrict creating new supply during closed hours.

22. Missing upgradeable initializers

In many upgradeable smart contracts, such as in `KreskoAsset.sol`, `KISS.sol`, and `KreskoAssetAnchor.sol`, the `initialize` function of these contracts does not execute the derived contract's initializers: `PausableUpgradeable`, `AccessControlEnumerableUpgradeable`. As a result, some storage variables will not have the correct value at the proxy initialization.

We have not identified any major issues as a result of this incorrect implementation, but for the sake of completeness, they should be amended.

Recommendation
We recommend executing all derived contract initializers on the parent initializer.

Status
Acknowledged

Client response
There is no need to call those initializers as they do not do anything.

23. Unused return value of low-level `delegatecall` function

In `AssetConfigurationFacet.sol`, the low-level `delegatecall` function success condition is not assessed by the `addAsset` function. As a result, if the executed `setFeedsForTicker` function reverts, the `addAsset` function will still conclude its execution.

Recommendation
We recommend checking that the execution of `delegatecall` is successful.

Status
Resolved

24. `Vault` token spending allowance is incorrectly adjusted, resulting in less than allowed token spending

---

The `redeem` function in the `Vault` contract burns the `owner`'s shares and transfers the corresponding amount of underlying asset tokens, specified as `assetAddr`, to the `receiver`. To ensure the `Vault` contract's balance of the underlying asset is sufficient for the transfer, the `redeem` function determines the balance in line `146` and adjusts the asset transfer amount if necessary.

If an approved spender attempts to redeem the `owner`'s shares, the allowance is checked and adjusted to accommodate the spent shares. However, this allowance adjustment does not consider the adjusted asset transfer amount, resulting in a higher decrease in the allowance amount. Consequently, the spender is unable to spend the full amount of the allowance.

If the `owner` and the spender are contracts, this could lead to unexpected behavior and, in the worst case, result in unrecoverable funds.

Recommendation
We recommend updating the allowance to reflect the adjusted asset transfer amount if the `Vault` contract's token balance is insufficient.

Status
Resolved

25. Incorrect calculation of the seizeAmount value in the SCDPFacet.getMaxLiqValueSCDP helper function due to misplaced arguments

---

The `getMaxLiqValueSCDP` helper function in the `SCDPFacet` contract calculates the maximum liquidation info for the specified `_repayAssetAddr` and `_seizeAssetAddr` token addresses.

One of those values is `seizeAmount`, which determines the amount of collateral tokens that can be seized in return for the repaid debt. This value is calculated in lines `113-116` and involves calling the `valueToAmount` function, which calculates the number of seizable tokens given the repaid debt value.

However, the arguments supplied to the `valueToAmount` function are incorrect. Specifically, the `seizeAssetPrice` and `maxLiqValue` arguments are swapped. As a result, the calculated `seizeAmount` is incorrect, leading to unexpected behavior when used off-chain or in other contracts.

Recommendation
We recommend swapping the `seizeAssetPrice` and `maxLiqValue` arguments in the `valueToAmount` function call.

Status
Resolved

26. Centralization risk

The Kresko Protocol smart contracts have active ownership, which allows privileged addresses to update many important parameters of the system at any given time:

- Change out the protocol-wide state variables

- Change Oracle price feeds and price information
- Pause minting and burning of Kresko Assets and depositing and withdrawal of collateral assets
- Withdraw, burn, and mint assets on behalf of users
- Updating any facets or proxy implementation contracts

Recommendation

We recommend implementing a timelock and multisig wallets to delay-sensitive operations and avoid a single point of key management failure. In the long term, consider renouncing ownership or reconsider the upgradability of contracts that do not need such features.

Status

Acknowledged

Client response

We are determined to launch a successful product. To do this with a novel (and complex) system, we do need the extensive capability to handle any possible scenarios that require modifying deployed code and/or data.

When we do see that the need for training wheels is no longer there and a Kresko DAO exists, we are committing to:

1. Removing all unnecessary mutability.
2. Transfer ownership of all core contracts to the DAO.

Informational Notes

27. Naming conventions

1. In `Errors.sol`, an `IErrorFieldProvider` interface is used to represent ERC-20 tokens containing metadata information, such as the token symbol. This could be renamed to `IERC20Metadata`, in the same way as implemented by the OpenZeppelin contracts library
2. In `Validations.sol`, the `validateCollateralArgs` may revert with `Errors.ELEMENT_DOES_NOT_MATCH_PROVIDED_INDEX`, indicating that the element does not match the index, while the check validates that the index is greater than the array length.
3. In `Modifiers.sol`, the `onlyRoleIf` NatSpec documentation states that this modifier ensures "only trusted contracts can act on behalf of `_account`", while the modifier is generic enough to receive any boolean parameter.
4. In `Assets.sol`, the `marketStatus` function returns `true` if the market is open and false if the market is not open, while the naming implies it returns different options or enumeration values.

Recommendation

We recommend

1. Renaming `IErrorFieldProvider` to `IERC20Metadata` or import this interface from the OpenZeppelin contracts library.
2. Renaming the error to `Errors.ARRAY_INDEX_OUT_OF_BOUNDS`.
3. Updating the modifier name to `onlyRoleIfAccountIsNotMsgSender`.
4. Updating the function name to `isMarketOpen`.

Status

Partially Resolved

28. Usage of low-level transfer function

In `SDICoverRecipient`, the low-level `transfer` function is used, while its usage is discouraged, as it forwards a fixed amount of 2300 gas, as the gas cost of EVM instructions may change and break already deployed contract systems that make fixed assumptions about gas costs.

Recommendation
We recommend using `call` instead of `transfer`.

Status
Acknowledged

29. KreskoAsset allowances get out of sync with rebases

The `KreskoAsset` token manages spending allowances by storing the rebased amounts rather than normalizing these amounts to their unrebased values. This leads to two potential problems:

1. During positive rebases, existing allowances might become insufficient, resulting in the unintended reversal of transactions.
2. During negative rebases, existing allowances might become excessively high, exposing the token owner to the risk of unintended loss of funds by the allowed spender.

Recommendation
We recommend utilizing unrebased amounts for allowances.

Status
Acknowledged

30. Inconsistent revert behavior when a user-provided parameter is greater than internal storage variables

In many instances of the Kresko protocol, some checks are performed to assess if the user-provided parameter is greater than the contract balance, debt, or any other relevant storage variable that is updated by the input.

The issue is that, in some cases, the transaction will revert if the value is greater than the internal storage variable, while in other cases, the value will be capped by it.

For example, in `SDeposits`, the `withdrawSCDP` function calls the `handleWithdrawSCDP` function to calculate the amount of collateral the user can receive, including fees. The issue is that if the requested amount is greater than the user's deposit principal and if there have been no fees accrued, `handleWithdrawSCDP` will revert with `Errors.NOTHING_TO_WITHDRAW`, and they will be unable to withdraw their collateral. This is inconsistent with the case where there are fees, where `amountOut` will be capped by the users' `depositsPrincipal`.

Another example is the `repaySCDP` function in `SCDPFacet.sol`, which reverts if `_repayAmount > repayAssetData.debt`, while the function can take the maximum debt to avoid the execution reverting. This can happen if another transaction that increments the debt (such as a swap) is executed before `SCDPFacet.repaySCDP`.

The function `SCDPFacet.liquidateSCDP` also reverts if the repay amount is greater than the asset debt, which is inconsistent with bounding the value depending on the minimum debt value and maximum liquidation value.

Recommendation
We recommend not reverting if the user-provided value is greater than the said amount and capping it to the amount when necessary, making sure to update the variable name from `amount` to `maxAmount`, or similar.

For example, `liquidateSCDP`'s argument could be renamed from `_repayAmount` to `_maxRepayAmount`, and the implementation updated accordingly.

Status
Acknowledged

31. The protocol returns 0 for the collateralization ratio if the user has no debt

In the `SCDPStateFacet` contract, the collateralization ratio value is 0 if there is no debt. Since the collateralization ratio is derived from the total collateral divided by total debt, this can be misleading since the higher the collateralization ratio, the healthier the SCDP is, and a value of zero is defined for the SCDP with no collateral to back their debt. Although the CR is undefined for an account with no debt, a value representing "infinity" might be more appropriate for some use cases.

Recommendation
We recommend returning `type(uint256).max` for the collateralization ratio of the SCDP with no debt.

Status
Resolved

32. ERC-165 interfaceId's are not self-evident

In `KISS.sol` and other contracts of the Kresko protocol, some ERC-165 `interfaceId`'s are hard coded as their plain `bytes4` value, which makes readability and maintainability harder. In particular, `cast 4byte 0x36372b07`, which fetches function signatures from the 4byte directory database, does not return any results.

Recommendation
We recommend using `type(Contract).interfaceId` whenever possible.

Status
Acknowledged

33. `Vault.maxMint` should assume the agent has infinite assets

In `Vault.sol`, the `maxMint` function is dependent on the user's balance, while in EIP-4626, it says it MUST NOT rely on it. Although the vault is not fully conformant with EIP-4626, it is recommended that the `maxMint` function be adjusted for the sake of consistency. Note: this issue was identified by the invariant tests present in **Test case 1**.

Recommendation
We recommend updating `maxMint` to return the maximum number of shares that can be minted from the `Vault` for the receiver, assuming they have infinite assets.

Status
Acknowledged

34. `SafetyCouncilFacet.safetyStateSet` is not reset to `false` if all assets return their pause state to false

In `SafetyCouncilFacet.sol`, the `toggleAssetsPaused` function sets `safetyState` to `true` if any of the `_assets` are paused during a toggle procedure. The problem is that this function does not reset it to `false` if the assets return to unpaused status, as this must be manually performed via the `setSafetyStateSet` function.

Recommendation
We recommend updating `toggleAssetsPaused` to verify if all possible assets are unpaused in order to disable the `safetyStateSet` storage variable. Alternatively, we recommend documenting the correct behavior and usage of `toggleAssetsPaused` and `setSafetyStateSet`.

Status
Acknowledged

35. `Validations.validateCollateralArgs` reverts with underflow if there are no deposited collateral assets

---

In `Validations.sol`, the `validateCollateralArgs` function will revert with underflow if the `self.depositedCollateralAssets` array length is zero.

Recommendation
We recommend updating the check to `_collateralIndex >= self.depositedCollateralAssets[_account].length` to avoid the underflow.

Status
Acknowledged

Client response
This is the desired behavior.

36. Potential unsafe casting of `toWad` if `_amount` is lower than zero

---

In `Math.sol`, the `toWad` function in lines `49-51` casts a signed 256-bit integer to an unsigned 256-bit integer without any input validation. If the `_amount` value is lower than zero, it will silently "wrap around" and will not revert.

Recommendation
We recommend checking that `_amount` is not lower than zero before casting it to `uint256`.

Status
Resolved

37. `Vault.setGovernance` does not implement a 2-step process for transferring the governance

In `Vault.sol`, the `setGovernance` function does not implement a 2-step process for setting the new governance. As a result, the governance of the contract can easily be lost due to a mistake when the transfer is performed.

Recommendation
We recommend implementing a 2-step process to transfer the governance similar to OpenZeppelin's Ownable2Step contract.

Status
Resolved

38. `Errors.id` and `Errors.symbol` perform external calls to possibly untrusted addresses

In `Errors.sol`, the `id` and `symbol` helper functions perform external calls to the `symbol` function from the `_addr` input. The impact of this depends on whether the outer function properly follows the checks-effects-interactions pattern or if it properly implements `nonReentrant` modifiers.

Although we were not able to verify any major issues in the current implementation, we recommend avoiding making external calls to possibly untrusted user-provided parameters.

Recommendation
We recommend avoiding making external calls to possibly untrusted addresses, making sure to follow the checks-effects-interactions patterns, and implementing reentrancy guards on state-changing functions.

Status
Acknowledged

Client response
These calls are made inside a revert clause, so this should be no problem.

39. Inconsistencies in Vault adaptation of ERC-4626

The `Vault` contract is intended to be an adaptation of ERC-4626: Tokenized Vaults for multiple underlying ERC-20 tokens.

The issue is that the adaptation contains many inconsistencies, which do not pose any major issues for the protocol or integrations but should be revised for better uniformity.

1. The `maxMint` function receives an `address user` argument, while `maxDeposit` does not.
2. The `previewWIthdraw` function reverts if the `sharesIn` parameter is greater than the total supply, while `previewRedeem` does not
3. The `redeem` function caps the user-provided parameter to the vault balance in order to not make the function revert, while `withdraw` does not

Recommendation
We recommend

1. Updating `maxDeposit` to also receive an `address user`
2. Updating both functions to revert if the return value is greater than the total amount, as ERC-4626 states that both functions MAY revert due to conditions that would also cause redeem/withdraw to revert
3. Updating `redeem` so that the `sharesIn` is not adjusted to make the transaction not revert, as the ERC states that "redeem burns exactly `shares` from `owner` and sends `assets` of underlying tokens to `receiver`"

Status
Acknowledged

40. Inconsistent check for minimum collateral ratio and liquidation threshold

In `Validations.sol`, the `validateMinCollateralRatio` function checks in line `54` for `_minCollateralRatio < _liqThreshold` before reverting. However, the `validateLiquidationThreshold` function in line `60` checks for `_liquidationThreshold >= _mionCollateralRatio` before reverting. This means in the former function, the minimum collateral ratio can be equal to the liquidation threshold, while it cannot be equal in the latter function.

Recommendation
We recommend using similar checks in both functions to ensure consistency.

Status
Resolved

41. Possible zero address of state variables

In `Kiss.sol` the `initialize` function configures the `vKiss` variable with an address. However, there is no check to ensure the address is not `address(0),` as the contract has no function to update the variable after initialization. A similar issue exists for the `asset` variable in the constructor for `ERC4626Upgradeable.sol` and the `kresko` variable in the `initialize` function for `KreskoAsset.sol`.

Recommendation
We recommend adding a check to ensure that the parameter values are not `address(0)` before configuring the variables.

Status
Acknowledged

Appendix

As part of our security review, we have implemented a fuzz testing campaign to validate the ERC-4626 properties of the `Vault` contract using the @crytic/properties repository containing Pre-built security properties for common Ethereum operations.

Since we could not find any major issues with the contract implementation related to the ERC-4626 standard, we have included the tests in the appendix as a reference for the Kresko protocol team to improve its test coverage. We recommend including it in the project repository to ensure no regressions are introduced if the contract changes.

Test case 1

```
diff --git a/src/CryticTester.sol b/src/CryticTester.sol
new file mode 100644
index 00000000..dcd8555b
--- /dev/null
+++ b/src/CryticTester.sol
@@ -0,0 +1,195 @@
+pragma solidity ^0.8.0;
+
+// NOTE: See https://github.com/crytic/properties for more information
+// NOTE: See https://github.com/crytic/medusa for installation
instructions
+// NOTE: Run with `$ medusa fuzz`
+
+import {ERC20} from "kresko-lib/token/ERC20.sol";
+import {CryticERC4626PropertyTests} from
"@crytic/properties/contracts/ERC4626/ERC4626PropertyTests.sol";
+// this token _must_ be the vault's underlying asset
+import {TestERC20Token} from
"@crytic/properties/contracts/ERC4626/util/TestERC20Token.sol";
+// change to your vault implementation
+import {VaultAsset} from "./contracts/core/vault/VTypes.sol";
+import {Vault} from "./contracts/core/vault/Vault.sol";
+import {IERC4626} from "@crytic/properties/contracts/util/IERC4626.sol";
+import {IERC20} from "kresko-lib/token/IERC20.sol";
+import {IAggregatorV3} from "kresko-lib/vendor/IAggregatorV3.sol";
+
```

```
+contract Oracle is IAggregatorV3 {
+    function latestRoundData()
+        public
+        view
+        returns (uint80 roundId, int256 answer, uint256 startedAt, uint256
updatedAt, uint80 answeredInRound)
+    {
+        return (42, 1e18, block.timestamp, block.timestamp, 42);
+    }
+
+    function decimals() external view returns (uint8) {
+        return 18;
+    }
+
+    function description() external view returns (string memory) {
+        return "Oracle";
+    }
+
+    function version() external view returns (uint256) {
+        return 1000;
+    }
+
+    function getRoundData(
+        uint80
+    ) external view returns (uint80 roundId, int256 answer, uint256
startedAt, uint256 updatedAt, uint80 answeredInRound) {
+        return latestRoundData();
+    }
+}
+
+contract VaultHarness is Vault {
+    address public _asset;
+
+    constructor(
+        address __asset,
+        string memory _name,
+        string memory _symbol,
+        uint8 _decimals,
+        uint8 _oracleDecimals,
```

```
+        address _feeRecipient,
+        address _sequencerUptimeFeed
+    ) Vault(_name, _symbol, _decimals, _oracleDecimals, _feeRecipient,
_sequencerUptimeFeed) {
+        _asset = __asset;
+    }
+
+    function asset() external view returns (IERC20) {
+        return IERC20(_asset);
+    }
+
+    function deposit(uint256 assets, address receiver) external returns
(uint256 shares) {
+        (shares, ) = deposit(_asset, assets, receiver);
+    }
+
+    function mint(uint256 shares, address receiver) external returns
(uint256 assets) {
+        (assets, ) = mint(_asset, shares, receiver);
+    }
+
+    function withdraw(uint256 assets, address receiver, address owner)
external returns (uint256 shares) {
+        (shares, ) = withdraw(_asset, assets, receiver, owner);
+    }
+
+    function redeem(uint256 shares, address receiver, address owner)
external returns (uint256 assets) {
+        (assets, ) = redeem(_asset, shares, receiver, owner);
+    }
+
+    function convertToShares(uint256 assets) external view returns
(uint256 shares) {
+        (shares, ) = previewDeposit(_asset, assets);
+    }
+
+    function convertToAssets(uint256 shares) external view returns
(uint256 assets) {
+        (assets, ) = previewRedeem(_asset, shares);
```

```
+    }
+
+    function previewDeposit(uint256 assets) external view returns (uint256
shares) {
+        (shares, ) = previewDeposit(_asset, assets);
+    }
+
+    function previewMint(uint256 shares) external view returns (uint256
assets) {
+        (assets, ) = previewMint(_asset, shares);
+    }
+
+    function previewWithdraw(uint256 assets) external view returns
(uint256 shares) {
+        (shares, ) = previewWithdraw(_asset, assets);
+    }
+
+    function previewRedeem(uint256 shares) external view returns (uint256
assets) {
+        (assets, ) = previewRedeem(_asset, shares);
+    }
+
+    function maxDeposit(address user) public view returns (uint256) {
+        // NOTE: assumes Vault.maxDeposit has been updated to receive
additional `user` parameter
+        return maxDeposit(_asset, user);
+    }
+
+    function maxMint(address user) external view returns (uint256 max) {
+        uint256 balance = IERC20(_asset).balanceOf(user);
+        uint256 depositLimit = maxDeposit(_asset);
+        if (balance > depositLimit) {
+            (max, ) = previewDeposit(_asset, depositLimit);
+        } else {
+            (max, ) = previewDeposit(_asset, balance);
+        }
+    }
+
+    function maxWithdraw(address owner) external view returns (uint256
```

```
max) {
+         (max, ) = previewRedeem(_asset, maxRedeem(_asset, owner));
+     }
+
+     function maxRedeem(address owner) external view returns (uint256) {
+         return maxRedeem(_asset, owner);
+     }
+}
+
+contract CryticTester is CryticERC4626PropertyTests {
+     constructor() {
+         TestERC20Token _asset = new TestERC20Token("Test Token", "TT",
18);
+         VaultHarness _vault = new VaultHarness(
+             address(_asset),
+             _asset.name(),
+             _asset.symbol(),
+             uint8(_asset.decimals()),
+             uint8(_asset.decimals()),
+             address(this),
+             address(0)
+         );
+         Oracle oracle = new Oracle();
+         VaultAsset memory config = VaultAsset({
+             token: IERC20(address(_asset)),
+             feed: IAggregatorV3(oracle),
+             staleTime: 0,
+             decimals: uint8(_asset.decimals()),
+             depositFee: 0,
+             withdrawFee: 0,
+             maxDeposits: 1337e18,
+             enabled: true
+         });
+         _vault.addAsset(config);
+
+         initialize(address(_vault), address(_asset), false);
+     }
+}
+
```

```
+/*
+$ medusa fuzz
+
+→ [FAILED] Assertion Test:
CryticTester.verify_maxMintIgnoresSenderAssets(uint256)
+Test for method "CryticTester.verify_maxMintIgnoresSenderAssets(uint256)"
resulted in an assertion failure after the following call sequence:
+[Call Sequence]
+1)
CryticTester.verify_maxMintIgnoresSenderAssets(106017213816649585183439734646412349227237308612656573918588622174545151509758) (block=69, time=441900,
gas=12500000, gasprice=1, value=0,
sender=0x0000000000000000000000000000000000030000)
+[Execution Trace]
+ => [call]
CryticTester.verify_maxMintIgnoresSenderAssets(106017213816649585183439734646412349227237308612656573918588622174545151509758)
(addr=0xA647ff3c36cFab592509E13860ab8c4F28781a66, value=0,
sender=0x0000000000000000000000000000000000030000)
+      => [call]
VaultHarness.maxMint(0xa647ff3c36cfab592509e13860ab8c4f28781a66)
(addr=0x9491F0Dfb965BC45570dd449801432599F0542a0, value=<nil>,
sender=0xA647ff3c36cFab592509E13860ab8c4F28781a66)
+            => [call]
Vault.maxMint(0x54919a19522ce7c842e25735a9cfecef1c0a06da,
0xa647ff3c36cfab592509e13860ab8c4f28781a66)
(addr=0x01375317AA980daaBF22f990a378ECCaD9B40dc0, value=<nil>,
sender=0x9491F0Dfb965BC45570dd449801432599F0542a0)
+                  => [call]
TestERC20Token.balanceOf(0xa647ff3c36cfab592509e13860ab8c4f28781a66)
(addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                        => [return (0)]
+                  => [call]
TestERC20Token.balanceOf(0x01375317aa980daabf22f990a378eccad9b40dc0)
(addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                        => [return (0)]
+                  => [call]
```

```
TestERC20Token.balanceOf(0x01375317aa980daabf22f990a378eccad9b40dc0)
(addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                        => [return (0)]
+                    => [call] Oracle.latestRoundData()
(addr=0xFC11D02bd1b627683935C7cD70B328Df9AF6275d, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                        => [return (42, 178900000000000000000, 441900,
441900, 42)]
+                    => [return (0)]
+            => [return (0)]
+      => [call]
TestERC20Token.mint(0xa647ff3c36cfab592509e13860ab8c4f28781a66,
106017213816649585183439734646412349227237308612656573918588622174545151509
758) (addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=0,
sender=0xA647ff3c36cFab592509E13860ab8c4F28781a66)
+            => [event]
Transfer(0x0000000000000000000000000000000000000000,
0xa647ff3c36cfab592509e13860ab8c4f28781a66,
106017213816649585183439734646412349227237308612656573918588622174545151509
758)
+            => [return ()]
+      => [call]
VaultHarness.maxMint(0xa647ff3c36cfab592509e13860ab8c4f28781a66)
(addr=0x9491F0Dfb965BC45570dd449801432599F0542a0, value=<nil>,
sender=0xA647ff3c36cFab592509E13860ab8c4F28781a66)
+            => [call]
Vault.maxMint(0x54919a19522ce7c842e25735a9cfecef1c0a06da,
0xa647ff3c36cfab592509e13860ab8c4f28781a66)
(addr=0x01375317AA980daaBF22f990a378ECCaD9B40dc0, value=<nil>,
sender=0x9491F0Dfb965BC45570dd449801432599F0542a0)
+                    => [call]
TestERC20Token.balanceOf(0xa647ff3c36cfab592509e13860ab8c4f28781a66)
(addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                        => [return
(106017213816649585183439734646412349227237308612656573918588622174545151509
758)]
+                    => [call]
```

```
TestERC20Token.balanceOf(0x01375317aa980daabf22f990a378eccad9b40dc0)
(addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                         => [return (0)]
+                   => [call]
TestERC20Token.balanceOf(0x01375317aa980daabf22f990a378eccad9b40dc0)
(addr=0x54919A19522Ce7c842E25735a9cFEcef1c0a06dA, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                         => [return (0)]
+                   => [call] Oracle.latestRoundData()
(addr=0xFC11D02bd1b627683935C7cD70B328Df9AF6275d, value=<nil>,
sender=0x01375317AA980daaBF22f990a378ECCaD9B40dc0)
+                         => [return (42, 178900000000000000000, 441900,
441900, 42)]
+                   => [return (19852711900000000000000000)]
+             => [return (19852711900000000000000000)]
+      => [event] AssertEqFail("Invalid: 0!=19852711900000000000000000,
reason: maxMint must assume the agent has infinite assets")
+      => [panic: assertion failed]
+*/
```

Audit Report for Kresko Labs Pte. Ltd - January 12, 2024

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Kresko Labs Pte. Ltd or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Oak Security GmbH