SOLIDIFIED

Audit Report for Kresko Labs Pte. Ltd - July 14, 2023

Summary

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

Process and Delivery

Independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on June 16, 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: `cc5ae9bbb7eab7a1c88ddfc4fce3d538d45e729c`

File list:

```
├── src/contracts
│   ├── diamond/*
│   ├── kiss/*
│   ├── kreskoasset/*
│   ├── libs/*
│   ├── minter
│   │   ├── amm-oracle/*
│   │   ├── facets/*
│   │   ├── interfaces/*
│   │   ├── libs/*
│   │   ├── InterestRateState.sol
│   │   ├── MinterModifiers.sol
│   │   ├── MinterStorage.sol
│   │   └── MinterTypes.sol
│   ├── staking
│   │   ├── KrStaking.sol
│   │   └── KrStakingHelper.sol
│   ├── vendor
│   │   ├── flux
│   │   │   ├── FluxPriceFeed.sol
│   │   │   └── FluxPriceFeedFactory.sol
```

Intended Behavior

The contracts implement a non-custodial, capital-efficient synthetic asset protocol that runs on the EVM. It facilitates the creation and management of securely collateralized synthetic assets using smart contracts written in Solidity. Users can deposit various Collateral Assets that have their value combined, enabling users to borrow synthetic assets referred to as Kresko Assets in an overcollateralized fashion. Users can also participate in the protocol by performing liquidations on unhealthy debt positions.

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	High	The usage of the ERC-2535: Diamonds, Multi-Facet Proxy standard imposes additional overhead to codebase management that should be considered. Specifically, global libraries and variables must be employed to ensure functions are correctly shared between facets, often necessitating the opening of multiple files to follow a single code path. Because of the decomposition of functions into various facets and libraries, editors' "Go to definition" shortcuts may not function as anticipated. Consequently, it becomes necessary to manually search for or find the correct function to continue analyzing a particular path. Moreover, the tooling available for the standard is still

		maturing, which can cause problems, such as issues with code coverage visualization. Due to the complex proxy upgradability pattern, additional care must be taken to ensure proxies function as intended. This includes implementing Diamond-specific tests, scripts, and helper functions, which add up to the time spent in development, revision, and maintenance. Lastly, some adaptation is required in order to extend or integrate with third-party protocols, such as with Redstone, requiring changes from the default implementation guidelines.
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 such that it should be easy to onboard new contributors
Level of Documentation	High	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 White Paper is not entirely synchronized with the latest updates in the documentation and the codebase.

Test Coverage	High	Due to an error with the hardhat-coverage plugin related to the use of the Diamond contract, it was not possible to verify the test coverage of the protocol. Nevertheless, based on our subjective assessment, we consider the test coverage to be high.

Issues Found

Solidified found that the Kresko Labs contracts contain 3 critical issues, 17 major issues, 34 minor issues, and 30 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	Unlimited Kresko asset minting due to faulty account liquidation	Critical	Resolved
2	Unlimited Kresko asset minting due to not considering the accrued interest when burning assets	Critical	Resolved
3	LibRedstone contains signerAddress associated with a public hardhat account	Critical	Acknowledged
4	KreskoAssetAnchor accounting differences between issue/destroy and deposit/mint/withdraw/redeem break system invariants	Major	Resolved
5	Kresko asset rebase can cause precision loss and leads to the ability to inflate collateral	Major	Resolved
6	Minting and burning Kresko assets or fully repaying interest before principal repayment within the same transaction leads to inflated debt accounting	Major	Resolved
7	Batch liquidation of accrued interest reverts when principal asset debt is zero	Major	Resolved
8	Incorrect total pool reward allocation calculation in KrStaking contract	Major	Resolved
9	Chainlink Oracle return values are not validated properly	Major	Resolved

10	Flux Oracle return values are not validated properly	Major	Acknowledged
11	Users are unable to repay debt or deposit collateral while the protocol is paused	Major	Acknowledged
12	LibCalculation does not use updated maxLiquidationMultiplier value	Major	Resolved
13	Liquidations can leave a position below the minimum debt value	Major	Acknowledged
14	Incorrect assumption on WadRay library can lead to inconsistent MinterState collateral assets array	Major	Resolved
15	UniswapV2Oracle is vulnerable to price manipulation attacks	Major	Acknowledged
16	Users can sweep exceeding liquidity from the KrStakingHelper contract	Major	Acknowledged
17	LibAssetUtility price aggregation averages out oracles price data in case they deviate	Major	Resolved
18	Users can prevent liquidations by frontrunning the transaction and slightly increasing their collateral	Major	Resolved
19	Users are unable to specify the maximum price they are willing to pay	Major	Acknowledged
20	Liquidator will pay more debt than necessary if the account does not have enough collateral to back their deposits	Major	Acknowledged
21	Ineffective safety council controls on user actions	Minor	Resolved
22	KISS stablecoin token is not pausable	Minor	Resolved
23	DepositWithdrawFacet.withdrawCollateralUnchecked is vulnerable to reentrancy attacks	Minor	Acknowledged

24	Authorization.transferSecurityCouncil does not properly validate if msg.sender has SAFETY_COUNCIL role	Minor	Resolved
25	LibRedstone returns the same index for all authorized signers	Minor	Resolved
26	LibAssetUtility lacks an L2 Sequencer Uptime Feed check	Minor	Resolved
27	Unbounded loops over arrays	Minor	Acknowledged
28	Unsafe type casting from signed to unsigned integer in LibDecimals	Minor	Resolved
29	Wrong MAX_MIN_DEBT_VALUE defined in Constants library	Minor	Resolved
30	Redstone timestamp validation is disabled	Minor	Acknowledged
31	Use of Solidity's transfer() function might render ETH impossible to withdraw	Minor	Resolved
32	Unnecessary payable specifier for functions may allow ETH to be sent and locked	Minor	Acknowledged
33	KrStaking operator can withdraw and steal user funds	Minor	Acknowledged
34	Emergency withdrawal in KrStaking contract executes additional, unnecessary logic, potentially preventing withdrawals	Minor	Resolved
35	EIP712_DOMAIN_TYPEHASH uses salt instead of chainId	Minor	Resolved
36	UniswapV2Oracle uses potentially outdated incentive amount	Minor	Acknowledged
37	Unexpected behavior if price Oracle with decimals other than 8 are used	Minor	Acknowledged
38	Centralization risk	Minor	Acknowledged

39	LibDiamondCut does not validate if the facet address parameter is the Diamond address	Minor	Acknowledged
40	Optimism does not support PUSH0 yet	Minor	Acknowledged
41	KISS code is not production ready	Minor	Resolved
42	KISS.supportsInterface does not return true for inherited contracts' interfaces	Minor	Resolved
43	KISS.grantRole overrides the _to function parameter in some cases, which can be confusing to users	Minor	Acknowledged
44	KISS.setMaxOperators does not validate if the current validator count is greater than the updated maximum	Minor	Acknowledged
45	bytes32 role values are not consistent across contracts	Minor	Acknowledged
46	Authorization.setupSecurityCouncil notice suggests performing ERC165 validation, but this is not done	Minor	Acknowledged
47	Loss of precision due to division before multiplication	Minor	Resolved
48	LibUI.krAssetInfoFor ignores krFactor in the debt calculation	Minor	Acknowledged
49	Some LibUI functions always return zero due to a typo in increment/assignment operator	Minor	Acknowledged
50	Missing input validations	Minor	Resolved
51	FluxPriceFeed getters' behavior is inconsistent	Minor	Acknowledged
52	LibCalculation._getMaxLiquidatableUSD can revert with an underflow depending on specific configuration values	Minor	Acknowledged
53	Diamond.sol: Inability to retrieve sent native tokens	Minor	Resolved

54	Authorization.sol: No check to ensure multisig has sufficient owners during setup	Minor	Acknowledged
55	LibUI.sol: Possible array index out of bounds	Note	Acknowledged
56	KrStaking.sol: Unnecessary swapping of array values	Note	Resolved
57	ConfigurationFacet.sol: Possible gas savings	Note	Resolved
58	Potentially misleading SafetyStateChange event description	Note	Resolved
59	Asset pause duration is unused	Note	Acknowledged
60	KreskoAsset allowances get out of sync with rebases	Note	Acknowledged
61	Misleading NewOperator event operator value	Note	Resolved
62	ReentrancyGuardUpgradeable contract is not properly initialized in the KrStaking contract	Note	Resolved
63	Code simplification	Note	Acknowledged
64	Code repetition	Note	Resolved
65	Missing events in important functions	Note	Resolved
66	Usage of tx.origin	Note	Resolved
67	The protocol returns 0 for the collateralization ratio if the user has no debt	Note	Acknowledged
68	Typos	Note	Resolved
69	Unclear documentation	Note	Resolved
70	Naming conventions	Note	Acknowledged
71	DiamondOwnershipFacet.sol: Redundant check for owner and pending owner	Note	Resolved
72	LibDiamondCut.sol: Unnecessary use of require statement	Note	Acknowledged

73	KISS.sol: Unused constant role variable	Note	Resolved
74	KISS.sol: Redundant initialization of variable	Note	Resolved
75	ERC4626Upgradeable.sol: Incorrect comment	Note	Resolved
76	KreskoAsset.sol: Redundant check for isRebase	Note	Resolved
77	Arrays.sol: Misleading revert statement	Note	Resolved
78	Authorization.sol: Redundant removal of the account from role members	Note	Resolved
79	AccountStateFacet.sol: Redundant declaration of library use	Note	Resolved
80	BurnFacet.sol: Possible zero burn	Note	Resolved
81	StabilityRateFacet.sol: Partial repayment of stability rate interest not enforced	Note	Acknowledged
82	ERC-165 interfaceIds are not self-evident	Note	Acknowledged
83	Kresko protocol does not support fee-on-transfer collateral tokens	Note	Resolved
84	Kresko protocol does not support native cryptocurrencies as collateral assets	Note	Resolved

Critical Issues

1. Unlimited Kresko asset minting due to faulty account liquidation

When fully liquidating an account by repaying all principal debt, the asset is removed from the `mintedKreskoAssets` array in the `_liquidateAssets` function of the `LiquidationFacet` contract due to incorrectly assuming the account's debt is zero.

However, if an account still has unpaid interest, with `irs().srUserInfo[params.account][params.repayAsset].debtScaled` being non-zero, this results in the ability to mint debt without the possibility of future liquidation. This is caused by not adding the asset to the `mintedKreskoAssets` array in the `MintFacet` contract's `mintKreskoAsset` function, as the position still has debt, specifically unpaid interest.

This vulnerability allows an attacker to intentionally establish a faulty account and mint debt without the risk of liquidation. Additionally, the attacker can withdraw the deposited collateral.

The issue arises not only from removing the Kresko asset from the `mintedKreskoAssets` array but also from the `LibAccount.getAccountKrAssetValue` function, which mistakenly determines the account debt as `0` due to iterating over an empty `mintedKreskoAssets` array. This function is widely employed to calculate an account's debt.

For a test case, refer to **Test case 1** in the **Appendix**.

Recommendation

We recommend removing the Kresko asset within the `_liquidateAssets` function in line 145 of the `LiquidationFacet` contract only if the asset's debt, including the stability rate interest, is zero.

Status
Resolved

2. Unlimited Kresko asset minting due to not considering the accrued interest when burning assets

---

When burning Kresko assets with the `burnKreskoAsset` function in the `BurnFacet` contract and the account repaying all principal debt while having interest accrued, the asset is mistakenly removed from the `mintedKreskoAssets` array.

Consequently, the account is able to mint unlimited Kresko assets as the `mintKreskoAsset` function in the `MintFacet` contract does not add the asset to the `mintedKreskoAssets` array again. Resulting in zero debt for this account due to iterating over an empty `mintedKreskoAssets` array.

This issue has been disclosed by the Kresko Labs team to Solidified shortly after delivering the draft audit report.

Recommendation
We recommend removing the Kresko asset within the `burnKreskoAsset` function in line 53 of the `BurnFacet` contract only if the asset's debt, including the stability rate interest, is zero.

Status
Resolved

3. LibRedstone contains signerAddress associated with a public hardhat account

---

In `src/contracts/minter/libs/LibRedstone.sol:82`, the address `0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266`, associated with Hardhat's local network deployer account, is used as an accepted signer address. Since its private key is known by

anyone, an attacker could use it to post invalid price data to the protocol and disrupt its correct functioning.

Recommendation
We recommend removing Hardhat's `Account #0` from the list of accepted signers. Additionally, it is recommended to refactor the testing framework to not require test parameters to be included in production code by using mock contracts.

Status
Acknowledged. Team's response: *"We will remove the signer for mainnet deployment"*.

Major Issues

4. `KreskoAssetAnchor` accounting differences between `issue`/`destroy` and `deposit`/`mint`/`withdraw`/`redeem` break system invariants

In `src/contracts/kreskoasset/ERC4626Upgradeable.sol:85` and line `106`, the ERC4626 interface is extended with the `issue` and `destroy` functions in order to support Kresko non-rebasing wrapper tokens (Kresko Asset Anchor). This contract is meant to be an ERC-4626 that handles the pro-rata representation of Kresko assets in order to allow easier integration for the rebasing KreskoAsset with other protocols and chains.

The `issue` function (called by `LibMint` via `MintFacet`) mints ERC4262 shares to the Kresko Asset contract and mints Kresko Assets to the receiver in order to keep a balance between krAssets and ERC4626 shares (it increases both the `totalSupply` and the `totalAssets` state variables of this vault). Conversely, the `destroy` function (called by `LibBurn` via `BurnFacet`) burns shares from the Kresko Asset contract and Kresko Assets from the depositor account (it decreases both the `totalSupply` and `totalAssets` state variables from this vault).

The problem is that the vault still allows users to `deposit`, `mint`, `withdraw` or `redeem` directly to the vault without going through the Diamond proxy. This will break system invariants since depositing, minting, withdrawing, and redeeming only increase/decrease `totalSupply`, not `totalAssets`. Because of that, if a user tries to deposit and then withdraw (or analogously mint and redeem) issued anchor assets, they will receive fewer assets than what they had initially. This is because calling the `issue` and then `deposit`/`mint` functions will have `totalSupply` increased twice, while `totalAssets` only once.

Giving more detail as to why this issue happens, the problem is that `issue` and `destroy` always increases `assets`/`shares` proportionally and does not give users any shares in return, but instead mint them to the Kresko address, while `deposit` and `withdraw` will increase shares supply and give back shares to users. So when a user mistakenly goes through the Vault directly, they will have increased the supply of shares, and when trying to withdraw, they will get fewer assets in return. On the other hand, users who operated only through the Diamond still have only `issued` assets, and for them, it does not matter if the supply of the shares changed since `destroy` will burn assets from users and shares from the Kresko address proportionally, even if shares supply changed, as this is the intended behavior to correctly handle rebases.

Proof of Concept

Suppose a user issues 10 assets, then deposits those 10 assets on the Vault, receiving 10 shares in return. If later they try to withdraw their assets, they will receive only 5 tokens.

	Action	user	Kresko	vault	totalAssets, totalSupply
1	issue	10,0	0,10	0,0	10,10
2	deposit	0,10	0,10	10,0	10,20
3	withdraw	5,0	0,10	5,0	10,10

For other scenarios, review the following table:

	Action	user1	user2	user3	Kresko	vault	totalAssets, totalSupply
1	issue1	10,0	0,0	0,0	0,10	0,0	10,10
2	issue2	10,0	10,0	0,0	0,20	0,0	20,20

3	deposit1	0,10	10,0	0,0	0,20	10,0	20,30
	case						
4.a	withdraw1	6,0	10,0	0,0	0,20	4,0	20,20
4.b	destroy2	0,10	0,0	0,0	0,5	10,0	10,15
4.c	issue3	0,10	0,0	10,0	0,35	10,0	30,45

For a test case, refer to **Test case 2** in the **Appendix**.

Recommendation

We recommend not allowing users to directly interact with the `deposit`, `mint`, `withdraw`, and `redeem` functions, and instead make sure that all shares issuance and destruction are performed through the Diamond since the ERC4626 vault is meant to only handle the pro-rata representation of Kresko Assets.

Status

Resolved

5. Kresko asset rebase can cause precision loss and leads to the ability to inflate collateral

---

If there has been a positive rebase for a Kresko asset that's used as collateral, an attacker is able to inflate the deposited collateral and steal minted Kresko assets.

This is possible by withdrawing everything but a small dust amount of the collateral, resulting in the call to `toNonRebasingAmount` in the `verifyAndRecordCollateralWithdrawal` function of the `LibCollateral` contract to round down to zero. This leads to `self.collateralDeposits[_account][_collateralAsset]` being set to zero without removing the address of the deposited collateral from `depositedCollateralAssets` in line 105.

Subsequently depositing the same collateral asset again will then add a duplicate entry to the `depositedCollateralAssets` array in line 133, as the amount of deposited collateral has been

set to zero. This results in an overestimated collateral value, as the same collateral is factored in multiple times and allows the attacker to mint Kresko assets without supplying additional collateral.

For a test case, refer to **Test case 3** in the **Appendix**.

Recommendation
We recommend preventing withdrawals that lead to a dust amount of collateral. This can be achieved by enforcing a minimum collateral amount for accounts.
Status
Resolved

6. Minting and burning Kresko assets or fully repaying interest before principal repayment within the same transaction leads to inflated debt accounting

---

When a Kresko asset with an assigned stability rate is minted and subsequently burned by the CDP owner within the same transaction, avoiding interest to be accrued, a duplicate `_kreskoAsset` entry is added to the `mintedKreskoAssets` array the next time the user mints the same Kresko asset. Determining the existing debt of the CDP for this Kresko asset in the `mintKreskoAsset` function of the `MintFacet` contract will therefore return zero, causing the asset to be erroneously added to the `mintedKreskoAssets` array in line 78.

This issue is caused by the `burnKreskoAsset` function of the `BurnFacet` contract, as the `_kreskoAsset` is not removed from the array if a stability rate is assigned.

The occurrence of duplicate entries in the `mintedKreskoAssets` array results in an overestimated CDP debt, given that the same debt is factored in multiple times. The CDP remains in this flawed state indefinitely, as the duplicate entries cannot be removed. Possibly resulting in the CDP being liquidated prematurely.

Similarly, if the CDP owner first fully repays the stability rate interest before repaying the principal debt, Subsequently minting the same Kresko asset again will add a duplicate entry to the `mintedKreskoAssets` array.

For a test case, refer to **Test case 4** in the **Appendix**.

Recommendation

We recommend only adding the `_kreskoAsset` to the `mintedKreskoAssets` array in the `mintKreskoAsset` function of the `MintFacet` contract if it is not already present.

Status

Resolved

7. Batch liquidation of accrued interest reverts when principal asset debt is zero

---

Accrued interest can be liquidated in batches with the `batchLiquidateInterest` function of the `InterestLiquidationFacet` contract. This function iterates over the `mintedKreskoAssets` array of the liquidatable CDP account, fully repaying the accrued stability rate interest. If the principal debt is zero, it attempts to remove the Kresko asset from the array.

However, the `repayFullStabilityRateInterest` function has already removed the Kresko asset from the array. Attempting to remove the asset from the array again will cause the `Arrays.removeAddress` function to revert due to the asset not being present in the array anymore.

Recommendation

We recommend removing the check if the Kresko asset's principal debt is zero and subsequently removing the Kresko asset from the `mintedKreskoAssets` array in the `batchLiquidateInterest` function.

Status
Resolved

8. Incorrect total pool reward allocation calculation in `KrStaking` contract

The `KrStaking` contract allows the operator to update the staking reward allocations (weight) for a specific pool using the `setPool` function. In addition to updating the allocation for the given pool, the `totalAllocPoint` value, which is utilized to determine the reward distribution, is also updated.

`totalAllocPoint` is calculated by adding the old pool allocation to the new allocation and then subtracting the result from the total allocation. However, this will result in incorrect results or underflow errors due to the non-associative nature of a subtraction.

For instance, assume a single pool has a current allocation of 100, resulting in a total allocation of 100. Updating the allocation to 200 would result in the following calculation: `totalAllocPoint -= 100 + 200`, which causes an underflow error by subtracting the result of the addition.

Recommendation
We recommend adapting the calculation of `totalAllocPoint` by first subtracting the old pool allocation and then adding the new allocation.

Status
Resolved

9. Chainlink Oracle return values are not validated properly

The `LibAssetUtility` library includes various functions utilizing Chainlink as an oracle to obtain the current price of an asset. However, the current implementation uses Chainlink's

`latestAnswer` function, which is deprecated and should no longer be used. Furthermore, the returned price is not validated, allowing for negative or stale prices to be returned.

Recommendation
We recommend using the `latestRoundData` function instead of the `latestAnswer` function and validating the returned data.

For further reference, please refer to the following article:
https://0xmacro.com/blog/how-to-consume-chainlink-price-feeds-safely

Status
Resolved

10. Flux Oracle return values are not validated properly

Certain Kresko assets, such as synthetic stock market assets, are mintable only during active market hours. To determine if the market is open, a custom Flux oracle operated by the Kresko team is used. However, checking the market open status with the `latestMarketOpen` function does not validate the current round data, allowing for stale data to be returned. This flaw is akin to the deprecated Chainlink `latestAnswer` function.

Recommendation
We recommend validating the current round data within the `FluxPriceFeed` oracle to prevent stale data from being returned.

Status
Acknowledged. Team's response: *"We will use Redstone values for market status in production. Their custom implementation on the market status data is pending"*.

11. Users are unable to repay debt or deposit collateral while the protocol is paused

While the protocol is paused, the collateral is frozen, and users accumulate interest. In addition, since they cannot add any additional collateral to their loans, their positions may end up being underwater by the time the protocol is unpaused, either through price changes or interest accrual.

In addition, in `src/contracts/minter/facets/LiquidationFacet.sol:36` and `src/contracts/minter/facets/BurnFacet.sol:35`, it is possible that repayments are paused, but liquidations are enabled. This would unfairly prevent borrowers from making their repayments while still allowing them to be liquidated. In addition, this contradicts the Gitbook documentation, which states that the `SafetyCouncilFacet` `Allows the safety council (multisig of 5) to toggle a pause status on each core functionality (Deposit/Withdraw/Repay/Borrow/Liquidation).`

Recommendation
We recommend allowing users to add more collateral or repay their debt when the protocol is paused and not accruing interest when the protocol is paused, in addition to pausing liquidations whenever repayments are paused.

Status
Acknowledged. Team's response: *"In the case of any safety state being enabled the possibility of liquidations happening are weighted in".*

12. `LibCalculation` does not use updated `maxLiquidationMultiplier` value

In `src/contracts/minter/libs/LibCalculation.sol:191`, the default maximum liquidation multiplier is assigned and returned to the `MaxLiquidationVars` intermediate variable, which in turn is used to calculate the maximum liquidation value that can be liquidated for a liquidation pair.

The problem is that `state.maxLiquidationMultiplier` might have already changed in `src/contracts/minter/facets/ConfigurationFacet.sol:143`. More specifically, from the

internal documentation from the Kresko team, these values will differ between the Stable Market (100.01%) and Volatile Market (101%). As a result, liquidators will receive less than expected.

Recommendation
We recommend using `state.maxLiquidationMultiplier` instead of `Constants.MIN_MAX_LIQUIDATION_MULTIPLIER`.

Status
Resolved

13. Liquidations can leave a position below the minimum debt value

---

In `src/contracts/minter/facets/LiquidationFacet.sol:76`, the liquidation repayment amount in USD is limited up to the maximum liquidatable value in USD. The issue is that the `liquidate` function never checks if the position will be left below the minimum debt value after this repayment. In line `130`, the function `_liquidateAssets` simply deducts the destroyed repayment amount from the liquidated account, and liquidate moves on to transferring the seized collateral to the liquidator.

Suppose the maximum liquidatable value is $10, the minimum debt value is $10, and repay amount is $9. Since the repayment amount is lower than the maximum liquidatable value, the `require` check will pass, and the position will be left with $1, below the minimum debt value.

Recommendation
We recommend validating that the position 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
Acknowledged. Team's response: *"It is not possible to deterministically create tons of dust positions this way."*.

14. Incorrect assumption on WadRay library can lead to inconsistent MinterState collateral assets array

In `src/contracts/minter/libs/LibCalculation.sol:108`, the `calcFee` function derives a "proof" to ensure that `transferAmount` is lower than the `depositValue`. The "proof" starts with the statement `depositValue <= oraclePrice * depositAmount`, commenting that the inequality should be "lower than or equal to" due to a potential loss of precision.

The problem is that this assumption incorrectly assumes the `WadRay` library always rounds down.

In fact, the NatSpec states that both `wadDiv` and `wadMul` perform a "rounding half up to the nearest WAD", and thus it is possible that dividing two WAD amounts results in a WAD that is higher than the multiplication.

Specifically, in `src/contracts/minter/libs/LibCollateral.sol:64`, `depositValue` results from multiplying the WAD values `oraclePrice` and `depositAmount`. However, this does not necessarily mean that `depositValue` is `<=` the multiplication of their non-wad values `oraclePrice` and `depositAmount`.

For example, consider the scenario where oraclePrice is `6666666666666666666`, `depositAmount` is `3`. This means that `wadMul(6666666666666666666, 3) = 20`, while `uint(6666666666666666666*3)/1e18 = 19`.

This incorrect assumption means `transferAmount` can be greater than or equal to `depositAmount` due to rounding to the nearest wad. If `transferAmount` is greater than `depositAmount,` line `120` will revert. If it equals `depositAmount`, `depositedCollateralAssets` will be left with an existing address even though the deposit has been zeroed out (due to the lack of `removeAddress`, which is applied only in line `128`). This would break an important invariant in the system, that `depositedCollateralAssets` should

only contain addresses if `collateralDeposits` is not zero, which can, in turn, make subsequent deposits duplicate the collateral address in the array.

We classify this issue as major, since, despite its high impact, it depends on specific oracle prices, fee value, and deposit amounts.

Recommendation
We recommend checking if the `transferAmount` is equal to the `depositAmount` and executing `removeAddress` so that the deposited collateral is removed from the minter state. Additionally, consider using invariant testing so that off-by-one and rounding errors have a higher probability of being caught by fuzzers.

Status
Resolved

15. `UniswapV2Oracle` is vulnerable to price manipulation attacks

In `src/contracts/minter/libs/LibStabilityRate.sol:71`, the `getPriceRate` function will get the current price rate between AMM and Oracle pricing. This is susceptible to price manipulation attacks, as anyone is able to add or remove liquidity to the AMM in order to change the `UniswapV2Oracle` TWAP.

Although using a Time-weighted average price oracle makes price manipulation attacks harder, pools with low liquidity may still be susceptible if an attacker has enough capital.

Recommendation
We recommend closely monitoring the discrepancy between the AMM price and the Oracle price, as well as the capital required for a TWAP manipulation attack. Incorporate suitable boundaries and checks into the system to ensure its resilience in adverse market conditions. Consider changing the stability rate mechanism to not depend on the difference between the AMM price and the Oracle price.

Status

Acknowledged. Team's response: *"Assets that have stability rate enabled should have their configuration adjusted according to the liquidity available"*.

16. Users can sweep exceeding liquidity from the `KrStakingHelper` contract

---

`KrStakingHelper` is a smart contract that contains helper functions allowing users to add liquidity to a pair and deposit liquidity tokens to staking (`addLiquidityAndStake`), to withdraw liquidity tokens from staking and remove the underlying (`withdrawAndRemoveLiquidity`), as well as claiming rewards from each pool (`claimRewardsMulti`).

When withdrawing liquidity, in `src/contracts/staking/KrStakingHelper.sol:114`, users can pass an amount higher than what they initially deposited since `withdrawFor` will not revert. As a consequence, the higher amount will be removed from the Uniswap V2 router, and the token balances will be deposited to the user-provided `to` address.

We classify this issue as major, since, despite its high impact, it has a precondition of the `KrStakingHelper` contract containing exceeding LP tokens.

Recommendation

We recommend updating `withdrawFor` to return the total amount withdrawn, and use this value in subsequent `approve` and `removeLiquidity` function calls.

Status

Acknowledged. Team's response: *"StakingHelper does not hold any funds"*.

17. `LibAssetUtility` price aggregation averages out oracles price data in case they deviate

---

In `src/contracts/minter/libs/LibAssetUtility.sol:202`, oracle prices are averaged out if they deviate by more than the `_oracleDeviationPct` parameter.

This calculation assumes that a deviation between oracles means the true market price is the average of both, which may not be the case. It is possible that only one of the oracles is faulty, meaning that the other oracle should be used instead.

Although it is possible to set Chainlink as the default oracle by using `_oracleDeviationPct` as 100%, it is not possible to set Redstone as the primary oracle. If Chainlink presents an abnormality, `_aggregatePrice` will return the average of a correct price and the wrong price data.

Recommendation
We recommend storing each oracle's last good price, along with its timestamp, and using this information to switch between a primary and secondary oracle, similar to Liquity's PriceFeed.sol smart contract. This allows for an automatic switch between oracles based on their deviation between rounds.

Status
Resolved

18. Users can prevent liquidations by frontrunning the transaction and slightly increasing their collateral

---

In the liquidation transaction, the caller has to specify the amount of debt they want to repay, `_repayAmount`. In `src/contracts/minter/facets/LiquidationFacet.sol:76`, the maximum value for that parameter is the maximum liquidatable value (MLV) in USD. Users can prevent liquidations by frontrunning the transaction and slightly increasing their collateral, which will, in turn, update the MLV.

We classify this issue as major since it can only temporarily prevent liquidation.

Recommendation

We recommend not reverting if `repayAmountUSD` is greater than the `maxLiquidableUSD`. Instead, it should just continue the execution with the MLV.

Status

Resolved

19. Users are unable to specify the maximum price they are willing to pay

---

In `src/contracts/minter/facets/MintFacet.sol:32`, users who want to mint Kresko assets via the `mintKreskoAsset` function are unable to specify the maximum price per asset, which can result in an unexpected expenditure when opening a position. This may pose a problem if the Kresko Asset price experiences a sudden increase, as users will be subject to a higher debt value and will pay a higher open fee.

In `src/contracts/minter/facets/LiquidationFacet.sol:36`, the function `liquidate` from `LiquidationFacet` does not allow liquidators to specify a maximum price they are willing to pay for the collateral they are liquidating. Although the liquidation incentive can be set to an amount exceeding 100%, potentially reducing the impact of a loss for the liquidator in this scenario, in the event of a drastic price change, liquidators might still end up paying more than they intended.

Recommendation

We recommend allowing users to specify a maximum acceptable price they are willing to pay.

Status

Acknowledged. Team's response: *"Feature that could be implemented in the future"*.

20. Liquidator will pay more debt than necessary if the account does not have enough collateral to back their deposits

If an account is liquidatable, a liquidator can seize part of the collateral of a borrower and repay its debt. If the debt exceeds the seized collateral, the liquidator will have repaid the debt in full but will only seize the collateral in part. This will happen both in `src/contracts/minter/facets/LiquidationFacet.sol:169` and in `src/contracts/minter/facets/InterestLiquidationFacet:159`.

For a test case, please refer to **Test case 5** in the **Appendix**.

Recommendation
We recommend deducting from the liquidator's balance the amount that was effectively seized from the borrower.

Status
Acknowledged. Team's response: *"Added an additional boolean parameter to explicitly allow this as it can be used to liquidate bad debt"*.

Minor Issues

21. Ineffective safety council controls on user actions

The safety council is able to temporarily suspend user actions, which include depositing and withdrawing collateral, as well as minting and burning Kresko assets, by using the `toggleAssetsPaused` function in the `SafetyCouncilFacet` contract.

However, functions executing user actions rely on the `safetyStateSet` boolean to decide if the asset's pause state should be considered. Due to the safetyStateSet boolean being set to false

by default and the absence of a dedicated setter, it is impossible to change its value to true. Consequently, the Safety Council cannot effectively pause user actions as intended.

Recommendation
We recommend implementing a setter for `safetyStateSet` in the `SafetyCouncilFacet` contract to enable the Safety Council to successfully manage user action states.

Status
Resolved

22. KISS stablecoin token is not pausable

The `KISS` stablecoin token has the ability to be paused by an address with the `PAUSER_ROLE` role. To prevent any kind of token transfer, the pausable state is checked in the `_beforeTokenTransfer` in the KISS contract. However, the `_beforeTokenTransfer` function is not executed during token transfers, minting, or burning. As a result, the pausing mechanism fails to function correctly, rendering the KISS token non-pausable.

Recommendation
We recommend updating the `transfer`, `transferFrom`, `_mint`, and `_burn` functions within the `ERC20Upgradeable` contract to call the `_beforeTokenTransfer` function. This will ensure that the required checks are performed before any token actions are executed, effectively enabling the pause functionality as designed.

Status
Resolved

23. DepositWithdrawFacet.withdrawCollateralUnchecked is vulnerable to reentrancy attacks

In `src/contracts/minter/facets/DepositWithdrawFacet.sol:73`, users can perform a flashloan-like operation by first withdrawing their collateral assets, and then having an MCR check applied after their callback has been executed.

The issue is that some state variables, such as the users' collateral asset balance and their deposited collateral assets list, will have been modified before the execution control is handed over to the user.

This can be used by an attacker to avoid paying close fees: since `recordCollateralWithdrawal` will remove the `_collateralAsset` from the `depositedCollateralAssets` minter state variable, during the callback execution, the attacker can then burn their Kresko Assets without paying fees from that collateral asset in `src/contracts/minter/libs/LibBurn.sol:130`. If the user has only one collateral and executes a `withdrawCollateralUnchecked`, then `chargeCloseFee` will not iterate through anything. Even though `burnKreskoAsset` contains a `nonReentrant` modifier, `withdrawCollateralUnchecked` does not, meaning that the reentrancy check will not stop the execution of this attack.

Another possible attack is withdrawing the collateral, depositing it to the AMM (which will update the TWAP after a `mint` from `src/contracts/vendor/uniswap/v2-core/UniswapV2Pair.sol:146`), updating the `stabilityRate` for an asset (calling `updateStabilityRateAndIndexForAsset` in `src/contracts/minter/facets/StabilityRateFacet.sol:100`), then repaying their interest with a reduced rate. The attacker can then undo all these operations to repay the collateral so that the MCR check passes after the callback execution.

We classify this issue as minor since this function can only be called by the manager in the current implementation.

Recommendation

We recommend adding a `nonReentrant` modifier to `withdrawCollateralUnchecked`, `updateStabilityRateAndIndexForAsset`, and other important operations that might be

vulnerable to a reentrancy attack caused by handing over the execution control to the user callback.

Status
Acknowledged. Team's response: *"Function is only called by permissioned contracts and does not need a re-entrancy guard."*.

24. Authorization.transferSecurityCouncil does not properly validate if msg.sender has SAFETY_COUNCIL role

---

In `src/contracts/libs/Authorization.sol:110`, the`transferSecurityCouncil` function does not properly validate if `msg.sender` has the `SAFETY_COUNCIL` role since it executes `hasRole` instead of a `checkRole`, which is essentially a no-op.

As a result, anyone can call this function and include any number of arbitrary `_newCouncil` addresses to the list of safety council members. By exploiting this vulnerability, an attacker can then upgrade important parameters of the system.

We classify this issue as major, since, despite its high impact, this function is not currently in use by the Diamond.

Recommendation
We recommend changing `hasRole` to `checkRole`. Additionally, we recommend ensuring static analyzers such as Slither are being run on the codebase so that similar mistakes can be easily flagged and prevented by the team (unused boolean return value).

Status
Resolved

25. LibRedstone returns the same index for all authorized signers

---

In `src/contracts/minter/libs/LibRedstone.sol:25`, `getAuthorisedSignerIndex` always returns `0` for all authorized signer addresses. This library was derived from `RedstoneConsumerBase`, but the original code states that:

```
  /**
   * @dev This function must be implemented by the child consumer contract.
   * It should return a unique index for a given signer address if the
signer
   * is authorised, otherwise it should revert
   * @param receivedSigner The address of a signer, recovered from ECDSA
signature
   * @return Unique index for a signer in the range [0..255]
   */
  function getAuthorisedSignerIndex(address receivedSigner) public view
virtual returns (uint8);
```

Recommendation

We recommend returning a unique index for the given signer.

Status

Resolved

26. LibAssetUtility lacks an L2 Sequencer Uptime Feed check

Optimistic rollup protocols move all execution off the layer 1 (L1) Ethereum chain, complete execution on a layer 2 (L2) chain, and return the results of the L2 execution back to the L1. These protocols have a sequencer that executes and rolls up the L2 transactions by batching multiple transactions into one transaction.

When utilizing Chainlink in L2 chains like Optimism, it is important to ensure that the prices provided are not falsely perceived as fresh, even when the sequencer is down.

Recommendation

We recommend following Chainlink's recommendation: Create the consumer contract for sequencer uptime feeds similarly to the contracts that you use for other Chainlink Data Feeds.

https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

Status
Resolved

27. Unbounded loops over arrays

Many instances of the codebase contain unbounded loops over collateral and Kresko asset arrays, such as in `src/contracts/minter/libs/LibAccount.sol:121`, `158`, `249`, in `src/contracts/minter/libs/LibBurn.sol:130`, `src/contracts/minter/libs/LibMint.sol:89`, and over pool IDs in `src/contracts/staking/KrStaking.sol:171`.

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 the protocols' operations and supporting collateral and Kresko assets so that users can not cause a Denial of Service due to unbounded loops. Additionally, consider refactoring how positions are being stored so that it is not necessary to perform unbounded iterations.

Status
Acknowledged. Team's response: *"Bookkeeping mechanism could be refactored in the future"*.

28. Unsafe type casting from signed to unsigned integer in LibDecimals

The `LibDecimals` library assists with the conversion between various decimal precisions. However, within the `oraclePriceToWad` function, an unsafe type casting from `int256` to `uint256` occurs, potentially resulting in large numbers close to `2^255`, leading to unexpected behavior and inflated prices and token amounts.

Negative prices can be provided to this function since both Oracles, Chainlink, and the custom Flux price feed contract, can possibly return negative values.

As the `oraclePriceToWad` function is only called from the `LibAssetUtility.wadPrice` functions, and those functions are not currently in use within the protocol, the vulnerability does not pose an immediate risk.

Recommendation
We recommend preventing negative prices from being returned by the oracle in the first place, or, alternatively, validating if the provided value in the `oraclePriceToWad` function is negative and reverting if it is.

Status
Resolved

29. Wrong MAX_MIN_DEBT_VALUE defined in Constants library

In `src/contracts/minter/MinterTypes.sol:44`, the `MAX_MIN_DEBT_VALUE` is set to `1000 gwei`, with comments indicating that it represents `$1,000` with 8 decimals. In reality, `1000 gwei` equals `1000000000000`, or `$10,000,` with 8 decimals.

Recommendation

We recommend changing the default value to `1_000 * 1e8`. Additionally, consider adding test cases for configuration and default values, and refactor the codebase to a different standard of decimals notation (e.g., `gwei` vs. `ether` vs. `1eX` exponential notation).

Status
Resolved

30. Redstone timestamp validation is disabled

The `validateTimestamp` function in the `LibRedstone` library is used to validate the received timestamp against the current time by calling the `validateTimestamp` function in the `RedstoneDefaultsLib` helper library. However, this call is presently deactivated, as it has been commented out, leading to a disabled timestamp validation.

Recommendation
We recommend reactivating the call by uncommenting `RedstoneDefaultsLib.validateTimestamp(receivedTimestampMilliseconds);` in line 115 of the `LibRedstone` contract, ensuring proper timestamp validation.

Status
Acknowledged. Team's response: *"We will enable the validation for mainnet deployment - timestamp validation disabled as testing purposes"*.

31. Use of Solidity's `transfer` function might render ETH impossible to withdraw

---

Solidity's `transfer` function has some notable shortcomings when a caller is a smart contract, which can render the `rescueNative` function in the `KrStaking` contract, used to rescue accidentally sent native tokens, unusable. Specifically, the transfer will inevitably fail when:

- The caller smart contract does not implement a payable fallback function.

- The caller smart contract implements a payable fallback function that uses more than 2300 gas units.
- The caller smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

The `sendValue` function available in OpenZeppelin Contract's `Address` library can be used to transfer the withdrawn Ether without being limited to 2300 gas units. Risks of reentrancy stemming from the use of this function can be mitigated by tightly following the "Check-effects-interactions" pattern and using OpenZeppelin Contract's `ReentrancyGuard` contract. For further reference on why using Solidity's transfer is no longer recommended, please refer to these articles:

- Stop using Solidity's transfer now
- Reentrancy after Istanbul

Recommendation
We recommend using Solidity's low-level `call` function or the `sendValue` function available in OpenZeppelin Contract's Address library to send native tokens.

Status
Resolved

32. Unnecessary `payable` specifier for functions may allow ETH to be sent and locked

---

Almost all publicly accessible functions in the `KrStaking` contract are marked as `payable` even though they do not expect to receive and process native tokens. This means that interacting users can accidentally send native tokens to these functions, which will get locked/lost and must be rescued by a permissioned operator.

Recommendation

We recommend removing the `payable` specifier on functions that do not expect to receive and process native tokens.

Status
Acknowledged. Team's response: *"Rescue native was introduced as part of audit fixes."*.

33. KrStaking operator can withdraw and steal user funds

The permissioned address with the `OPERATOR_ROLE` can withdraw any amount of staked tokens from users via the `withdrawFor` function. While this function is intended to be used by the KrStakingHelper helper contract, users who staked tokens without the helper contract are vulnerable to having their tokens stolen by the operator.

Recommendation
We recommend withdrawing the `depositToken` to the user's address.

Status
Acknowledged. Team's response: *"Operator would be multisig / verified contract, also if we send users the fund, krStakingHelper contract would not work"*.

34. Emergency withdrawal in KrStaking contract executes additional, unnecessary logic, potentially preventing withdrawals

The `emergencyWithdraw` function in the `KrStaking` contract is intended to allow users to withdraw their staked tokens in case of an emergency. However, the function executes additional, unnecessary logic by calling the `updatePool` function on line 288. Since the accrued rewards will be forfeited in an emergency withdrawal, invoking the `updatePool` function is not required and could potentially prevent users from withdrawing their tokens if there are any issues with the `updatePool` function.

Recommendation

We recommend removing the `updatePool` function call within the `emergencyWithdraw` function to keep the function logic to a minimum.

Status
Resolved

35. `EIP712_DOMAIN_TYPEHASH` uses `salt` instead of `chainId`

The EIP-712 domain type hash in the `Meta` library is defined as the `keccak256` hash of `"EIP712Domain(string name,string version,uint256 salt,address verifyingContract)"`. However, calculating the domain separator hash in the `domainSeparator` function uses the current chain ID instead of the `salt` value. This means that the domain separator hash will differ from the expected value, possibly leading to incorrect signatures.

Recommendation
We recommend renaming `uint256 salt` to `uint256 chainId` in the `EIP712_DOMAIN_TYPEHASH` definition.

Status
Resolved

36. `UniswapV2Oracle` uses potentially outdated incentive amount

The `UniswapV2Oracle` incentivizes users to update the cumulative price for a given asset with the updateWithIncentive function. At the end of the function execution, the caller receives `3 ether` in `incentiveToken`. However, this value is supposed to be determined by the `incentiveAmount` storage variable. As a result, when the `incentiveAmount` is updated, the `updateWithIncentive` function will transfer the outdated amount of `3 ether` to the caller.

Recommendation

We recommend using the `incentiveAmount` instead of hardcoding the token amount to `3 ether`.

Status
Acknowledged. Team's response: "*Incentive is experimental and the uniswap oracle can be redeployed if changed.*".

37. Unexpected behavior if price Oracle with decimals other than 8 are used

---

The Kresko protocol is currently designed to work with Chainlink and Redstone price feeds that use 8 decimals. However, potential issues may arise from using oracles with decimals other than 8, as many calculations assume that the price returned has 8 decimals.

For example, in the `LibStabilityRate` library, the `getPriceRate` function calculates the price rate between the Automated Market Maker (AMM, i.e., Kresko's Uniswap v2 fork) and the oracle using the `wadDiv` operation. This function performs a division of the AMM price (in 18 decimals) by the oracle price (in 8 decimals), followed by a division by `10` in order to reduce the resulting precision from 28 to 27 decimal places, yielding the desired `RAY` precision. Employing an oracle with a different number of decimals leads to imprecise calculations and inaccurate outcomes.

Moreover, the `divByPrice` function in the `LibDecimals` library is expected to return the price in 18 decimals. However, if the oracle decimals are equal or greater than 18 decimals, the oracle price will be returned as-is without multiplying by the given `_value`. This results in an incorrect value being returned and the precision possibly larger than 18 decimals. This will cause the interest repayment calculation in `StabilityRateFacet.repayStabilityRateInterestPartial` in line 141 to be incorrect. Such errors may cause over- or under-accounting of the repaid interest.

Recommendation

We recommend refraining from using Oracles with decimals other than 8, or, alternatively, updating the code to support Oracles with a different number of decimals, also considering the possibility of using Oracles with more than 18 decimals.

Status
Acknowledged. Team's response: "*We will be using 8 decimals oracles only.*".

38. Centralization risk

The Kresko Synthetic Asset 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 timelock and multisig wallets to delay sensitive operations and avoid a single point of key management failure. Long term, we recommend renouncing ownership or upgradability of contracts that do not need such features.

Status
Acknowledged. Team's response: "*Initially we will use a multisig and then move to a more detailed governance process with timelock*".

39. LibDiamondCut does not validate if the facet address parameter is the Diamond address

---

In `src/contracts/diamond/libs/LibDiamondCut.sol:132`, no checks are performed requiring `_facetAddress` to be different than `address(this)`. In contrast, the LibDiamond.sol reference implementation contains this check preventing the replacement and removal of immutable functions.

Recommendation
We recommend including the additional check from the reference implementation. Long term, we recommend ensuring that libraries and functions from the ERC-2535 standard are reused thoroughly from the reference implementations.

Status
Acknowledged. Team's response: "*We will continue with the reference implementation for now*".

40. Optimism does not support `PUSH0` yet

The Kresko smart contracts are compiled with Solidity version `>=0.8.20`, which supports EIP-3855, and the new `PUSH0` instruction, which pushes a zero value onto the stack. This allows for more efficient and optimized bytecode, improving the performance of smart contracts.

The issue is that `PUSH0` is still not supported on many Layer 2. As a result, contracts compiled with Solidity version `0.8.20` may not be able to be deployed to Optimism mainnet.

Recommendation
We recommend using Solidity version `0.8.17` until Optimism fully supports `PUSH0`.

Status
Acknowledged. Team's response: "*We will adjust deployment solc version*".

41. `KISS` code is not production ready

Many instances of `src/contracts/kiss/KISS.sol` contain code used for testing purposes.

In line `52`, the input validation on the `admin_` address is commented out. The `require` check was meant to prevent EOAs from being assigned to the admin address.
In lines `76-79`, the `initializer` function contains blocks stating `Deployer does not need roles, uncomment for mainnet`.

Recommendation
We recommend reverting the changes used for testing and, additionally, consider reviewing the testing processes so that production code does not need to be altered in the local development environment.

Status
Resolved

42. `KISS.supportsInterface` does not return `true` for inherited contracts' interfaces

---

In `src/contracts/kiss/KISS.sol:83-92`, the function `supportsInterface` from the KISS smart contract does not return `true` for inherited contracts' interfaces. For example, some functions from `AccessControlEnumerableUpgradeable` have not been exhaustively considered in the function implementation, and as such, they will not be considered.

Recommendation
We recommend using `super.supportsInterface` whenever extending from a contract, so that the inherited contracts' interfaces are also validated.

Status
Resolved

43. `KISS.grantRole` overrides the `_to` function parameter in some cases, which can be confusing to users

---

In `src/contracts/kiss/KISS.sol:138`, the function `grantRule` from the `KISS` smart contract overrides the `_to` parameter if the `_role` being attributed is the `OPERATOR` role and if there is a `pendingOperator` due to the two-step transfer process. This inconsistency can be confusing to users, as in some cases, the `_to` value is directly used (either if the role is not `Role.OPERATOR` or if the operator is being initialized for the first time).

Recommendation

We recommend not overriding the `grantRole` function and instead adding a new function `grantRoleOperator,` that implements the two-step transfer process only for the OPERATOR role.

Status

Acknowledged. Team's response: "*Caller of function should have this knowledge*".

44. `KISS.setMaxOperators` does not validate if the current validator count is greater than the updated maximum

---

In `src/contracts/kiss/KISS.sol:132`, the `setMaxOperators` function from the `KISS` smart contract does not apply any input validation on the `_maxOperators` parameter. In particular, it does not check if the current validator count is greater than the new value. When this happens, it will have no effect on the existing number of validators, which can be confusing to the person executing this function.

Recommendation

We recommend validating that `_maxOperators` is lower than the current number of operators and requesting the `ADMIN` to remove some operators if updating to a lower value is intended.

Status

Acknowledged

45. `bytes32` role values are not consistent across contracts

In `src/contracts/libs/Authorization.sol:23-34`, `src/contracts/staking/KrStaking.sol:15`, and `src/contracts/vendor/flux/FluxPriceFeed.sol:12-13`, the `bytes32` constant values for access control roles are not consistent across contracts. For example, some contracts define an `ADMIN_ROLE` equal to `keccak256("ADMIN_ROLE")`, while others use `ADMIN` equal to `keccak256("kresko.roles.minter.admin")`. This inconsistency can cause operational issues if a developer needs to quickly update roles on different contracts.

Recommendation

We recommend using the same standard of `bytes32` role values across all contracts.

Status

Acknowledged. Team's response: "*This concerns Flux Price Feed. We will use Redstone values for market status in production. Their custom implementation on the market status data is pending*".

46. `Authorization.setupSecurityCouncil` notice suggests performing `ERC165` validation, but this is not done

---

In `src/contracts/libs/Authorization.sol:96`, a comment on the function `setupSecurityCouncil` reads `Checks if the target contract implements the ERC165 interfaceId for the multisig`, suggesting that it would perform a check to verify that the `_councilAddress` address would be a multisig. However, no checks are performed by the function. Instead, it only verifies if `IGnosisSafeL2(_councilAddress).isOwner(msg.sender)` returns true. It is possible that `_councilAddress` implements a `isOwner` function but does not implement other important multisig features required by the `Authorization` library.

Recommendation

We recommend adding a check to verify that `_councilAddress` implements all required methods from the multisig.

Status

Acknowledged. Team's response: "*Removing the natspec referring to ERC165 validation*".

47. Loss of precision due to division before multiplication

1. In `src/contracts/minter/libs/LibDecimals.sol`, lines `81` and `91`, the functions `divByPrice` and `fromWadPriceToUint` from `LibDecimals` perform division before multiplication when converting from and to WAD. As a result, they might incur an unnecessary loss of precision.
2. In `src/contracts/minter/libs/LibCalculation.sol:159-163`, the function `_getMaxLiquidatableUSD` from `LibCalculation` performs division before multiplication when converting from and to WAD. As a result, they might incur an unnecessary loss of precision.

Recommendation

We recommend performing multiplication before division whenever there is no risk of overflow.

Status

Resolved

48. LibUI.krAssetInfoFor ignores krFactor in the debt calculation

In `src/contracts/minter/libs/LibUI.sol:353`, the function `krAssetInfoFor` from `LibUI` ignores the `krFactor` when calculating the debt amount in USD. This is a mistake, as the protocol always includes the `krFactor` when calculating the debt of a position, as seen in `src/contracts/minter/libs/LibAccount.sol:161`. As a result, the value shown in the UI or used by third-party contracts integrating `LibUI` will show a different value from what is registered by the protocol.

Recommendation

We recommend changing `true` to `false` to ensure the `krFactor` is not ignored when calculating the debt amount in USD. Additionally, consider properly testing the `LibUI` contract and other secondary or helper libraries, even if they are not on the critical path of the protocol.

Status

Acknowledged. Team's response: "*Non-Issue - UI specific contract*".

49. Some LibUI functions always return zero due to a typo in increment/assignment operator

---

In `src/contracts/minter/libs/LibUI.sol:322` and `362`, the functions `collateralAssetInfoFor` and `krAssetInfoFor` from `LibUI` always return `0` as the result of `totalCollateralUSD` and `totalDebtUSD` due to a typo in the increment/assignment operator. Because no assignment is made, the increment is a no-op.

Recommendation

We recommend replacing the aforementioned lines with `totalDebtUSD += amountUSD;` and `totalCollateralUSD += amountUSD;`, respectively.

Status

Acknowledged. Team's response: "*Non-Issue - UI specific contract*".

50. Missing input validations

---

1. In `src/contracts/minter/facets/ConfigurationFacet.sol:117-120`, the function `updateMinimumCollateralizationRatio` from `ConfigurationFacet` updates the contract's minimum collateralization ratio (MCR). The issue is that it lacks an important check regarding the liquidation threshold (LT). Although the minimum default values guarantee that MCR >= LT, it is possible that both of these values are updated throughout the normal operations of the protocol. A later reduction of the MCR to a value lower than the LT could make positions instantly liquidatable. For example, if the MCR was updated to 150%, the LT to 140%, and then the MCR to 130%.

2. In `src/contracts/minter/facets/ConfigurationFacet.sol:153`, the function `updateAMMOracle` from `ConfigurationFacet` does not validate that `_ammOracle` is not equal to the `address(0)`. Setting this value to zero as a mistake would break features related to the Stability Rate mechanism of the protocol.
3. In `src/contracts/minter/facets/StabilityRateFacet.sol:88`, the function `updateStabilityRateParams` from `StabilityRateFacet` does not validate that `_setup.stabilityRateBase` is greater than or equal to `1 RAY`. If a wrong value is set by mistake, it could lead to negative interest rates in `src/contracts/minter/libs/LibStabilityRate.sol:87` and break features related to the Stability Rate mechanism of the protocol.
4. In `src/contracts/minter/amm-oracle/UniswapV2Oracle.sol:84`, the function `setAdmin` from `UniswapV2Oracle` does not validate that `_newAdmin` is not the `address(0)`. Setting this value as zero as a mistake would make it impossible for the admin to configure pairs, withdraw fees, and perform other important management operations on this contract.
5. In `src/contracts/staking/KrStaking.sol:313`, the function `addPool` from `KrStaking` does not validate that `_startBlock` should not be in the future, as this will cause reverts

Recommendation

1. We recommend adding a safety check in `updateMinimumCollateralizationRatio,` requiring the minimum collateralization ratio to be greater than or equal to the liquidation threshold. In case the admin wishes to lower these two values, it will be possible by first updating the LT and then the MCR.
2. We recommend adding a check that `_ammOracle` differs from `address(0)` in `updateAMMOracle`.
3. We recommend validating that `_setup.stabilityRateBase` is at least `1 RAY` in `updateStabilityRateParams`.
4. We recommend validating that `_newAdmin` is different from `address(0)` in `setAdmin`.
5. We recommend validating that `_startBlock` is not in the future.

Status

Resolved

51. FluxPriceFeed getters' behavior is inconsistent

In `src/contracts/staking/FluxPriceFeed.sol`, the getter functions from `FluxPriceFeed` are not consistent if an invalid `_roundId` is passed as a parameter. In particular, `getAnswer` and `getTimestamp` return 0, while `getMarketOpen` reverts.

Recommendation
We recommend always returning the default value or always reverting in case of an invalid `_roundId` parameter.

Status
Acknowledged. Team's response: "*We will use Redstone values for market status in production. Their custom implementation on the market status data is pending*".

52. LibCalculation._getMaxLiquidatableUSD can revert with an underflow depending on specific configuration values

In `src/contracts/minter/libs/LibCalculation.sol:155-157`, the `valuePerUSDRepaid` local variable is calculated as the debt factor (which depends on the krFactor, the liquidation threshold, and the collateral factor) minus the liquidation incentive close fee.

Depending on these values, this calculation can revert with an underflow. In particular, a high enough liquidation incentive can make the subtraction negative and underflow, which can, in theory, be caused by a configuration mistake or a lack of validation by the admin.

Recommendation
We recommend validating and monitoring that the liquidation incentive is not so high that it will make the above calculation underflow.

Status
Acknowledged. Team's response: "*Recommendation taken*".

53. Diamond.sol: Inability to retrieve sent native tokens

The contract has a `receive` function, but there is no way to retrieve any tokens sent. This will result in any tokens sent to the contract being locked and unretrievable.

Recommendation
We recommend adding a function to retrieve sent tokens to prevent them from being locked in the contract.

Status
Resolved

54. Authorization.sol: No check to ensure multisig has sufficient owners during setup

The `setupSecurityCouncil` function does not check to ensure that the multisig contract has sufficient owners (>=5) when establishing the `SAFETY_COUNCIL`. This could result in an insufficient number of signers during setup. This check is being done within the `transferSecurityCouncil` function.

Recommendation
We recommend adding a check to ensure that the multisig contract has a sufficient number of owners during setup.

Status
Acknowledged. Team's response: "*Non-Issue*".

Informational Notes

55. LibUI.sol: Possible array index out of bounds

In the `batchOraclevalues` function, the for loop iterates over `_assets` and accesses indexes in both `_priceFeeds` and `_marketStatusFeeds`. However, if `_assets` are not the same size as `_priceFeeds` and `_marketStatusFeeds,` this could lead to an index out-of-bounds error.

Recommendation

We recommend adding a check to ensure that `_priceWithOrcaleDecimals >= 0.`

Status

Acknowledged. Team's response: "*Non-Issue: UI specific contract*".

56. KrStaking.sol: Unnecessary swapping of array values

In the function on line 145 the following statement is being executed `rewards.tokens[rewardIndex] = pool.rewardTokens[rewardIndex]`. However, both `rewards.tokens` and `pool.rewardTokens` reference the same array.

Recommendation

We recommend removing the aforementioned line with the unnecessary array swap.

Status

Resolved

57. ConfigurationFacet.sol: Possible gas savings

There are multiple writes to storage when updating the `CollateralAsset` struct in the `updateCollateralAsset` function.

Recommendation

We recommend updating the `CollateralAsset` struct in memory and then writing to storage once. This is what is being done when updating the `KrAsset` struct in the `updateKreskoAsset` function.

Status
Resolved

58. Potentially misleading `SafetyStateChange` event description

When multiple assets have their paused status updated with the `toggleAssetsPaused` function in the `SafetyCouncilFacet` contract, the `description` value of the `SafetyStateChange` event might be set to paused if a single asset's paused status is toggled to `true`, even if this asset's paused status has been unpaused (i.e., toggled to `false`). Off-chain monitoring systems might then falsely report that an asset has been paused when in fact, it has been unpaused.

Recommendation
We recommend emitting the correct description value for each asset's paused status and, if a separate event is needed to indicate that an asset has been paused, emitting a separate event for this purpose.

Status
Resolved

59. Asset pause duration is unused

Assets can be temporarily paused on a per-action with the `toggleAssetsPaused` function in the `SafetyCouncilFacet` contract. This function allows providing a specific duration for the pause. However, the `ensureNotPaused` function in the `MinterModifiers` contract does not check the duration of the pause. This could lead to unexpected behavior if the pause is assumed to be time-limited, causing assets to be paused indefinitely until the safety council unpauses them.

While there is a comment in `ISafetyCouncilFacet.sol` in line 18 indicating that this feature is not implemented yet, the `toggleAssetsPaused` function allows the duration to be set.

Recommendation
We recommend checking both `timestamp0` and `timestamp1` in the `ensureNotPaused` function in `MinterModifiers` to determine if the pause is currently active.

Status
Acknowledged. Team's response: "*Non-Issue*".

60. 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. Team's response: "Non-Issue".

61. Misleading NewOperator event operator value

In the `KISS` contract, when the `Role.OPERATOR` role is granted to the current `pendingOperator` through the `grantRole` function, a `NewOperator` event is emitted. However, the `operator` value within the event is set to `msg.sender`, which represents the current admin rather than the

`pendingOperator`. This leads to off-chain monitoring systems generating false reports regarding the new operator.

Recommendation
We recommend using the `pendingOperator` as the `operator` event value when emitting the `NewOperator` event.

Status
Resolved

62. `ReentrancyGuardUpgradeable` contract is not properly initialized in the KrStaking contract

---

The `KrStaking` contract inherits from the `ReentrancyGuardUpgradeable` contract. The `ReentrancyGuardUpgradeable` contract uses a `uint256 _status` mutex to protect against reentrancy attacks by using the `nonReentrant` modifier. This variable is initialized in the `__ReentrancyGuard_init` function and is set to the value of `_NOT_ENTERED = 1`. When the `nonReentrant` modifier is used on a function, it prevents reentering the function if `_status` is not equal to `_NOT_ENTERED`.

However, the `KrStaking` contract does not call the `__ReentrancyGuard_init` function in its `initialize` function, which means that the `_status` variable is not properly initialized and is left at the default value of `0`.

While this does not have any security implications, nor does it affect the functionality of the reentrancy guard, it is not considered a best practice and should be fixed.

Recommendation
We recommend calling `__ReentrancyGuard_init` in the `initialize` function of the `KrStaking` contract to properly initialize the `ReentrancyGuardUpgradeable` contract.

Status

Resolved

63. Code simplification

1. In `src/contracts/minter/facets/AccountStateFacet.sol:138`, the function `calcExpectedFee` from `AccountStateFacet` validates that the `_feeType` is lower than or equal to 1. This validation could be avoided if the parameter type of `_feeType` was changed to an `enum Fee`.
2. In `src/contracts/vendor/flux/FluxPriceFeed.sol:69`, the function `latestTransmissionDetails` from `FluxPriceFeed` performs an unnecessary restriction that `msg.sender` is equal to `tx.origin` to require that this function is only callable by an EOA. Not only can this check be bypassed through a contract's constructor, but also these same variables are available through other external methods.

Recommendation

1. We recommend changing the function parameter type from `uint256` to `Fee` in order to remove the unnecessary validation from `calcExpectedFee`.
2. We recommend removing the unnecessary EOA check from `latestTransmissionDetails`.

Status

Acknowledged

64. Code repetition

1. In `src/contracts/minter/amm-oracle/UniswapV2Oracle.sol:188`, the function `updateWithIncentive` from `UniswapV2Oracle` contains code repetition from the function `update`. The whole update logic is copied/pasted to the new function, with the sole difference being that `updateWithIncentive` sends `incentiveToken` to the `msg.sender` at the end of the execution, which hinders both readability and maintainability.

2. In `src/contracts/staking/KrStaking.sol:244` and `350`, then `271` and `382`, the functions `withdraw`/`withdrawFor`, `claim`/`claimFor` contain code repetition since the only difference is the `msg.sender` being a function parameter or a hardcoded value on the function body.

Recommendation
We recommend removing the repetitive code by creating internal functions that abstract away the common logic and using this function when necessary.

Status
Resolved

65. Missing events in important functions

---

1. In `src/contracts/vendor/flux/FluxPriceFeedFactory.sol:39`, no event is emitted on the `transferOwnership` function.
2. In `src/contracts/staking/KrStaking.sol:271` and `335`, no events are emitted on `claim` and `setPool`. As a comparison, MasterChefV2 emits `Harvest` and `LogSetPool` for similar function execution calls.

Recommendation
We recommend emitting events on important function calls.

Status
Resolved

66. Usage of tx.origin

In `src/contracts/minter/facets/LiquidationFacet.sol:108`, the `liquidate` function emits a `MinterEvent.LiquidationOccurred` event. From the NatSpec in `src/contracts/libs/Events.sol:232`, the parameter `liquidator` is "The account performing the liquidation". However, `tx.origin` is being used here instead of `msg.sender`.

Because of how address aliasing works on Optimism, the value of `msg.sender` at the top level (the very first contract being called) is always equal to `tx.origin`, so these should be equal. However, for the sake of consistency, it could be more appropriate to use `msg.sender` in this case.

Recommendation

We recommend using `msg.sender` instead of `tx.origin`.

Status

Resolved

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

---

In `src/contracts/minter/facets/AccountStateFacet.sol:108` and `src/contracts/minter/libs/LibUI.sol:329`, the protocol returns 0 for the collateralization ratio if the user has no debt. Since the collateralization ratio is derived from the account collateral divided by the account debt, this can be misleading since the higher the collateralization ratio, the healthier the position is, and a value of zero is defined for an account with no collateral to back its 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` as the collateralization ratio of an account with no debt.

Status

Acknowledged. Team's response: "*Non-Issue*".

68. Typos

---

1. In `src/contracts/diamond/libs/LibDiamondCut.sol:12`, it should be `but _calldata`
2. In `src/contracts/kreskoasset/IKreskoAsset.sol:12`, it should be `denominator`
3. In `src/contracts/minter/libs/LibCalculation.sol:79`, it should be `collateral asset.`
4. In `src/contracts/minter/interfaces/IBurnFacet.sol:10`, it should be `kresko asset`
5. In `src/contracts/minter/interfaces/IConfigurationFacet.sol:34`, it `should be Updates a previously added kresko asset`
6. In `src/test/oracle/00-feeds-and-redstone.ts:231` and `234`, it should be `$15`
7. In `src/contracts/libs/Arrays.sol:18`, it should be `Error.ARRAY_INDEX_MISMATCH` or similar error code
8. In `src/contracts/minter/interfaces/IBurnHelperFacet.sol:9` and `16`, it should be `enough`
9. In `src/contracts/minter/libs/LibMint.sol:33` it should be `minted`

Status
Resolved

69. Unclear documentation

In `src/contracts/kiss/KISS.sol:138`, the function `grantRole` from `KISS` overrides `AccessControl.grantRole` by implementing a cooldown period of `pendingOperatorWaitPeriod` minutes for setting a new `OPERATOR_ROLE`. However, the first time the operator is set, this unlock time is bypassed in line `168`. The documentation does not specify or make it clear why this behavior is intended.

Recommendation
We recommend refactoring the code to guarantee that the unlock time is applied to all operators or document why the first-time operator setup bypasses the unlock time.

Status
Resolved

70. Naming conventions

1. Throughout the protocol, some boolean variables are named on the negative form rather than on the affirmative form. For example, the variables `_ignoreCollateralFactor` and `_ignoreKFactor` are used when calculating a collateral or Kresko asset value. This can be confusing to readers, as it introduces an additional cognitive load when making sense of a function call since the false value means the factors should be applied.
2. In `src/contracts/kreskoasset/KreskoAsset.sol` and `src/contracts/kreskoasset/Rebase.sol`, the rebase nomenclature for the denominator parameter can confusing if the rebase is either `positive` or `negative`. When the rebase is `positive`, it acts as a de facto denominator, dividing the amount value by this parameter. However, when the rebase is `negative`, it is used instead as a multiplier, as no division is performed.

Recommendation

1. We recommend naming boolean values in the affirmative form rather than in the negative form. In addition, it should also be considered if, due to the simplicity of this operation, these parameters can be removed altogether, and the caller decides if they want to apply the krFactor/cFactor or not. Alternatively, it can be considered if creating additional functions with explicit naming (e.g., `*WithCFactor`) can improve readability and maintainability.
2. We recommend naming the rebase `denominator` parameter as `factor` or other generic term that can be applied both to a division or multiplication. Alternatively, always perform a multiplication, and use the inverse in case the rebase is positive. For example, instead of a positive rebase of 2 dividing the amount by two, a `factor` of `0.5` could be multiplied by the amount to simplify the calculation.

Status

Acknowledged

71. DiamondOwnershipFacet.sol: Redundant check for owner and pending owner

The `onlyOwner` modifier on `transferOwnership` and the `onlyPendingOwner` modifier on `acceptOwnership` are redundant. This is because `ds.initateOwnershipTransfer` already checks that `self.contractOwner` is the caller and `ds.finalizeOwnershipTranser` also checks that `self.pendingOwner` is the caller.

Recommendation

We recommend moving the `onlyOwner` modifier to `initiateOwnershipTransfer` and the `onlyPendingOwner` modifier to `finalizeOwnershipeTransfer` instead.

Status

Resolved

72. LibDiamondCut.sol: Unnecessary use of require statement

The `_facetAddress` parameter in the `removeFunctions` function and its associated `require` statement are not needed to carry out this function call.

Recommendation

We recommend removing the parameter and the associated `require` statement.

Status

Acknowledged. Team's response: "*Part of the standard, we do not want to modify it*".

73. KISS.sol: Unused constant role variable

The `OPERATOR_ROLE` variable is not used at all within the contract. Also, this role is already defined in the `Role` library in `Authorizations.sol`.

Recommendation

We recommend removing the unused variable.

Status

Resolved

74. KISS.sol: Redundant initialization of variable

The variable initialization on line 71 is already being done on line 58.

Recommendation

We recommend removing the redundant variable initialization.

Status

Resolved

75. ERC4626Upgradeable.sol: Incorrect comment

On line 81, the comment for the `issue` function reads, “When new KreskoAssets are burned:”. However, this should read “When new KreskoAssets are minted:”. A similar issue also exists for the comments of the `mint` function in `LibMint.sol`.

Recommendation

We recommend commenting to reflect the correct action of the function.

Status

Resolved

76. KreskoAsset.sol: Redundant check for isRebase

On lines 77, 83, 159, 173, and 192, the condition if statement is redundant as the `rebase` and `unrebase` functions in `Rebase.sol` already checks for `isRebase` with `denominator == 0`.

Recommendation

We recommend leaving the check for `isRebase` to the `rebase` and `unrebase` functions.

Status

Resolved

77. Arrays.sol: Misleading revert statement

The `removeAddress` function reverts with `Error.ARRAY_OUT_OF_BOUNDS` if `_addresses[_elementIndex] != _elementToRemove`. If the index is within bounds, but the values do not match, this will revert with a misleading error message.

Recommendation

We recommend changing the error message to `Error.INCORRECT_INDEX`.

Status

Resolved

78. Authorization.sol: Redundant removal of the account from role members

---

The statement on line 148 is redundant and is already being done in the `_revokeRoke` function.

Recommendation

We recommend removing the redundant statement.

Status

Resolved

79. AccountStateFacet.sol: Redundant declaration of library use

The statement on line `19` is redundant and is already being done on line `17`. This issue also exists in `LibBurn.sol` on line `28` which is already specified in `MinterState.sol`.

Recommendation

We recommend removing the redundant statement.

Status

Resolved

80. BurnFacet.sol: Possible zero burn

The `burnKreskoAsset` function on line `39` does not check to ensure `debtAmount != 0`. If `_burnAmount == type(uint256).max,` this could result in an unnecessary burn on line 59. A similar check is being done on line 36 in `BurnFacetHelper.sol`.

Recommendation

We recommend adding a check to ensure `debtAmount != 0`.

Status

Resolved

81. StabilityRateFacet.sol: Partial repayment of stability rate interest not enforced

The `require(_kissRepayAmount < maxKissRepayAmount,...)` statement in line 125 is not partial. For example, if `maxKissRepayAmount == 50,` this will allow the user to repay up to `49,` which is almost the entire amount.

Recommendation

We recommend adding an upper bound well to the max figure to enforce partiality.

Status

Acknowledged

82. ERC-165 interfaceIds are not self-evident

In `src/contracts/kiss/KISS.sol:91`, `src/contracts/kreskoasset/KreskoAsset.sol:66` and `src/contracts/kreskoasset/KreskoAssetAnchor.sol:68`, some ERC-165 `interfaceIds` 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

83. Kresko protocol does not support fee-on-transfer collateral tokens

All functions of the protocol do not expect fee-on-transfer tokens. It can be problematic if additional collateral assets are integrated and are able to charge fees in the future.

Recommendation

We recommend documenting that fee-on-transfer tokens are not supported.

Status

Resolved

84. Kresko protocol does not support native cryptocurrencies as collateral assets

The Gitbook documentation lists "Native cryptocurrencies: existing digital assets available within the relevant blockchain ecosystem such as ETH." as possible Collateral Asset Types. However, all protocol functions are only programmed to work with ERC20 tokens.

Recommendation

We recommend updating the documentation to match the implementation protocol by specifying that native cryptocurrencies must be used in their wrapped format (such as WETH).

Status

Resolved

Appendix

Test case 1

```
diff --git a/src/test/minter/03-liquidation.ts
b/src/test/minter/03-liquidation.ts
index b50cb31..26d3ab5 100644
--- a/src/test/minter/03-liquidation.ts
+++ b/src/test/minter/03-liquidation.ts
@@ -11,7 +11,7 @@ import {
 import { expect } from "@test/chai";
 import { fromBig, getInternalEvent, toBig } from "@kreskolabs/lib";
 import { Error } from "@utils/test/errors";
-import { addMockCollateralAsset, depositCollateral, getCollateralConfig }
from "@utils/test/helpers/collaterals";
+import { addMockCollateralAsset, depositCollateral, depositMockCollateral,
getCollateralConfig, withdrawCollateral } from
"@utils/test/helpers/collaterals";
 import { mintKrAsset } from "@utils/test/helpers/krassets";
 import { getExpectedMaxLiq, getCR, liquidate } from
"@utils/test/helpers/liquidations";
 import { LiquidationOccurredEvent } from
"types/typechain/src/contracts/libs/Events.sol/MinterEvent";
@@ -422,6 +422,93 @@ describe("Minter", () => {

expect(afterKreskoCollateralBalance).lt(beforeKreskoCollateralBalance);
             });

+            it.only("should allow unlimited kresko asset minting due to
faulty account liquidation", async function () {
+                const userThree = hre.users.userThree;
+                const deposits = toBig(15);
+                const borrows = toBig(10);
+
+                this.collateral.setPrice(10);
+                this.krAsset.setPrice(10);
+
+                await this.collateral.setBalance(userThree, deposits);
+                await depositCollateral({
```

```
+                    user: userThree,
+                    amount: deposits,
+                    asset: this.collateral,
+                });
+
+                await mintKrAsset({
+                    user: userThree,
+                    amount: borrows,
+                    asset: this.krAsset,
+                });
+
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.false;
+
+                this.collateral.setPrice(7); // @audit-info cause a 30%
collateral price drop
+
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.true;
+
+                const liquidationAmount = toBig(10);
+
+                /* Setup liquidator */
+
+                await
this.collateral!.mocks.contract.setVariable("_balances", {
+                    [hre.users.liquidator.address]: toBig(100),
+                });
+                await depositMockCollateral({
+                    user: hre.users.liquidator,
+                    asset: this.collateral,
+                    amount: toBig(100_000),
+                });
+
+                await mintKrAsset({
+                    user: hre.users.liquidator,
+                    asset: this.krAsset,
+                    amount: liquidationAmount,
+                });
```

```
+
+                await wrapContractWithSigner(hre.Diamond,
hre.users.liquidator).liquidate(
+                    userThree.address,
+                    this.krAsset.address,
+                    liquidationAmount,
+                    this.collateral.address,
+                    await
hre.Diamond.getMintedKreskoAssetsIndex(userThree.address,
this.krAsset.address),
+                    await
hre.Diamond.getDepositedCollateralAssetIndex(userThree.address,
this.collateral.address),
+                );
+
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.false;
+
+                await this.collateral.setBalance(userThree, deposits);
+
+                this.collateral.setPrice(10);
+
+                await depositCollateral({
+                    user: userThree,
+                    amount: deposits,
+                    asset: this.collateral,
+                });
+
+                await mintKrAsset({
+                    user: userThree,
+                    amount: borrows,
+                    asset: this.krAsset,
+                });
+
+                const mintedKreskoAssetsBefore = await
hre.Diamond.getMintedKreskoAssets(userThree.address);
+                expect(mintedKreskoAssetsBefore).to.deep.equal([]); //
@audit-issue Account has no minted kresko assets
+
```

```
+                this.collateral.setPrice(1);
+
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.false; //
@audit-issue Account is not considered liquidatable
+
+                await withdrawCollateral({
+                    user: userThree,
+                    asset: this.collateral,
+                    amount: deposits,
+                });
+            });
+
             it("should liquidate up to LT with a single CDP", async
function () {
                 const userThree = hre.users.userThree;
                 const deposits = toBig(15);
```

Test case 2

```
diff --git a/src/test/minter/04-mint-repay.ts
b/src/test/minter/04-mint-repay.ts
index 2d71a64..8c64059 100644
--- a/src/test/minter/04-mint-repay.ts
+++ b/src/test/minter/04-mint-repay.ts
@@ -25,12 +25,13 @@ import {
     KreskoAssetMintedEventObject,
     OpenFeePaidEventObject,
 } from "types/typechain/src/contracts/libs/Events.sol/MinterEvent";
+import { KreskoAssetAnchor } from
"types/typechain/src/contracts/kreskoasset/KreskoAssetAnchor";

 const INTEREST_RATE_DELTA = toBig("0.000001");
 const INTEREST_RATE_PRICE_DELTA = toBig("0.0001", 8);

 describe("Minter", () => {
-    withFixture(["minter-test"]);
+    withFixture(["minter-test", "kresko-assets"]);
```

```
     beforeEach(async function () {
         this.collateral = this.collaterals.find(c => c.deployArgs!.name
=== defaultCollateralArgs.name)!;
@@ -90,6 +91,78 @@ describe("Minter", () => {

expect(kreskoAssetTotalSupplyAfter.eq(kreskoAssetTotalSupplyBefore.add(mint
Amount)));
             });

+            // @audit-issue KreskoAssetAnchor accounting differences
between issue/destroy and deposit/mint/withdraw/redeem break system
invariants
+            it.only("should mint, deposit, withdraw, and have the same
amount", async function () {
+                // Initially the Kresko asset's total supply should be 0
+                const kreskoAssetTotalSupplyBefore = await
this.krAsset.contract.totalSupply();
+                expect(kreskoAssetTotalSupplyBefore).to.equal(0);
+                // Initially, the array of the user's minted kresko assets
should be empty.
+                const mintedKreskoAssetsBefore = await
hre.Diamond.getMintedKreskoAssets(hre.users.userOne.address);
+                expect(mintedKreskoAssetsBefore).to.deep.equal([]);
+
+                // Mint Kresko asset
+                const mintAmount = toBig(10);
+                await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).mintKreskoAsset(
+                    hre.users.userOne.address,
+                    this.krAsset.address,
+                    mintAmount,
+                );
+
+                // Confirm the array of the user's minted Kresko assets
has been pushed to.
+                const mintedKreskoAssetsAfter = await
hre.Diamond.getMintedKreskoAssets(hre.users.userOne.address);
+
```

```
expect(mintedKreskoAssetsAfter).to.deep.equal([this.krAsset.address]);
+                // Confirm the amount minted is recorded for the user.
+                const amountMinted = await
hre.Diamond.kreskoAssetDebt(hre.users.userOne.address,
this.krAsset.address);
+                expect(amountMinted).to.equal(mintAmount);
+                // Confirm the user's Kresko asset balance has increased
+                const userBalance = await
this.krAsset.mocks.contract.balanceOf(hre.users.userOne.address);
+                expect(userBalance).to.equal(mintAmount);
+                // Confirm that the Kresko asset's total supply increased
as expected
+                const kreskoAssetTotalSupplyAfter = await
this.krAsset.contract.totalSupply();
+
expect(kreskoAssetTotalSupplyAfter.eq(kreskoAssetTotalSupplyBefore.add(mint
Amount)));
+                // @audit up to this here, this test is the same as the
previous one
+
+                // @audit now start the specific issue
+                const KreskoAsset: KreskoAsset = this.krAsset.contract;
+                const KreskoAssetAnchor: KreskoAssetAnchor =
this.krAsset.anchor;
+
+                // @audit before
+                expect(await
KreskoAsset.balanceOf(hre.users.userOne.address)).to.equal(mintAmount);
+                expect(await
KreskoAsset.balanceOf(KreskoAssetAnchor.address)).to.equal(0);
+                expect(await
KreskoAssetAnchor.balanceOf(hre.users.userOne.address)).to.equal(0);
+
+                // @audit deposit to KreskoAssetAnchor
+                console.log("approve");
+                await wrapContractWithSigner(KreskoAsset,
hre.users.userOne).approve(
+                    KreskoAssetAnchor.address,
+                    mintAmount,
```

```
+                );
+                console.log("deposit");
+                await wrapContractWithSigner(KreskoAssetAnchor,
hre.users.userOne).deposit(
+                    mintAmount,
+                    hre.users.userOne.address,
+                );
+                expect(await
KreskoAsset.balanceOf(hre.users.userOne.address)).to.equal(0);
+                expect(await
KreskoAsset.balanceOf(KreskoAssetAnchor.address)).to.equal(mintAmount);
+                expect(await
KreskoAssetAnchor.balanceOf(hre.users.userOne.address)).to.equal(mintAmount
);
+
+                // @audit withdraw from KreskoAssetAnchor
+                console.log("withdraw");
+                await wrapContractWithSigner(KreskoAssetAnchor,
hre.users.userOne).withdraw(
+
KreskoAssetAnchor.maxWithdraw(hre.users.userOne.address),
+                    hre.users.userOne.address,
+                    hre.users.userOne.address,
+                );
+
+                // @audit after
+
+                // @audit-issue the following will pass, but it is a bug.
The user lost 50% of their assets just by depositing/withdrawing
+                expect(await
KreskoAsset.balanceOf(hre.users.userOne.address)).to.equal(mintAmount.div(2
));
+                // @audit-issue the following will pass, but it is a bug.
The anchor should not have any assets left
+                expect(await
KreskoAsset.balanceOf(KreskoAssetAnchor.address)).to.equal(mintAmount.div(2
));
+                expect(await
KreskoAssetAnchor.balanceOf(hre.users.userOne.address)).to.equal(0);
```

```
+            });
+
             it("should allow successive, valid mints of the same Kresko
asset", async function () {
                 // Initially the Kresko asset's total supply should be 0
                 const kreskoAssetTotalSupplyInitial = await
this.krAsset.contract.totalSupply();
```

Test case 3

```
diff --git a/src/test/minter/02-deposit-withdraw.ts
b/src/test/minter/02-deposit-withdraw.ts
index 7308f65..be53569 100644
--- a/src/test/minter/02-deposit-withdraw.ts
+++ b/src/test/minter/02-deposit-withdraw.ts
@@ -275,13 +275,13 @@ describe("Minter - Deposit Withdraw", () => {
         describe("#withdraw", () => {
             beforeEach(async function () {
                 // Deposit collateral
-                await expect(
-                    wrapContractWithSigner(hre.Diamond,
this.depositArgs.user).depositCollateral(
-                        this.depositArgs.user.address,
-                        this.collateral.contract.address,
-                        this.depositArgs.amount,
-                    ),
-                ).not.to.be.reverted;
+                // await expect(
+                //     wrapContractWithSigner(hre.Diamond,
this.depositArgs.user).depositCollateral(
+                //         this.depositArgs.user.address,
+                //         this.collateral.contract.address,
+                //         this.depositArgs.amount,
+                //     ),
+                // ).not.to.be.reverted;

                 this.collateral = this.collaterals![0];
                 this.depositAmount = this.depositArgs.amount;
```

```
@@ -322,6 +322,129 @@ describe("Minter - Deposit Withdraw", () => {
                    expect(userOneBalance).to.equal(this.initialBalance);
                });

+                it.only("should allow an account to withdraw almost all
asset collateral and cause dust deposit", async function () {
+                    this.krAsset = this.krAssets!.find(k =>
k.deployArgs!.name === defaultKrAssetArgs.name)!;
+
+                    // grant operator role to deployer for rebases
+                    await this.krAsset!.contract.grantRole(Role.OPERATOR,
hre.users.deployer.address);
+                    const assetInfo = await this.krAsset!.kresko();
+
+                    // Add krAsset as a collateral with anchor and cFactor
of 1
+                    await wrapContractWithSigner(hre.Diamond,
hre.users.deployer).addCollateralAsset(
+                        this.krAsset!.contract.address,
+                        await getCollateralConfig(
+                            this.krAsset.contract,
+                            this.krAsset!.anchor!.address,
+                            toBig(1),
+                            toBig(1.05),
+                            assetInfo.oracle,
+                            assetInfo.oracle,
+                        ),
+                    );
+
+                    /* Get some krAssets by minting via user three */
+
+                    const arbitraryUser: SignerWithAddress =
hre.users.userThree;
+                    await
this.collateral.mocks!.contract.setVariable("_balances", {
+                        [arbitraryUser.address]: this.initialBalance,
+                    });
+                    await
this.collateral.mocks!.contract.setVariable("_allowances", {
```

```
+                        [arbitraryUser.address]: {
+                            [hre.Diamond.address]: this.initialBalance,
+                        },
+                    });
+
+                    // Allowance for Kresko
+                    await
this.krAsset!.contract.connect(arbitraryUser).approve(
+                        hre.Diamond.address,
+                        hre.ethers.constants.MaxUint256,
+                    );
+
+                    // Deposit some collateral
+                    await wrapContractWithSigner(hre.Diamond,
arbitraryUser).depositCollateral(
+                        arbitraryUser.address,
+                        this.collateral.address,
+                        this.depositArgs.amount,
+                    );
+
+                    const mintAmount = toBig(100);
+
+                    // Mint some krAssets
+                    await wrapContractWithSigner(hre.Diamond,
arbitraryUser).mintKreskoAsset(
+                        arbitraryUser.address,
+                        this.krAsset!.address,
+                        mintAmount,
+                    );
+
+                    // send krAsset from user three to user one
+                    await
this.krAsset!.contract.connect(arbitraryUser).transfer(
+                        hre.users.userOne.address,
+                        mintAmount,
+                    );
+
+                    // asset user one received krAsset from user three
+                    const userOneKrAssetBalance = await
```

```
this.krAsset!.contract.balanceOf(hre.users.userOne.address);
+                    expect(userOneKrAssetBalance).to.equal(mintAmount);
+
+                    /* Start */
+
+                    // // Allowance for Kresko
+                    await
this.krAsset!.contract.connect(hre.users.userOne).approve(
+                        hre.Diamond.address,
+                        hre.ethers.constants.MaxUint256,
+                    );
+
+                    // Deposit some collateral
+                    await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).depositCollateral(
+                        hre.users.userOne.address,
+                        this.krAsset.address,
+                        mintAmount,
+                    );
+
+                    // Rebase the asset according to params
+                    const denominator = 4;
+                    const positive = true;
+
+                    await
this.krAsset!.contract.rebase(toBig(denominator), positive, []);
+
+                    const rebasedMintAmount = mintAmount.mul(denominator);
+                    const withdrawAmount = rebasedMintAmount.sub(1);
+
+                    await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).withdrawCollateral(
+                        hre.users.userOne.address,
+                        this.krAsset.address,
+                        withdrawAmount,
+                        0, // The index of this.collateral.address in the
account's depositedCollateralAssets
+                    );
+
```

```
+                    // Ensure the change in the user's deposit is
recorded.
+                    const amountDeposited = await
hre.Diamond.collateralDeposits(
+                        hre.users.userOne.address,
+                        this.krAsset.address,
+                    );
+                    expect(amountDeposited).to.equal(0); // @audit-info
It's rounded down to 0
+
+                    // Ensure that the collateral asset is still in the
account's deposited collateral
+                    // assets array.
+                    let depositedCollateralAssets = await
hre.Diamond.getDepositedCollateralAssets(
+                        hre.users.userOne.address,
+                    );
+
expect(depositedCollateralAssets).to.deep.equal([this.krAsset.address]);
+
+
+                    // Deposit once more some krAsset collateral
+                    await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).depositCollateral(
+                        hre.users.userOne.address,
+                        this.krAsset.address,
+                        toBig(1),
+                    );
+
+                    depositedCollateralAssets = await
hre.Diamond.getDepositedCollateralAssets(
+                        hre.users.userOne.address,
+                    );
+
expect(depositedCollateralAssets).to.deep.equal([this.krAsset.address,
this.krAsset.address]); // @audit-info duplicate collateral asset
+                });
+
                it("should allow an account to withdraw a portion of their
```

```
deposit", async function () {
                    const withdrawAmount = this.depositAmount.div(2);

@@ -660,6 +783,7 @@ describe("Minter - Deposit Withdraw", () => {
                        arbitraryUser.address,
                        this.krAsset!.address,
                    );
+
                    // Ensure that the collateral balance is adjusted by
the rebase

expect(depositsBefore).to.not.bignumber.equal(finalDeposits);

expect(finalDeposits).to.bignumber.equal(expectedDepositsAfter);
```

Test case 4

```
diff --git a/src/test/minter/04-mint-repay.ts
b/src/test/minter/04-mint-repay.ts
index 2d71a64..c11a94d 100644
--- a/src/test/minter/04-mint-repay.ts
+++ b/src/test/minter/04-mint-repay.ts
@@ -25,7 +25,7 @@ import {
     KreskoAssetMintedEventObject,
     OpenFeePaidEventObject,
 } from "types/typechain/src/contracts/libs/Events.sol/MinterEvent";
-
+import { mine } from "@nomicfoundation/hardhat-network-helpers";
 const INTEREST_RATE_DELTA = toBig("0.000001");
 const INTEREST_RATE_PRICE_DELTA = toBig("0.0001", 8);

@@ -793,6 +793,90 @@ describe("Minter", () => {
             });
         });

+        describe("#burn cause duplicates", () => {
+            beforeEach(async function () {
+                // Create userOne debt position
```

```
+                this.mintAmount = toBig(20);
+
+                // Load userThree with Kresko Assets
+                await
this.collateral.mocks!.contract.setVariable("_balances", {
+                    [hre.users.userThree.address]: this.initialBalance,
+                });
+                await
this.collateral.mocks!.contract.setVariable("_allowances", {
+                    [hre.users.userThree.address]: {
+                        [hre.Diamond.address]: this.initialBalance,
+                    },
+                });
+                expect(await
this.collateral.contract.balanceOf(hre.users.userThree.address)).to.equal(
+                    this.initialBalance,
+                );
+                await expect(
+                    wrapContractWithSigner(hre.Diamond,
hre.users.userThree).depositCollateral(
+                        hre.users.userThree.address,
+                        this.collateral.address,
+                        toBig(10000),
+                    ),
+                ).not.to.be.reverted;
+
+                await wrapContractWithSigner(hre.Diamond,
hre.users.userThree).mintKreskoAsset(
+                    hre.users.userThree.address,
+                    this.krAsset.address,
+                    this.mintAmount,
+                );
+
+                await hre.network.provider.send("evm_setAutomine",
[false]);
+
+                await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).mintKreskoAsset(
+                    hre.users.userOne.address,
```

```
+                    this.krAsset.address,
+                    this.mintAmount,
+                );
+            });
+
+            after(async function () {
+                await hre.network.provider.send("evm_setAutomine",
[true]);
+            });
+
+            it.only("should be able to mint and repay within same
transaction to cause duplicate minted kresko assets", async function () {
+                // Burn Kresko asset
+                const burnAmount = toBig(20);
+                const kreskoAssetIndex = 0;
+
+                await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).burnKreskoAsset(
+                    hre.users.userOne.address,
+                    this.krAsset.address,
+                    burnAmount,
+                    kreskoAssetIndex,
+                );
+
+                // mine a new block
+                await mine();
+
+                // Confirm the user no long holds the burned Kresko asset
amount
+                const userBalance = await
this.krAsset.contract.balanceOf(hre.users.userOne.address);
+
expect(userBalance).to.equal(this.mintAmount.sub(burnAmount));
+
+                // Confirm the array of the user's minted Kresko assets
still contains the asset's address
+                let mintedKreskoAssetsAfter = await
hre.Diamond.getMintedKreskoAssets(hre.users.userOne.address);
+
```

```
expect(mintedKreskoAssetsAfter).to.deep.equal([this.krAsset.address]);
+                expect(mintedKreskoAssetsAfter.length).to.equal(1);
+
+                await wrapContractWithSigner(hre.Diamond,
hre.users.userOne).mintKreskoAsset(
+                    hre.users.userOne.address,
+                    this.krAsset.address,
+                    this.mintAmount,
+                );
+
+                // mine a new block
+                await mine();
+
+                mintedKreskoAssetsAfter = await
hre.Diamond.getMintedKreskoAssets(hre.users.userOne.address);
+
+                expect(mintedKreskoAssetsAfter.length).to.equal(2);
+
expect(mintedKreskoAssetsAfter).to.deep.equal([this.krAsset.address,
this.krAsset.address]);
+            });
+        });
+
         describe("#burn", () => {
             beforeEach(async function () {
                 // Create userOne debt position
```

Test case 5

```
diff --git a/src/test/minter/03-liquidation.ts
b/src/test/minter/03-liquidation.ts
index b50cb31..fe9656d 100644
--- a/src/test/minter/03-liquidation.ts
+++ b/src/test/minter/03-liquidation.ts
@@ -454,6 +454,69 @@ describe("Minter", () => {
                 expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.false;
```

```
            });

+           it.only("should not allow liquidator to pay more debt than
necessary", async function () {
+               const collateral2 = await addMockCollateralAsset({
+                   name: "Collateral",
+                   decimals: 18,
+                   factor: 1,
+                   price: 10,
+               });
+
+               const userThree = hre.users.userThree;
+               const [deposits1, deposits2] = [toBig(10), toBig(5)];
+               const borrows = toBig(10);
+
+               this.collateral.setPrice(10);
+               this.krAsset.setPrice(10);
+
+               await Promise.all([
+                   await this.collateral.setBalance(userThree,
deposits1),
+                   await collateral2.setBalance(userThree, deposits2),
+                   await depositCollateral({
+                       user: userThree,
+                       amount: deposits1,
+                       asset: this.collateral,
+                   }),
+                   await depositCollateral({
+                       user: userThree,
+                       amount: deposits2,
+                       asset: collateral2,
+                   }),
+                   await mintKrAsset({
+                       user: userThree,
+                       amount: borrows,
+                       asset: this.krAsset,
+                   }),
+               ]);
+
```

```
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.false;
+
+                // seemingly random order of updates to test that the
liquidation works regardless
+                // @audit `debt` will be fully repaid but liquidator will
receive only what's left of `collateral`
+                this.collateral.setPrice(6.75);
+                await collateral2.update({
+                    factor: 0.975,
+                    name: "updated",
+                });
+                await this.krAsset.update({
+                    factor: 1.05,
+                    name: "updated",
+                    closeFee: 0.02,
+                    openFee: 0,
+                    supplyLimit: 1_000_000,
+                });
+
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.true;
+
+                const ans = await liquidate(userThree, this.krAsset,
this.collateral);
+
+                expect(await
getCR(userThree.address)).to.be.lessThan(1.4);
+                expect(await
hre.Diamond.isAccountLiquidatable(userThree.address)).to.be.true;
+
+                // @audit-issue this will revert, as liquidator has
received less collateral (10) than what he was entitled (10.5), meaning
that he spent more kresko assets repaying debt than necessary
+                expect(ans.amountToSeize).to.equal(ans.collateralSeized);
+            });
+
             it("should liquidate up to LT with multiple CDPs", async
function () {
```

```
                const collateral2 = await addMockCollateralAsset({
                    name: "Collateral",
diff --git a/src/utils/test/helpers/liquidations.ts
b/src/utils/test/helpers/liquidations.ts
index 8555d04..20129c6 100644
--- a/src/utils/test/helpers/liquidations.ts
+++ b/src/utils/test/helpers/liquidations.ts
@@ -127,6 +127,18 @@ export const liquidate = async (user:
SignerWithAddress, krAsset: any, collatera
         amount: liquidationAmount,
     });

+    const collateralDepositValue = await
hre.Diamond.getCollateralValueAndOraclePrice(
+        collateral.address,
+        await hre.Diamond.collateralDeposits(user.address,
collateral.address),
+        false,
+    );
+
+    const collateralAsset = await
hre.Diamond.collateralAsset(collateral.address);
+
+    const amountToSeize = liquidationAmount
+        .wadMul(collateralAsset.liquidationIncentive)
+        .wadDiv(collateralDepositValue.oraclePrice);
+
     const tx = await wrapContractWithSigner(hre.Diamond,
hre.users.liquidator).liquidate(
         user.address,
         krAsset.address,
@@ -140,6 +152,7 @@ export const liquidate = async (user:
SignerWithAddress, krAsset: any, collatera
     return {
         collateralSeized: fromBig(depositsBefore.sub(depositsAfter), await
collateral.contract.decimals()),
         debtRepaid: fromBig(debtBefore.sub(debtAfter), 18),
+        amountToSeize: fromBig(amountToSeize, 27),
         // tx,
```

```
    };
};
```

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