Korea Gold Exchange Digital Asset

Security Assessment

CertiK Assessed on May 21st, 2026

CertiK Assessed on May 21st, 2026

Korea Gold Exchange Digital Asset

The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	EVM Compatible	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE
Solidity	Preliminary comments published on 03/20/2026 Final report published on 05/21/2026

Vulnerability Summary

37	28	2	7	0
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined

Count	Category	Status	Description
4	Centralization	4 Acknowledged	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
0	Major		Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
5	Medium	5 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
8	Minor	8 Resolved	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
20	Informational	15 Resolved, 2 Partially Resolved, 3 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | KOREA GOLD EXCHANGE DIGITAL ASSET

Audit Summary

Executive Summary

Vulnerability Summary

Codebase

Audit Scope

Approach & Methods

Review Notes

Dependencies

Third Party Dependencies

Assumptions

Findings

KGE-02 : Two-Step Upgrade Pattern In `CommodityToken`

KGE-06 : Centralization Risk

KGE-28 : `setOrderStatus()` Enables Arbitrary Lifecycle Rewind

KGE-29 : Manager Controls Token Split With No User Commitment

KGE-07 : Zero-Address `MINT_APPROVER_ROLE` Enables Signature Bypass In `mintWithAuthorization()`

KGE-08 : Mandatory Inline `permit()` In `issue()` Is Griefable Via Front-Running

KGE-09 : `lockTime` Recorded But Not Enforced

KGE-23 : `cancelAuthorization()` Is Replayable And Emits Ambiguous `AuthorizationUsed` Events

KGE-30 : Frozen Accounts Cannot Revoke Allowances; Stale Allowances Survive Wipe

KGE-10 : Signature Malleability Of `ecrecover`

KGE-11 : Paused State Prevents Allowance Reduction/Revocation

KGE-12 : Zero-Output Issuances Are Not Rejected

KGE-13 : `getAmountIn()` Uses Inefficient And Unsound Search Bounds

KGE-14 : Missing Input Validations

KGE-15 : Freeze Status Can Be Changed Without Emitting `Frozen` / `Unfrozen` Events

KGE-31 : Premature Decimal Normalization In `getAmountOut` Causes Systematic Output Underestimation

KGE-33 : `getAmountIn()` Has Unenforced Input And Configuration Preconditions Causing Either Gas Exhaustion Or Division-By-Zero

KGE-03 : User-Provided `extraCost` Is Unverified

KGE-04 : `RedeemLock` Order Lifecycle Has No On-Chain Fulfillment Enforcement

KGE-16 : Missing Emit Events

KGE-17 : Hardcoded Role Hash Reduces Readability And Maintainability

KGE-18 : Redundant Event Emission

KGE-19 : Inconsistency Between Comment And Code

KGE-20 : Missing Validation Of Authorization Time Window

KGE-21 : Unnecessary `userOrders` Zero-Check In `RedeemLock()

KGE-22 : `vaultedWeight` Is Non-Operational And Unsigned In `mintWithAuthorization()`

KGE-24 : Invalid Use Of Access Control Modifier

KGE-25 : Redundant `ERC20Upgradeable` And `ERC20PermitUpgradeable` Inheritance

KGE-26 : Minor Typos

KGE-27 : AuthorizationUsed Event Misattributes Nonce To Caller (Msg.Sender) Instead Of Authorizer (Signer)

KGE-32 : Inclusive Time Bounds Deviate From EIP-3009 Reference

KGE-34 : Issuer Validates Whitelist But Not Freeze Status, Granting Users A Costless Unilateral Abort On Signed Orders

KGE-35 : Withdraw Paths Trust Nominal Amounts: Token Onboarding Must Exclude Asymmetric-Debit Semantics

KGE-36 : `renounceRole()` Override Does Not Protect `DEFAULT_ADMIN_ROLE`

KGE-37 : `mintWithAuthorization()` Shares The EIP-3009 Nonce Bucket

KGE-38 : `sub256()` Helper Refactor Reduces Clarity And Reintroduces Avoidable Cost

KGE-39 : `wipeFrozenAccount()` Cannot Execute During Pause

Formal Verification

Considered Functions And Scope

Verification Results

Appendix

Disclaimer

CODEBASE | KOREA GOLD EXCHANGE DIGITAL ASSET

Repository

https://github.com/CrederLabs/KGLD_contract

Commit

d5bd05976a550052fb2fd2897788c694eafefc0e

AUDIT SCOPE | KOREA GOLD EXCHANGE DIGITAL ASSET

CrederLabs/KGLD_contract
contracts/CommodityToken/CommodityToken.sol
contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol
contracts/RedeemLock/RedeemLock.sol
contracts/CommodityToken/CommodityTokenProxy.sol

APPROACH & METHODS | KOREA GOLD EXCHANGE DIGITAL ASSET

This audit was conducted for Korea Gold Exchange Digital Asset to evaluate the security and correctness of the smart contracts associated with the Korea Gold Exchange Digital Asset project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | KOREA GOLD EXCHANGE DIGITAL ASSET

Dependencies

Third Party Dependencies

The protocol is serving as the underlying entity to interact with third party protocols. The third parties that the contracts interact with are:

- ERC20 Tokens

The scope of the audit treats third party entities as black boxes and assumes their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets. Moreover, updates to the state of a project contract that are dependent on the read of the state of external third party contracts may make the project vulnerable to read-only reentrancy. In addition, upgrades of third parties can possibly create severe impacts, such as introducing fees, etc.

Assumptions

Within the scope of the audit, assumptions are made about the intended behavior of the protocol in order to inspect consequences based on those behaviors. Assumptions made within the scope of this audit include:

- `RedeemLock.redeemToken()` is assumed to be the in-house CommodityToken, and any future upgrade to CommodityToken is assumed to preserve atomic ERC20 transfer semantics (no fee-on-transfer, no partial transfers, no asymmetric debit). Non-atomic transfer semantics introduced via upgrade would silently corrupt order accounting in RedeemLock.
- Token onboarding excludes fee-on-transfer (sender- or recipient-side), deflationary, rebasing, hook-based, and other non-standard ERC20 implementations; only tokens with exact-amount transfer semantics on both sender and recipient sides are paired into CommodityTokenIssuer. The existing on-chain delta-checks act as safeguards not a substitute.

Beyond initial onboarding, the team is assumed to continuously monitor token implementations — including upgrades to proxy-deployed tokens (USDC, USDT, etc.) and parameter changes that could activate dormant fee logic — and to remove or pause affected pairs immediately upon any change to transfer semantics.

FINDINGS | KOREA GOLD EXCHANGE DIGITAL ASSET

This report has been prepared for Korea Gold Exchange Digital Asset to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 37 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
KGE-02	Two-Step Upgrade Pattern In `CommodityToken`	Governance	Centralization	Acknowledged
KGE-06	**Centralization Risk**	**Centralization**	**Centralization**	**Acknowledged**
KGE-28	**`setOrderStatus()` Enables Arbitrary Lifecycle Rewind**	**Design Issue, Centralization**	**Centralization**	**Acknowledged**
KGE-29	**Manager Controls Token Split With No User Commitment**	**Design Issue, Centralization**	**Centralization**	**Acknowledged**
KGE-07	Zero-Address `MINT_APPROVER_ROLE` Enables Signature Bypass In `mintWithAuthorization()`	Access Control	Medium	Resolved
KGE-08	Mandatory Inline `permit()` In `issue()` Is Griefable Via Front-Running	Denial of Service, Coding Issue	Medium	Resolved
KGE-09	`lockTime` Recorded But Not Enforced	Volatile Code, Design Issue	Medium	Resolved
KGE-23	`cancelAuthorization()` Is Replayable And Emits Ambiguous `AuthorizationUsed` Events	Volatile Code	Medium	Resolved
KGE-30	Frozen Accounts Cannot Revoke Allowances; Stale Allowances Survive Wipe	Volatile Code	Medium	Resolved

ID	Title	Category	Severity	Status
KGE-10	Signature Malleability Of `ecrecover`	Volatile Code	Minor	Resolved
KGE-11	Paused State Prevents Allowance Reduction/Revocation	Design Issue	Minor	Resolved
KGE-12	Zero-Output Issuances Are Not Rejected	Volatile Code	Minor	Resolved
KGE-13	`getAmountIn()` Uses Inefficient And Unsound Search Bounds	Volatile Code, Design Issue, Gas Optimization	Minor	Resolved
KGE-14	Missing Input Validations	Volatile Code	Minor	Resolved
KGE-15	Freeze Status Can Be Changed Without Emitting `Frozen` / `Unfrozen` Events	Volatile Code	Minor	Resolved
KGE-31	Premature Decimal Normalization In `getAmountOut` Causes Systematic Output Underestimation	Incorrect Calculation	Minor	Resolved
KGE-33	`getAmountIn()` Has Unenforced Input And Configuration Preconditions Causing Either Gas Exhaustion Or Division-By-Zero	Volatile Code	Minor	Resolved
KGE-03	User-Provided `extraCost` Is Unverified	Design Issue	Informational	Partially Resolved
KGE-04	`RedeemLock` Order Lifecycle Has No On-Chain Fulfillment Enforcement	Volatile Code, Design Issue	Informational	Acknowledged
KGE-16	Missing Emit Events	Coding Style	Informational	Resolved
KGE-17	Hardcoded Role Hash Reduces Readability And Maintainability	Coding Issue	Informational	Resolved
KGE-18	Redundant Event Emission	Coding Issue	Informational	Resolved

ID	Title	Category	Severity	Status
KGE-19	Inconsistency Between Comment And Code	Inconsistency	Informational	Resolved
KGE-20	Missing Validation Of Authorization Time Window	Volatile Code	Informational	Resolved
KGE-21	Unnecessary `userOrders` Zero-Check In `RedeemLock()	Coding Issue	Informational	Resolved
KGE-22	`vaultedWeight` Is Non-Operational And Unsigned In `mintWithAuthorization()`	Design Issue	Informational	Resolved
KGE-24	Invalid Use Of Access Control Modifier	Logical Issue	Informational	Resolved
KGE-25	Redundant `ERC20Upgradeable` And `ERC20PermitUpgradeable` Inheritance	Coding Issue	Informational	Resolved
KGE-26	Minor Typos	Coding Issue	Informational	Resolved
KGE-27	AuthorizationUsed Event Misattributes Nonce To Caller (Msg.Sender) Instead Of Authorizer (Signer)	Inconsistency	Informational	Resolved
KGE-32	Inclusive Time Bounds Deviate From EIP-3009 Reference	Design Issue	Informational	Resolved
KGE-34	Issuer Validates Whitelist But Not Freeze Status, Granting Users A Costless Unilateral Abort On Signed Orders	Volatile Code	Informational	Acknowledged
KGE-35	Withdraw Paths Trust Nominal Amounts: Token Onboarding Must Exclude Asymmetric-Debit Semantics	Volatile Code	Informational	Partially Resolved
KGE-36	`renounceRole()` Override Does Not Protect `DEFAULT_ADMIN_ROLE`	Volatile Code	Informational	Resolved

ID	Title	Category	Severity	Status
KGE-37	`mintWithAuthorization()` Shares The EIP-3009 Nonce Bucket	Volatile Code	Informational	Acknowledged
KGE-38	`sub256()` Helper Refactor Reduces Clarity And Reintroduces Avoidable Cost	Coding Issue, Design Issue	Informational	Resolved
KGE-39	`wipeFrozenAccount()` Cannot Execute During Pause	Volatile Code	Informational	Resolved

KGE-02 | Two-Step Upgrade Pattern In `CommodityToken`

Category	Severity	Location	Status
Governance	● Centralization	contracts/CommodityToken/CommodityToken.sol (base): 443~445	● Acknowledged

Description

The upgrade flow is designed to appear as a two-step approval process:

- `updateAuditedImpl()` allows `UPGRADE_AUDITOR_ROLE` to pre-authorize a new implementation.
- `_authorizeUpgrade()` allows `UPGRADER_ROLE` to execute the upgrade on a pre-authorized implementation.

However, this separation is not enforced on-chain, as nothing prevents an address from being granted both `UPGRADE_AUDITOR_ROLE` and `UPGRADER_ROLE` . This will allow the address to approve any implementation and upgrade to it unilaterally. In that configuration, the intended auditor check becomes self-approval rather than an independent control.

Recommendation

We recommend enforcing independent upgrade approval on-chain rather than relying on operational assumptions. Moreover, a time delay could be enforced between implementation approval and actual upgrade.

Additionally, we recommend applying the centralization mitigation strategy to each privileged role.

Alleviation

[Korea Gold Exchange Digital Asset , 03/22/2026]: When an implementation upgrade is required, we plan to follow the process below:

- Share the new implementation code and deployment address with the auditor, and grant the UPGRADE_AUDITOR_ROLE
- The auditor reviews the code and approves the new implementation address
- The holder of the UPGRADER_ROLE executes the upgrade
- Revoke the UPGRADE_AUDITOR_ROLE after the upgrade is completed

We would appreciate it if you could review the contract for any potential vulnerabilities in this upgrade process.

[CertiK, 04/02/2026]: Thanks for the comment. Revoking UPGRADE_AUDITOR_ROLE after the upgrade is completed is one of the options. If the team would like to maintain the role, timelock+multisig wallet can be considered as another option to manage the UPGRADE_AUDITOR_ROLE role

---

[CertiK, 04/28/2026]: The on-chain auditor/upgrader separation (`InvalidUpgrader` check in `isValidUpgrade`) and the 72-hour approval staleness bound are correctly implemented in commit `dcc80f1` . However, the recommended timelock

between approval and upgrade remains absent and is deferred by the team for regulatory-response flexibility. We cannot currently verify that it will be enforced.

[CertiK, 04/14/2026]: As an additional note on reentrancy protection, the client stands with the current implementation, considering that the audit gate and initializer protection provide sufficient protection against practical exploit scenarios without adding a reentrancy guard. We accept this position on record.

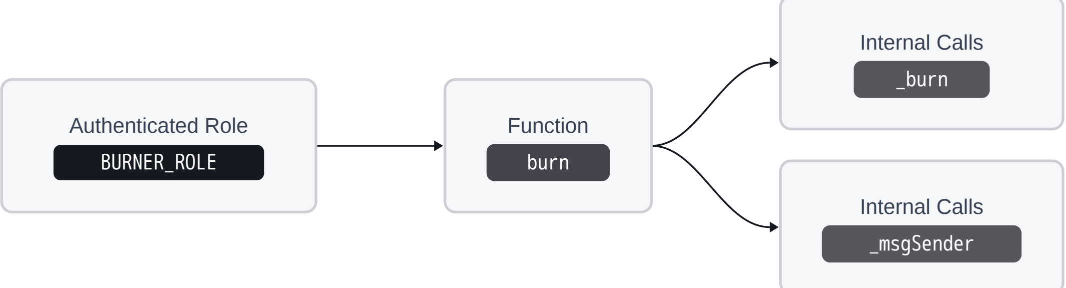
KGE-06 | Centralization Risk

Category	Severity	Location	Status
Centralization	Centralization	contracts/CommodityToken/CommodityToken.sol (base): 102, 108, 131, 252, 260, 276, 429, 443; contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (base): 45, 52, 109, 128, 140, 168, 274, 398, 420; contracts/RedeemLock/RedeemLock.sol (base): 120, 137, 159, 203, 215	Acknowledged

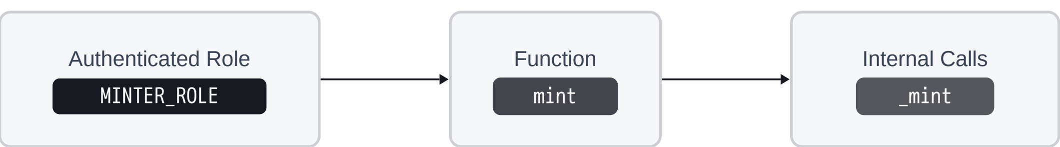
Description

CommodityToken

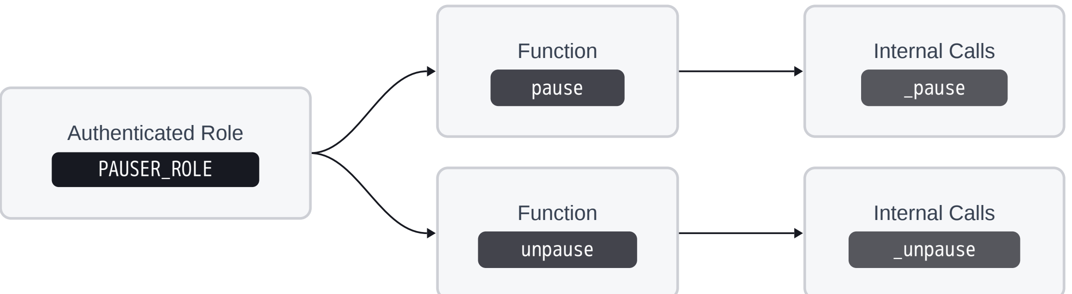
In the contract `CommodityToken` the role `BURNER_ROLE` has authority over the functions shown in the diagram below. Any compromise of the `BURNER_ROLE` account may allow the hacker to exploit this authority and burn tokens held by that same privileged account, resulting in a loss of funds controlled by the burner wallet and an unintended reduction in total supply.



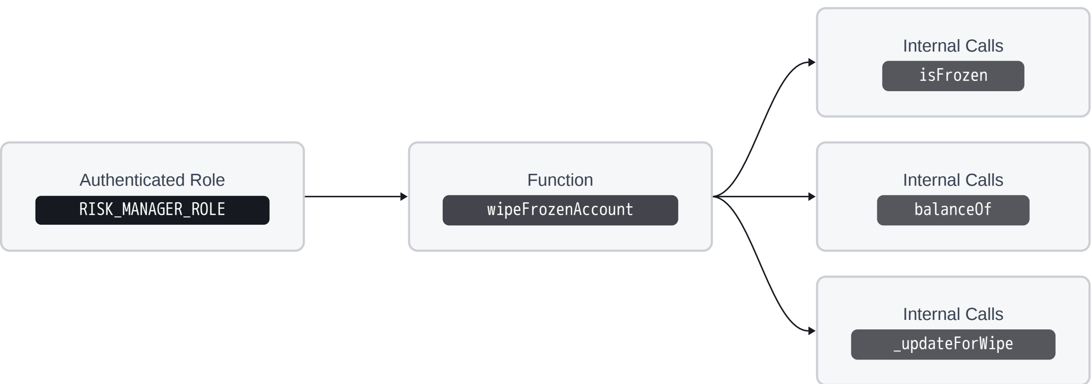
In the contract `CommodityToken` the role `MINTER_ROLE` has authority over the functions shown in the diagram below. Any compromise of the `MINTER_ROLE` account may allow the hacker to exploit this authority and mint arbitrary amounts of tokens to arbitrary addresses.



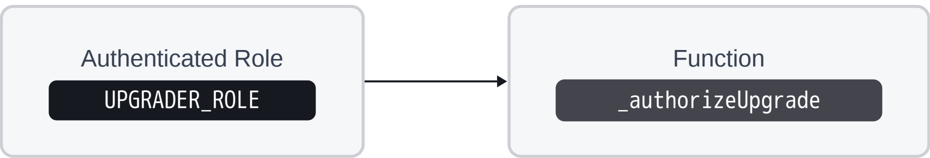
In the contract `CommodityToken` the role `PAUSER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `PAUSER_ROLE` account may allow the hacker to take advantage of this authority and pause or unpause the contract.



In the contract `CommodityToken` the role `RISK_MANAGER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `RISK_MANAGER_ROLE` account may allow the hacker to take advantage of this authority and freeze and unfreeze arbitrary accounts and to wipe the balances of frozen accounts.

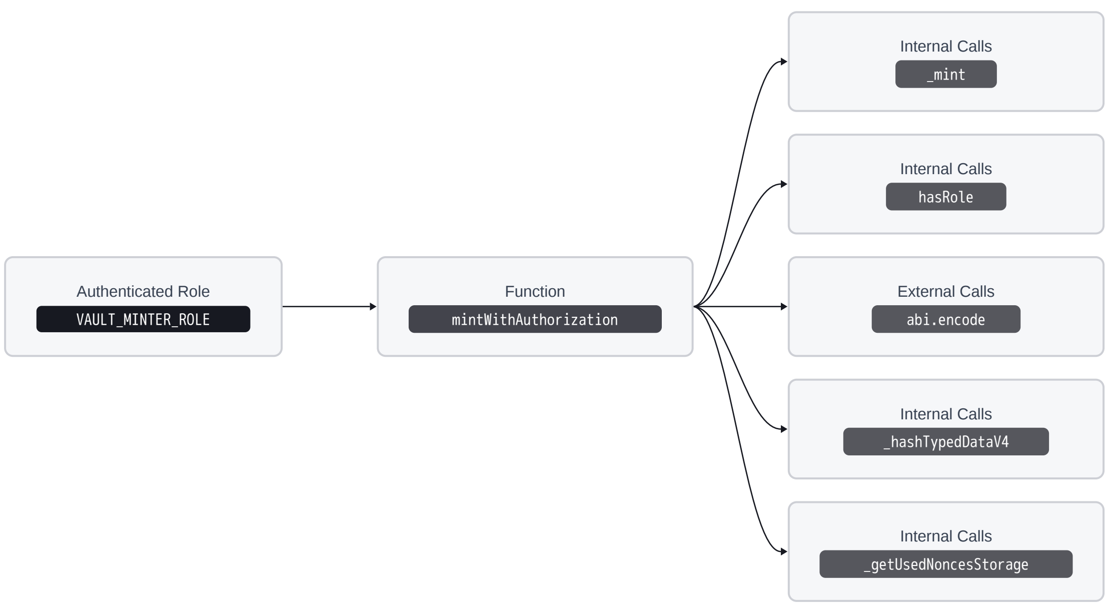


In the contract `CommodityToken` the role `UPGRADER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `UPGRADER_ROLE` account may allow the hacker to take advantage of this authority and execute upgrades to implementations that have already been approved through the audit gate, including activating such upgrades at unintended times. If role separation is not maintained, or if the attacker also gains control of `UPGRADE_AUDITOR_ROLE` , this may lead to an upgrade to a malicious implementation and full compromise of the proxied contract.



In the contract `CommodityToken` the role `UPGRADE_AUDITOR_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `UPGRADE_AUDITOR_ROLE` account may allow the hacker to take advantage of this authority and may allow a hacker to designate arbitrary implementations as audited and therefore eligible for future upgrade by `UPGRADER_ROLE` . If role separation is not maintained, or if the attacker also gains control of `UPGRADE_AUDITOR_ROLE` , this may lead to an upgrade to a malicious implementation and full compromise of the proxied contract.

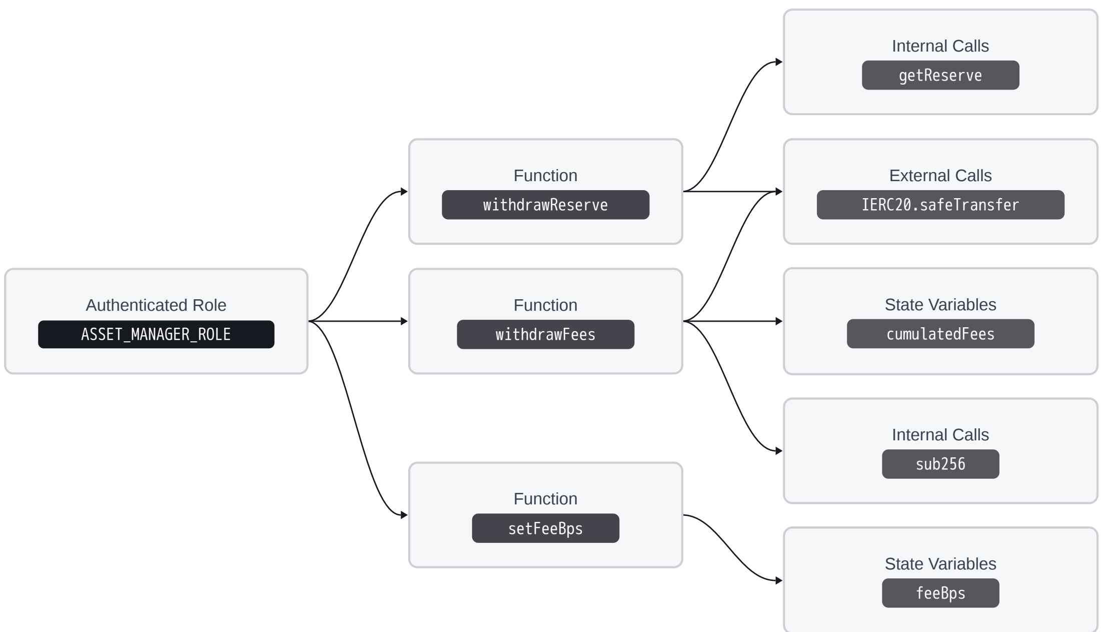
In the contract `CommodityToken` the role `VAULT_MINTER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `VAULT_MINTER_ROLE` account may allow the hacker to take advantage of this authority and execute any still-valid mint authorizations signed by `MINT_APPROVER_ROLE`, thereby consuming approved mint nonces and controlling the timing of authorized issuance.



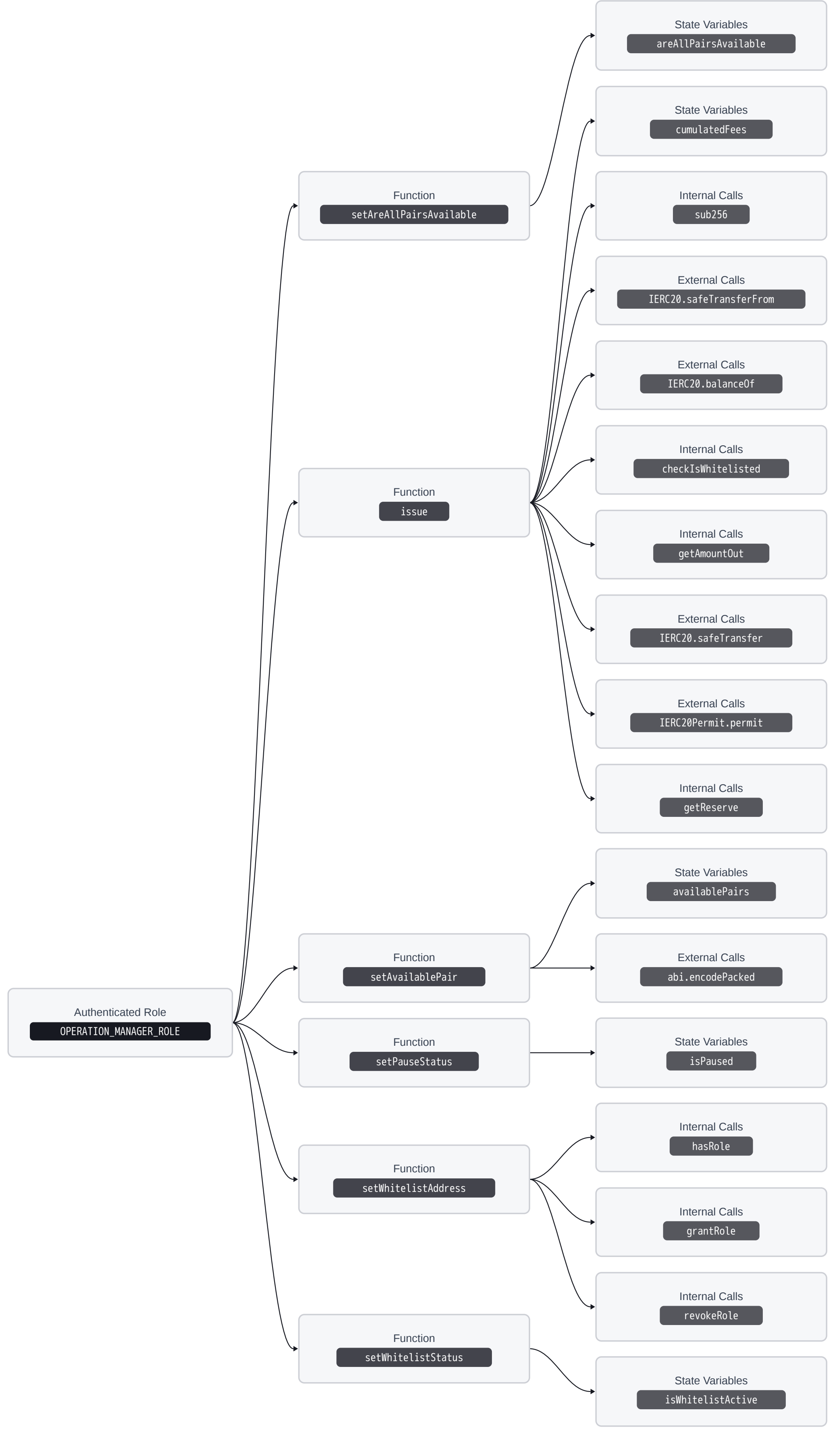
In the contract `CommodityToken`, the role `DEFAULT_ADMIN_ROLE` has authority over the role administration system inherited from `AccessControl`. Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow a hacker to grant or revoke privileged roles, including roles responsible for minting, pausing, risk management, and upgrades. As a result, compromise of this role can lead to full administrative takeover of the contract, including unauthorized minting, censorship, balance wiping through delegated risk roles, and malicious upgrade enablement through reassignment of upgrade-related roles.

CommodityTokenIssuer

In the contract `CommodityTokenIssuer` the role `ASSET_MANAGER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `ASSET_MANAGER_ROLE` account may allow the hacker to take advantage of this authority and withdraw reserve assets and accumulated fees to arbitrary addresses, and set the issuance fee up to 100%.



In the contract `CommodityTokenIssuer` the role `OPERATION_MANAGER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `OPERATION_MANAGER_ROLE` account may allow the hacker to take advantage of this authority and pause or unpause issuance, enable arbitrary token pairs or globally open all pairs, disable the whitelist or whitelist arbitrary addresses, and execute the `issue()` flow itself. Because `issue()` also relies on operator-supplied pricing parameters, compromise of this role may allow the attacker to perform unauthorized or mispriced issuances and potentially extract reserve assets from the issuer.

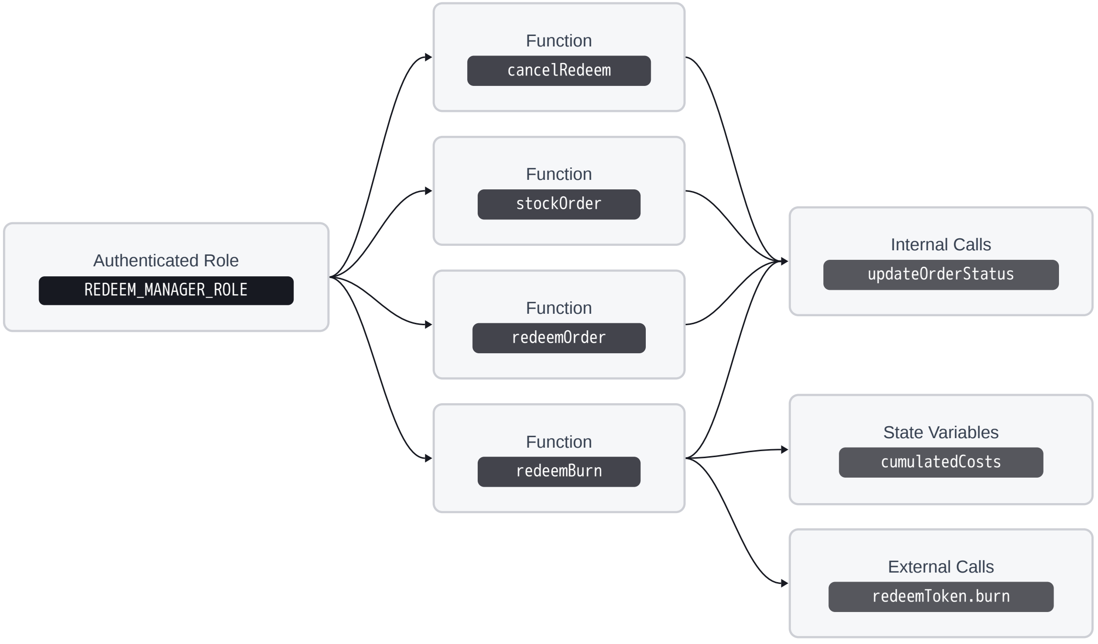

Authenticated Role
OPERATION_MANAGER_ROLE
Function
setAreAllPairsAvailable
Function
issue
Function
setAvailablePair
Function
setPauseStatus
Function
setWhitelistAddress
Function
setWhitelistStatus
State Variables
areAllPairsAvailable
State Variables
cumulatedFees
Internal Calls
sub256
External Calls
IERC20.safeTransferFrom
External Calls
IERC20.balanceOf
Internal Calls
checkIsWhitelisted
Internal Calls
getAmountOut
External Calls
IERC20.safeTransfer
External Calls
IERC20Permit.permit
Internal Calls
getReserve
State Variables
availablePairs
External Calls
abi.encodePacked
State Variables
isPaused
Internal Calls
hasRole
Internal Calls
grantRole
Internal Calls
revokeRole
State Variables
isWhitelistActive

In the contract `CommodityTokenIssuer` , the role `DEFAULT_ADMIN_ROLE` has authority over the role administration system inherited from `AccessControl` . Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow a hacker to grant or revoke privileged roles, including `ASSET_MANAGER_ROLE` and `OPERATION_MANAGER_ROLE` . As a result, compromise of this role can lead to full administrative takeover of the issuer, including reserve and fee withdrawal, fee manipulation, whitelist and pair-management control, pausing of issuance, and abuse of the issuance flow itself.

RedeemLock

In the contract `RedeemLock` the role `ASSET_MANAGER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `ASSET_MANAGER_ROLE` account may allow the hacker to take advantage of this authority and withdraw the accumulated extra costs collected by the contract to arbitrary addresses.

In the contract `RedeemLock` , the role `REDEEM_MANAGER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `REDEEM_MANAGER_ROLE` account may allow the hacker to take advantage of this authority and arbitrarily advance, revert, or finalize redemption orders by calling `stockOrder()` , `redeemOrder()` , `redeemBurn()` , and `cancelRedeem()` .



In the contract `RedeemLock` , the role `DEFAULT_ADMIN_ROLE` has authority over the role administration system inherited from `AccessControl` . Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow a hacker to grant or revoke privileged roles, including `ASSET_MANAGER_ROLE` and `REDEEM_MANAGER_ROLE` . As a result, compromise of this role can lead to full

administrative takeover of the redemption workflow, including misappropriation of accumulated costs and manipulation of redemption order processing.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[Korea Gold Exchange Digital Asset , 04/02/2026]: The current operational policy for role management is as follows:

- The **DEFAULT_ADMIN_ROLE** for all contracts is managed via a multisignature wallet.

The following roles are managed via multisig:

- **UPGRADER_ROLE** (CommodityToken)
- **PAUSER_ROLE** (CommodityToken)
- **RISK_MANAGER_ROLE** (CommodityToken)
- **ASSET_MANAGER_ROLE** (both CommodityTokenIssuer and RedeemLock use multisig)

The following role is under consideration and not yet finalized (multisig vs EOA):

- **UPGRADE_AUDITOR_ROLE** (CommodityToken)

The following role is managed using AWS KMS:

- **VAULT_MINTER_ROLE** (CommodityToken)
- **OPERATION_MANAGER_ROLE** (CommodityTokenIssuer)
- **REDEEM_MANAGER_ROLE** (RedeemLock)

The following roles are used for bridging operations:

- **MINTER_ROLE** (CommodityToken)
- **BURNER_ROLE** (CommodityToken)

The following role is currently managed by an EOA:

- **MINT_APPROVER_ROLE** (CommodityToken)

KGE-28 | `setOrderStatus()` Enables Arbitrary Lifecycle Rewind

Category	Severity	Location	Status
Design Issue, Centralization	● **Centralization**	**contracts/RedeemLock/RedeemLock.sol (update_20260408): 223~224**	● **Acknowledged**

Description

`setOrderStatus()` allows `REDEEM_MANAGER_ROLE` to step any non-terminal order back to its immediately preceding status. While `Burned` and `Cancelled` are protected, the check permits `Redeemed → Stocked` and `Stocked → Pending` independently. Applied sequentially, these two calls walk any `Redeemed` order back to `Pending` with no compensating side effects. This exposes several distinct issues:

1. A user can call `cancelRedeemRequest()` on the rewound order and receive a full refund from the shared pool, even if physical gold has already been dispatched.
2. `REDEEM_MANAGER_ROLE` is the operational role that processes redemptions — granting it simultaneous power to undo them creates a direct conflict of interest with no separation of duties. This power should at minimum be restricted to `DEFAULT_ADMIN_ROLE`.
3. Because `cancelRedeemRequest()` does not zero `goldWeight` and `extraCost`, any future relaxation of the reversion guard risks a double-refund from the same order.
4. The natspec is inconsistent with the implementation on three counts: (a) it describes the caller as *"the admin"* whereas the guard enforces `REDEEM_MANAGER_ROLE` (b) it implies the function only applies to `Redeemed` orders whereas `Stocked` orders are equally affected (c) and it describes the operation as a *"cancel"* whereas no cancellation side effects (refund, field zeroing) are executed.

Recommendation

We recommend that the client apply the following steps:

1. Remove the `Stocked → Pending` reversion path entirely.
2. Gate `Redeemed → Stocked` behind `DEFAULT_ADMIN_ROLE`, not `REDEEM_MANAGER_ROLE`, and emit a dedicated event with an explicit justification parameter.
3. Zero `goldWeight` and `extraCost` in `cancelRedeemRequest`.
4. Correct the natspec to accurately reflect the caller role, the affected states, and the absence of cancellation semantics.

Alleviation

[CertiK, 04/09/2026]: The removal of `setOrderStatus()` and the introduction of `cancelRedeemedOrder()` in commit hash 6facb6e51fbfc0666e8405031f0c24d387d3fa21 address the primary concerns. The two-step rewind, the double-refund vector, and the separation-of-duties issue are **resolved**.

A **residual centralization risk** remains: `DEFAULT_ADMIN_ROLE` can repeatedly invoke `cancelRedeemedOrder` to rewind any `Redeemed` order back to `Stocked`, indefinitely preventing it from reaching `Burned`. Please refer to **KGE-06** for mitigation advice on privileged role management.

[CertiK, 04/27/2026]: The client has replaced `DEFAULT_ADMIN_ROLE` privileges on `setOrderStatus()` with `ASSET_MANAGER_ROLE` in commit hash 4f23f4d5f83c08e9302b9c6277e69887abeb94c4. However, this does not dissipate the centralization concern, as the ability to revert `Redeemed → Stocked` remains in the hands of a privileged role, and can still be used to indefinitely delay finalization of orders.

[Korea Gold Exchange Digital Asset , 04/30/2026]: Issue acknowledged. I won't make any changes for the current version.

Due to the nature of the redeem process, it is not feasible to eliminate the centralized components. Accordingly, KGE-28 is acknowledged and will be closed in an Acknowledged state.

KGE-29 | Manager Controls Token Split With No User Commitment

Category	Severity	Location	Status
Design Issue, Centralization	**Centralization**	**contracts/RedeemLock/RedeemLock.sol (update_20260408): 69~70**	**Acknowledged**

Description

The EIP-2612 permit signature authorizes a single total amount `_goldWeight + _extraCost` , but the split between the two values is chosen freely by `REDEEM_MANAGER_ROLE` with no on-chain constraint. `_goldWeight` is later burned and `_extraCost` is credited to `cumulatedCosts` , withdrawable by `ASSET_MANAGER_ROLE` via `withdrawCosts` . Because the user never signs over the individual components, a malicious or compromised manager can set `_goldWeight = 0, _extraCost = total` and route the entirety of the user's escrowed funds to protocol revenue without any on-chain proof of user consent.

Recommendation

We recommend that the client replace the current permit flow with a dedicated EIP-712 authorization struct signed by the user over the exact `(goldWeight, extraCost)` split.

Alleviation

[Korea Gold, 04/08/2026]: We acknowledge that the current design does not enforce the split between goldWeight and extraCost on-chain. This is intentional, as the redeem flow is designed to be operationally managed by the server. If the amount corresponding to goldWeight is not properly burned, a discrepancy would arise between the Proof of Reserve and the token supply, which would directly create an operational loss for the company. Accordingly, the REDEEM_MANAGER_ROLE is considered highly sensitive and will be subject to strict control.

KGE-07 | Zero-Address `MINT_APPROVER_ROLE` Enables Signature Bypass In `mintWithAuthorization()`

Category	Severity	Location	Status
Access Control	Medium	contracts/CommodityToken/CommodityToken.sol (base): 314~315	Resolved

Description

Notwithstanding the general signature malleability issue, `mintWithAuthorization()` relies on raw `ecrecover` and only checks that the recovered `signer` holds `MINT_APPROVER_ROLE`.

However, `ecrecover` returns `address(0)` for invalid or malformed signatures. If `address(0)` is granted `MINT_APPROVER_ROLE`, this check can be bypassed entirely, allowing minting with arbitrary (invalid) signatures.

This creates a misconfiguration-dependent authorization bypass that can lead to unauthorized minting.

Recommendation

We recommend that the client:

1. Use `ECDSA.recover` instead of raw `ecrecover`.
2. Explicitly reject zero-address signers as an extra-safeguard, e.g.:

```
if (signer == address(0) || !hasRole(MINT_APPROVER_ROLE, signer)) {
    revert InvalidSignature();
}
```

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by applying the two recommended changes in commit 87e17c3c5c4802db235a926e0a2395794168c141.

KGE-08 | Mandatory Inline `permit()` In `issue()` Is Griefable Via Front-Running

Category	Severity	Location	Status
Denial of Service, Coding Issue	Medium	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (base): 324~332	Resolved

Description

In `issue()`, the contract unconditionally calls `IERC20Permit(_taIn).permit()` before `safeTransferFrom()` at `CommodityTokenIssuer.sol`.

A third party can front-run the `permit()`, consume the nonce, and set the allowance before `issue()` executes. When the original `issue()` call is later processed, it reverts on the `permit` call even though sufficient allowance already exists.

As a result, issuance is unnecessarily dependent on successful inline `permit` execution, which enables griefing/DoS of the flow and forces users to produce a new signature.

Recommendation

We recommend not making `permit` a mandatory step inside `issue()`, i.e. check the current allowance first and only call `permit` when allowance is insufficient.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by making the use of a permit non-mandatory in commit ec34cc415e73b6a9c8b3249b5f5e33eb1ecd8f9b.

KGE-09 | `lockTime` Recorded But Not Enforced

Category	Severity	Location	Status
Volatile Code, Design Issue	● Medium	contracts/RedeemLock/RedeemLock.sol (base): 84~85	● Resolved

Description

The struct `RedeemLock` stores a per-order `lockTime` (set to `block.timestamp` at order creation) and includes it in the `orderId`, but it is never used to enforce any time-based behavior (no minimum lock, no expiry, no timeout-based cancellation, no SLA checks). As a result, the name `lockTime` can mislead integrators/auditors into assuming time-locked semantics that don't exist, and it adds storage/event surface without functional effect.

Recommendation

We recommend that the client:

1. Implement explicit time-based logic that uses `lockTime` (e.g., order expiry, timeout-based user cancellation if stuck in `Pending/Stocked`, or enforce a minimum delay before `redeemBurn()`).
2. Document the intended behavior.

Otherwise, please remove `lockTime` from `OrderData` (and from the event / `orderId` derivation) if its role is only informational, to save gas and improve code readability.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by removing `lockTime` in commit 67b0f453d5ee6814bc53f35cbae5f9c2312bd582.

KGE-23 `cancelAuthorization()` Is Replayable And Emits Ambiguous `AuthorizationUsed` Events

Category	Severity	Location	Status
Volatile Code	Medium	contracts/CommodityToken/CommodityToken.sol (base): 223~224	Resolved

Description

`cancelAuthorization()` does not check whether `usedNonces[authorizer][nonce]` is already `true`, so the same signed cancellation message can be replayed indefinitely and will keep emitting `AuthorizationUsed(authorizer, nonce)` even though the authorization state does not change after the first call. On-chain, this does not revert/undo any previously executed mint/transfer (it is effectively idempotent), but the replayable and semantically ambiguous event ("used" vs "canceled") can confuse off-chain monitoring/workflows—especially if systems interpret `AuthorizationUsed` as a business-critical signal.

Recommendation

We recommend that the client:

1. Add a guard in `cancelAuthorization()` to revert (or no-op without emitting) if `usedNonces[authorizer][nonce]` is already `true`, preventing replay log spam.
2. Emit a dedicated `AuthorizationCanceled(authorizer, nonce)` event (instead of reusing `AuthorizationUsed`) so cancellation cannot be confused with "authorization consumed via transfer/mint".

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by applying the recommended changes in commit 489b4a21a98ae3fde66e932a7f45b7893bdfe2e2.

KGE-30 | Frozen Accounts Cannot Revoke Allowances; Stale Allowances Survive Wipe

Category	Severity	Location	Status
Volatile Code	Medium	contracts/CommodityToken/CommodityToken.sol (update_20260408): 365~366	Resolved

Description

`_approve()` unconditionally reverts when either the owner or spender is frozen, with no carveout for zero-value revocations:

```
if (isFrozen(_owner)) revert AccountFrozen(_owner);
if (isFrozen(_spender)) revert AccountFrozen(_spender);
```

This produces two related issues:

1. **Revocation blocked during freeze.** A token owner cannot call `approve(spender, 0)` while the spender is frozen — precisely the moment they are most likely to want to. The contract already applies this pattern correctly for pause (`if (_amount > 0) { _requireNotPaused(); }`), but the frozen check has no equivalent carveout.
2. **Stale allowances survive `wipeFrozenAccount`.** `wipeFrozenAccount` burns the frozen account's balance but does not clear its outgoing allowances. If the wiped address is later re-credited (e.g. restitution or re-onboarding), previously approved spenders immediately regain access up to the old allowance amounts with no intervention required.

Recommendation

We recommend that the client:

1. Allow zero-value approvals regardless of frozen status: add `if (_amount > 0)` as a guard around the frozen checks in `_approve`, mirroring the existing pause logic.
2. Clear all outgoing allowances of the wiped account in `wipeFrozenAccount`, or at minimum document explicitly that re-crediting a wiped address requires a manual allowance audit.

Alleviation

[Korea Gold Exchange Digital Asset , 04/09/2026]: updated to frozen account can revoke allowances

https://github.com/CrederLabs/KGLD_contract/pull/1/changes/1650702163edadcc8862f767f11dc123b29f93b9

We acknowledge that allowances associated with a wiped account are not explicitly cleared in the current implementation.

`wipeFrozenAccount` is designed as an administrative action to burn the account's balance, rather than to fully reset all approval relationships.

[CertiK, 04/09/2026]: The first recommendation is addressed in commit hash 1650702163edadcc8862f767f11dc123b29f93b9— zero-value approvals are now correctly permitted regardless of frozen status. The second recommendation remains open: `wipeFrozenAccount()` still does not clear outgoing allowances/stale approvals on a wiped address become immediately usable upon re-crediting. We reiterate our recommendation to clear allowances in `wipeFrozenAccount` or document the re-crediting risk explicitly.

[CertiK, 04/28/2026]: The wipe-path owner-side reset is correctly implemented in commit c9205923fc235edd06db5b94e85dad8b18356e11. Two residual points remain, both addressable through documentation:

1. **Spender-side gap on wipe.** If a wiped address is ever re-credited, pre-existing third-party approvals to it remain live. We do not recommend a code fix — most production stablecoins (e.g., USDC, USDT) leave the same gap. Please add an inline NatSpec note on `wipeFrozenAccount()` and a `RISK_MANAGER_ROLE` runbook entry requiring manual review of spender-side allowances before any re-credit.

2. **Unfreeze comment/code inconsistency.** The comment above `AllowanceNonceStorage` states *"When an account is unfrozen, allowances are being reset"*, but only `wipeFrozenAccount()` increments the nonce — `unfreeze` does not. We assume freeze/unfreeze is a temporary hold preserving allowances; please confirm and fix the comment. If invalidation on unfreeze is intended, a `RISK_MANAGER_ROLE`-gated `resetAllowances(address)` handle would be cleaner.

With these documentation updates in place, we are comfortable fully closing this finding.

[CertiK, 04/30/2026]: The client has applied the suggested changes, thus resolving the finding.

KGE-10 | Signature Malleability Of `ecrecover`

Category	Severity	Location	Status
Volatile Code	Minor	contracts/CommodityToken/CommodityToken.sol (base): 235, 312, 502, 561	Resolved

Description

The `ecrecover()` function is subject to signature malleability. Signature malleability is possible within the Elliptic Curve cryptographic system. An elliptic curve is symmetric on the `X` -axis, meaning two points can exist with the same `X` value. In the `r` ,'s', and `v` representations, this permits us to carefully adjust `s` to produce a second valid signature for the same `r` , thus breaking the assumption that a signature cannot be replayed in what is known as a replay-attack.

Recommendation

We suggest to consider the example in `ECDSA.sol` from the OpenZeppelin library to prevent signature malleability.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by using `ECDSA.recover()` in commit 9989150fc1f3ef2d4207bc2a28a48c7f548d39ad.

KGE-11 | Paused State Prevents Allowance Reduction/Revocation

Category	Severity	Location	Status
Design Issue	● Minor	contracts/CommodityToken/CommodityToken.sol (base): 349~350	● Resolved

Description

The contract applies `whenNotPaused` to `_approve()` , which prevents users from modifying allowances while the contract is paused.

As a result, users cannot reduce or revoke previously granted allowances during a pause. This may expose them to unintended risk if a spender remains approved while transfers are otherwise disabled.

Recommendation

We recommend that the client enable allowance decreases (or at least, full revocation) during pause. For example:

1. Allow `_approve()` when `_amount == 0`
2. Or introduce a separate function to revoke allowances while paused.

Alleviation

[Korea Gold Exchange Digital Asset , 03/31/2026]: allow calling "approve" when the allowance is 0 even if the contract is paused

---

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit d5a4f264f31893c8bd87ef6966915b8071fd34da.

KGE-12 | Zero-Output Issuances Are Not Rejected

Category	Severity	Location	Status
Volatile Code	Minor	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (base): 274~275	Resolved

Description

Function `issue()` does not enforce that the computed issuance amount is nonzero. In particular, `getAmountOut()` may return `quoteData.amtOut == 0` when `_amtIn` is too small relative to token decimals, exchange-rate scaling, fees, and/or `_retainingDecimals` truncation.

If `_amtOutMin` is set to `0`, the current checks do not prevent execution:

1. The slippage check passes because `0 < 0` is false,
2. The reserve check passes because requesting `0` output requires no reserve,
3. The contract then pulls `_taIn` from `owner`,
4. Transfers `0` `_taOut`,
5. And records the input-side fee.

As a result, the user may pay a nonzero `_amtIn` while receiving no output.

Recommendation

We recommend that the client add an explicit validation in `issue()` that rejects zero-output issuance, for example by reverting when `quoteData.amtOut == 0`. Optionally, to avoid dust transactions that are guaranteed to round down to zero, the protocol may also enforce:

1. `_amtIn > 0`,
2. `_amtOutMin > 0`,
3. Or a minimum economically meaningful trade size,

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by adding the recommended checks in commit afd8c842917751232ba254a6a717c76fe38ce610.

KGE-13 | `getAmountIn()` Uses Inefficient And Unsound Search Bounds

Category	Severity	Location	Status
Volatile Code, Design Issue, Gas Optimization	● Minor	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (base): 206~209	● Resolved

Description

View function `getAmountIn()` performs a binary search over `[0, IERC20(_taIn).totalSupply()]` to invert `getAmountOut()` .

This has two issues:

1. **Inefficient bounds** The pricing formula is quasi-linear in `_amtIn` , with only small rounding errors from `Math.mulDiv` (i.e., `floor(a·b/c)` differs from the exact value by < 1). A direct algebraic inversion provides a close approximation, making a full-range binary search unnecessary.

2. **Unsound upper bound** `totalSupply()` is unrelated to the required `_amtIn` . In some configurations, the required input may exceed this bound, causing the search to **saturate early and underestimate** **`_amtIn`** .

As a result, users relying on this function to prepare permits may submit transactions with insufficient `_amtIn` , leading to unexpected reverts in `issue()` (via slippage checks).

Recommendation

We recommend that:

1. The client start the global binary search with lower and upper bounds based on direct approximate inversions of the pricing formula.
2. If used for quoting only and compatible with design of the client, consider returning a safe over-approximation of `_amtIn` without dichotomy. Update the natspec accordingly.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit 83a177ca12de3eada4875cc4d064d5c363b54c16. Please consider documenting more explicitly that `getAmountIn()` may return a slight overestimate of the required input under normal conditions.

KGE-14 | Missing Input Validations

Category	Severity	Location	Status
Volatile Code	Minor	contracts/CommodityToken/CommodityTokenProxy.sol (base): 20~21	Resolved

Description

The proxy constructor in `CommodityTokenProxy` only checks that `_logic != address(0)` before storing it as the implementation.

As a result, the constructor accepts any non-zero address, including EOAs or addresses with no code. If such an address is provided and `_data` is empty, deployment succeeds, and the proxy is initialized with a non-contract implementation. Subsequent fallback calls delegate to an address with no code, making the proxy unusable.

A similar issue exists for the `DEFAULT_ADMIN_ROLE` of contract `CommodityToken`.

Recommendation

We recommend checking the `_logic` code length before storing it as the implementation:

```
    require(_logic.code.length > 0, "UUPSProxy: logic has no code");
```

Likewise, please check any privileged address, including `DEFAULT_ADMIN_ROLE`, against the zero address, unless such a role can be disabled.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by adding a code length check in commit 150242faf52bd3574d0ddce239e47d11aaaffd82.

KGE-15 | Freeze Status Can Be Changed Without Emitting `Frozen` / `Unfrozen` Events

Category	Severity	Location	Status
Volatile Code	Minor	contracts/CommodityToken/CommodityToken.sol (base): 115~116, 120~121	Resolved

Description

The contract exposes dedicated `freeze()` and `unfreeze()` functions that emit `Frozen` and `Unfrozen` events. This suggests that these events are intended to serve as the canonical monitoring surface for account freeze status.

However, freeze state is ultimately implemented through `FROZEN_ROLE`. Since `FROZEN_ROLE` is administered through `AccessControl`, an authorized caller can directly invoke `grantRole(FROZEN_ROLE, account)` or `revokeRole(FROZEN_ROLE, account)` through the proxy, bypassing the dedicated wrapper functions. In that case, the role state changes successfully, but only the generic `RoleGranted` / `RoleRevoked` events are emitted, while `Frozen` / `Unfrozen` are not.

As a result, off-chain monitoring systems that rely exclusively on `Frozen` / `Unfrozen` events may miss actual freeze-state changes and maintain an incomplete or inconsistent view of which accounts are frozen.

Recommendation

We recommend ensuring that either freeze-state changes cannot bypass the canonical event flow, or enforcing all freeze-state changes are made through the `AccessControl` library functions.

Alleviation

[Korea Gold Exchange Digital Asset , 04/08/2026]:

- remove frozen/unfrozen events
- trace frozen/unfrozen by RoleGranted and roleRevoked

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit 452fc0b9320e59be781b59fb32214b65e441352a.

KGE-31 Premature Decimal Normalization In `getAmountOut` Causes Systematic Output Underestimation

Category	Severity	Location	Status
Incorrect Calculation	Minor	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (update_20260408): 268~269	Resolved

Description

`getAmountOut()` applies decimal normalization before the exchange-rate ratio:

```
uint256 rawAmtOut = Math.mulDiv(
    Math.mulDiv(amtInAfterFee, 10 ** dOut, 10 ** dIn),
    _exRateIn,
    _exRateOut
);
```

When `dOut < dIn`, the inner `mulDiv` floors `amtInAfterFee * 10**dOut / 10**dIn` before multiplying by `_exRateIn / _exRateOut`, introducing a systematic downward bias. The correct formulation keeps all numerator terms together before the final division:

```
uint256 rawAmtOut = Math.mulDiv(
    Math.mulDiv(amtInAfterFee, 10 ** dOut * _exRateIn, 10 ** dIn),
    1,
    _exRateOut
);
```

Since `issue()` uses `quoteData.amtOut` directly for settlement, users on low-decimal output pairs (e.g. 18-in / 6-out) may be systematically underpaid or encounter spurious `SlippageExceeded` reverts on edge-case input amounts.

Recommendation

We recommend that the client reorder the computation to accumulate numerator terms before dividing, eliminating the intermediate floor.

Alleviation

[CertiK, 04/09/2026]: The client has applied the suggested changes in commit hash 277e8523cfd8403f16483e5bdbda50dce94a77a0, thus resolving the finding.

KGE-33 `getAmountIn()` Has Unenforced Input And Configuration Preconditions Causing Either Gas Exhaustion Or Division-By-Zero

Category	Severity	Location	Status
Volatile Code	Minor	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (update_20260409): 198~199	Resolved

Description

`getAmountIn()` is a `view` quoting helper for off-chain UX (never called inside `issue()`). The function relies on a closed-form ceiling inverse of the forward formula

$$\mathtt{amtOut} = \left\lfloor \frac{\left\lfloor \mathtt{amtIn} \cdot \frac{D-f}{D} \right\rfloor \cdot 10^{d_{\text{out}}} \cdot r_{\text{in}}}{10^{d_{\text{in}}} \cdot r_{\text{out}}} \right\rfloor_{\text{step}}$$

(where $\lfloor \cdot \rfloor_{\text{step}}$ floors to multiples of $\text{step} = 10^{d_{\text{out}} - r_{\text{ret}}}$ with $r_{\text{ret}} =$ `_retainingDecimals`), followed by a correction loop:

```
while (q.amtOut < _amtOut) {
    approxAmtIn += 1;
    q = getAmountOut(...);
}
```

Two defects, both stemming from unenforced preconditions:

1. Loop is dead code on aligned inputs, unbounded on non-aligned ones. Because `getAmountOut` only produces multiples of step, the ceiling inverse already guarantees `q.amtOut >= _amtOut` on the first evaluation when `_amtOut` is a multiple of step — so the loop never executes under the docstring's assumption ("*_amtOut is already the truncated output amount*"). When the assumption is violated, the requested output simply does not exist in the codomain: the loop walks `approxAmtIn` one unit at a time looking for a non-existent crossing, requiring $\sim 10^{10}$ iterations in production decimals. Guaranteed out-of-gas.

2. 100% fee divides by zero. `setFeeBps` accepts `_newFeeBps <= D`, including the boundary value `10000`. The inverse computes `feeDenom = D - feeBps`, which becomes zero, and the subsequent division reverts. An admin-misconfiguration footgun, sharing root cause with the first point: implicit math preconditions (aligned `_amtOut`, $f < D$) that the code never validates.

Recommendation

We recommend that the client apply the following steps:

1. In `setFeeBps()`, tighten the bound to strict inequality.
2. In `getAmountIn()`, validate alignment and remove the loop:

```
uint256 step = 10 ** sub256(dOut, _retainingDecimals);
if (_amtOut % step != 0) revert AmountOutNotAligned(_amtOut, step);
// ... existing ceiling inverse ...
require(q.amtOut >= _amtOut);
return q;
```

3. Document in NatSpec: `_amtOut` must be aligned, the function returns a safe over-approximation, `feeBps` must be strictly less than `BIAS_POINT_DENOMINATOR`.

Alleviation

[CertiK, 04/30/2026]: Point 1 (unbounded correction loop on non-aligned inputs) is resolved by the alignment guard `if (_amtOut % step != 0) revert InvalidAmountOut();` and the post-computation sufficiency check, which together make the function terminate in O(1) on valid inputs and revert immediately on invalid ones. Point 2 (100% fee → division by zero) is fully resolved as suggested. We thus mark the finding as **resolved**.

Note: Please consider tightening the NatSpec on `getAmountIn()` to make its contract unambiguous: (a) state explicitly that `_amtOut` must be a multiple of `10 ** (dOut - retainingDecimals)`, (b) state that the returned `amtIn` is *sufficient* but *not guaranteed minimal* — exotic decimal/rate configurations may produce a small overshoot, and (c) clarify the existing comment "*`_amtOut` should apply retainingDecimals truncation and fee deduction*", which can be read ambiguously.

[CertiK, 05/21/2026]: Subsequent to the original alleviation, the team migrated the fee accounting in `getAmountIn()` and `getAmountOut()` from an inclusive-on-gross model to an additive-on-net model (`amtIn = net × (D + feeBps) / D`) in commit hash d5bd05976a550052fb2fd2897788c694eafefc0e. We have reviewed the new implementation and confirm that the two preconditions originally raised in this finding remain satisfied — the alignment guard is preserved and the new divisor `(D + feeBps)` structurally eliminates the divide-by-zero path — so the finding stays **Resolved**.

KGE-03 | User-Provided `extraCost` Is Unverified

Category	Severity	Location	Status
Design Issue	● Informational	contracts/RedeemLock/RedeemLock.sol (base): 70	● Partially Resolved

Description

`redeemLock(uint256 _goldWeight, uint256 _extraCost)` lets the user freely choose `_extraCost`, and the contract does not validate it against any on-chain fee schedule, quote, bounds, or signature. The contract simply escrows `_goldWeight + _extraCost` and later, if the order reaches `redeemBurn()`, treats `extraCost` as protocol "costs" by adding it to `cumulatedCosts`. This makes correct fee payment dependent on off-chain policy (e.g., the redeem manager refusing to process orders with insufficient `extraCost`) rather than an on-chain invariant.

Recommendation

We kindly request that the client clarify their design and answer the following questions:

1. Is `_extraCost` supposed to come from an off-chain quote (shipping/handling/tax) that the user accepts?
2. What is the intended behavior if `_extraCost` is too low/too high—should such orders be rejectable on-chain, or only operationally?
3. Should `_extraCost` be enforced via an **admin-configurable on-chain formula** or an **authorized signed quote**, instead of trusting user input?

Alleviation

[CertiK, 04/08/2026]: The team introduced changes in commit 79649e5c4c2bb603439313788bdb9ff02184ba9e, adopting an EIP-2612 `permit` flow so that gold price calculations are handled entirely off-chain by the server, and gating `redeemLock` behind `REDEEM_MANAGER_ROLE`. While this addresses the original concern (the user can no longer freely set `extraCost` without manager involvement), the new implementation introduces several issues that, taken together, shift the trust problem from the user to the manager in a worse form:

1. **Permit only binds the total amount, not the order split.** The user signs an EIP-2612 approval for `_goldWeight + _extraCost`, but the manager freely chooses how to split that total between `_goldWeight` (later burned) and `_extraCost` (later credited to `cumulatedCosts` and withdrawable via `withdrawCosts`). A malicious or compromised manager can set `_goldWeight = 0, _extraCost = total` and route 100% of the user's escrowed tokens to protocol revenue, with no on-chain proof that the user agreed to that split. The original finding asked the contract to enforce fee correctness on-chain; the new version provides *no* on-chain guarantee about how user funds are partitioned.

2. **New `setOrderStatus` removes lifecycle guarantees.** The previous version exposed only `cancelRedeem` (`Redeemed → Stocked`). The new `setOrderStatus` lets `REDEEM_MANAGER_ROLE` move any order to any status with no transition validation and without running the side effects of the canonical paths (`cancelRedeemRequest`'s refund, `redeemBurn`'s burn and `cumulatedCosts` update). Combined with the unverified split above, this gives the manager arbitrary control

over both the *amount* attributed to costs and the *lifecycle state* of any order — including rewinding terminal states to replay refunds or burns against the shared `redeemToken` pool.

3. **Stale order fields after cancellation enable replay under rewind.** `cancelRedeemRequest` marks an order `Cancelled` but does not zero `orders[orderId].goldWeight` or `orders[orderId].extraCost`. If the manager subsequently uses `setOrderStatus` to move the order back to `Pending`, the original values are still present and a second `cancelRedeemRequest` will refund the user again from the shared pool. The same applies to `Burned → Redeemed` rewinds enabling repeated `redeemBurn` calls.

4. **Permit front-running griefing.** `redeemLock` calls `IERC20Permit(...).permit(...)` without try/catch when the current allowance is insufficient. Because `(v, r, s)` is visible in the mempool, any observer can submit `permit()` directly to the token, consuming the EIP-2612 nonce and causing the manager's `redeemLock` to revert on the inner `permit`. Standard mitigation is to wrap `permit` in try/catch and rely on the subsequent `transferFrom` for correctness.

5. **`orderId` stability regression.** The previous `orderId` hash included `block.timestamp` (`lockTime`), which has been removed. This weakens uniqueness guarantees and, combined with the permissive `setOrderStatus` and stale-field issues above, makes replay-style state corruption easier to construct.

6. **Loss of user-initiated lock (centralization regression).** Previously the user called `redeemLock` directly (`msg.sender` was the depositor). The new version restricts `redeemLock()` to `REDEEM_MANAGER_ROLE`, so users have no on-chain path to lock their own tokens without manager cooperation. If the manager is offline or censoring, users cannot initiate redemptions at all. Worth documenting explicitly whether this is intentional.

7. **Enum rename `Canceled → Cancelled`.** Minor, but any off-chain indexer, subgraph, or client previously decoding the `Canceled` variant will break silently on upgrade. Worth coordinating with downstream consumers.

[Korea Gold, 04/08/2026]:

1. We acknowledge that the current design does not enforce the split between goldWeight and extraCost on-chain. This is intentional, as the redeem flow is designed to be operationally managed by the server. If the amount corresponding to goldWeight is not properly burned, a discrepancy would arise between the Proof of Reserve and the token supply, which would directly create an operational loss for the company. Accordingly, the REDEEM_MANAGER_ROLE is considered highly sensitive and will be subject to strict control.

2. We agree that introducing `setOrderStatus` adds operational flexibility but may weaken lifecycle safety if left unrestricted. To mitigate this risk, we plan to restrict the function as follows:

No status transition will be allowed once an order is in the Burned state.

No status transition will be allowed once an order is in the Cancelled state.

Status changes through `setOrderStatus()` will only be allowed to the immediately preceding state.

3. To mitigate permit front-running and griefing/DoS risk, we plan to wrap the permit call in try/catch.

4. To strengthen orderId uniqueness, we plan to include block.timestamp in the orderId generation logic.

5. The remaining items will be clarified through documentation, and the related operational risks will be managed through internal controls.

[CertiK, 04/08/2026] The client has resolved some issues, acknowledge some other but some are still pending, in that, we need the written intent of the client to close the discussion. Please answer to Point 6 and remediate point 3 below.

1. Permit only binds the total amount, not the order split — Pending, differed to another finding. We have opened a separate finding to address this issue.

2. `setOrderStatus()` lifecycle guarantees — Pending, differed to another finding We have opened a separate finding to address this issue.

3. Stale order fields after cancellation — Partially mitigated. Blocking `Cancelled` as a source state in `setOrderStatus()` closes the most direct replay vector described in our prior comment. However, `goldWeight` and `extraCost` are still not zeroed on cancellation. Combined with the two-step rewind path (`Redeemed → Stocked → Pending`), an order that has reached `Redeemed` can still be walked back to `Pending` and cancelled for a full refund against the shared pool. We recommend zeroing order fields on both `cancelRedeemRequest` and any sanctioned reversion path.

4. Permit front-running griefing — Resolved. `redeemLock` now wraps the `permit` call in a `try/catch` block and falls through to `transferFrom` regardless of outcome. This correctly handles the case where the permit nonce has been consumed by a front-runner.

5. `orderId` uniqueness — Resolved. `block.timestamp` has been reintroduced alongside the per-owner nonce.

6. Loss of user-initiated lock — Not addressed. We kindly request that the team explicitly confirm whether this centralization is intentional and whether a permissionless fallback is planned and in that case, implement it.

[CertiK, 04/09/2026]: Of the remaining open questions 3 and 6:

- **3. Stale order fields after cancellation. Resolved.** The appropriate fields are now zeroed
- **6. Loss of user-initiated lock.** In absence of an explicit answer of the client or an update in the concerned function, we deduce that centralization of the redemption path is intentional. Please refer to Finding **KGE 29** for the general redemption-related centralization issues.

KGE-04 | `RedeemLock` Order Lifecycle Has No On-Chain Fulfillment Enforcement

Category	Severity	Location	Status
Volatile Code, Design Issue	● Informational	contracts/RedeemLock/RedeemLock.sol (base): 95~96	● Acknowledged

Description

The `RedeemLock` flow is effectively an escrow + status tracker, where all meaningful lifecycle progression is controlled by privileged roles (`REDEEM_MANAGER_ROLE` / `ASSET_MANAGER_ROLE`). The contract does not enforce any on-chain guarantees around order fulfillment (e.g., deadlines, required sequencing beyond basic status checks, automatic expiry, or user-triggered escape hatches once the order leaves `Pending`). Concretely:

- A user can only recover funds by canceling while `Pending`; once an order is moved to `Stocked` (by `REDEEM_MANAGER_ROLE`), the user has no on-chain mechanism to force completion, force refund, or time out the order.
- As noted in another finding, although `lockTime` is recorded, it is not used to enforce any SLA/timeout behavior.

This design makes user protection and "order correctness" primarily an operational/trust assumption rather than a protocol-enforced property.

Recommendation

We kindly request that the client clarify their design and answer the following questions:

1. Is this lack of on-chain enforcement intentional (i.e., the system is explicitly custodial/ops-driven)?
2. Is chronological, user-level ordering of redemptions an intended design requirement? If so, is this ordering expected to be enforced on-chain or handled off-chain?
3. More generally, what are the expected operational assumptions and controls behind `REDEEM_MANAGER_ROLE`?
4. What happens operationally if an order is moved to `Stocked` but is never redeemed/burned? Do users have any contractual/operational recourse, and should the contract include an on-chain timeout-based cancellation/refund path?

Alleviation

[Korea Gold Exchange Digital Asset , 03/31/2026]:

- 1,2.
  - The overall redeem process is primarily handled off-chain. Therefore, the absence of on-chain enforcement is intentional by design.

- 3.
  - The REDEEM_MANAGER_ROLE is assigned to an account managed by our centralized server, which is secured using a KMS-based key management system.
  - Authorized merchants (or agents) responsible for physical delivery submit API requests to update each stage of the redeem process. The server validates these requests and updates the order status accordingly.
- 4.
  - By design, once an order reaches the Stocked state, it is considered irreversible from both the user and server perspectives. Therefore, no additional control mechanisms (such as cancellation or rollback) were implemented for this stage.
- 4-1.
  - However, to introduce more operational flexibility, we plan to remove the `cancelRedeem` function and instead implement a function that allows controlled updates to the orderStatus.

[CertiK, 04/07/2026]: The team introduced changes in commit cfc1ea708f86b71d93ab6445dbf7ef761458dc9e adopting an EIP-2612 `permit` flow so that `_goldWeight` and `_extraCost` are determined off-chain by a KMS-secured server (`REDEEM_MANAGER_ROLE`) rather than supplied directly by the user. The client's stated rationale is that fulfillment is intentionally off-chain, and the server is the source of truth for pricing.

While this removes the *user-supplied* `_extraCost` from the equation, it does not actually establish an on-chain invariant tying the escrowed amounts to the user's intent — it simply moves the unverified value from the user to the manager. Specifically, the EIP-2612 permit signed by the user only commits to a *total* allowance of `_goldWeight + _extraCost` ; it does not commit to the *split* between the two fields. The manager is free to call `redeemLock` with any `(_goldWeight, _extraCost)` decomposition that sums to the approved total, including `_goldWeight = 0, _extraCost = total` , in which case the entire user deposit is later credited to `cumulatedCosts` and becomes withdrawable via `withdrawCosts` . The KMS protection on the server key guards against key extraction but not against a buggy server, a malicious insider with API access, or a compromised merchant submitting bad API requests; in all of those cases, the on-chain contract provides no backstop and no proof that the recorded split matches what the user agreed to. The original finding asked the contract to enforce fee correctness on-chain, and the new implementation still does not.

[CertiK, 04/08/2026]: Although the client has confirmated that the overall redemption flow is intentionally custodial and off-chain driven, the concern that user funds are not bound on-chain to the split between `goldWeight` and `extraCost` remains open regardless of KMS protection, which guards against key extraction but not against a buggy server, a malicious insider, or a compromised merchant. Please refer to Finding KGE-28 as a follow-up.

KGE-16 | Missing Emit Events

Category	Severity	Location	Status
Coding Style	Informational	contracts/CommodityToken/CommodityToken.sol (base): 443	Resolved

Description

There should always be events emitted in sensitive functions that are controlled by centralization roles.

Recommendation

It is recommended to emit events in sensitive functions that are controlled by centralization roles.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit cad1020bea7f84f3767947f87d21ddee36017046.

KGE-17 | Hardcoded Role Hash Reduces Readability And Maintainability

Category	Severity	Location	Status
Coding Issue	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 29~30	● Resolved

Description

The contract defines the role identifier as a hardcoded hash. For instance:

```
// keccak256("UPGRADER_ROLE");
bytes32 public constant UPGRADER_ROLE =
    0x189ab7a9244df0848122154315af71fe140f3db0fe014031783b0946b8c9d2e3;
```

In Solidity, expressions such as `keccak256("UPGRADER_ROLE")` are evaluated at **compile time** when the input is a constant string literal. As a result, defining the role as

```
bytes32 public constant UPGRADER_ROLE = keccak256("UPGRADER_ROLE");
```

would produce the **same bytecode and gas cost** as the hardcoded hash.

Using the literal hash therefore does not provide runtime or gas advantages, while making the code less readable. Reviewers must recompute the hash to understand which semantic role it corresponds to, which slightly increases maintenance and auditing overhead.

Recommendation

For clarity and maintainability, consider defining role identifiers using the canonical form. Optionally, the resulting hash may be included as a comment for reference.

If the literal hash was intentionally used (for example to enforce cross-system consistency or avoid recomputation in external tooling), **please clarify the rationale** for this choice.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and commented the hashes in commit f0d12632f59691c610e10d6d5158d470b33cc311.

KGE-18 | Redundant Event Emission

Category	Severity	Location	Status
Coding Issue	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 117~118, 122~123, 127~128, 257~258, 262~263	● Resolved

Description

Some functions emit custom events in addition to calling underlying functions that already emit standard events. For example, `freeze()` and `unfreeze()` , `mint()` and `burn()` .

However, `grantRole()` / `revokeRole()` already emit `RoleGranted` / `RoleRevoked` , and `_mint()` / `_burn()` already emit `Transfer` events. These additional events therefore duplicate information already present in logs.

Recommendation

We recommend that the client rely on the existing `AccessControl` and ERC-20 events to avoid redundant logging. Otherwise, if specific logs are needed, please clarify the rationale for emitting these additional events.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit 452fc0b9320e59be781b59fb32214b65e441352a.

KGE-19 | Inconsistency Between Comment And Code

Category	Severity	Location	Status
Inconsistency	Informational	contracts/CommodityToken/CommodityToken.sol (base): 53~54, 126~127	Resolved

Description

Comment of the `FROZEN_ROLE` describes `RISK_MANAGER_ROLE` as the required role to set an address frozen:

```
53      // @notice only RISK_MANAGER_ROLE can add/remove accounts to/from this role
```

However, the function `selfFreeze()` allows any user to set oneself as a frozen address.

Recommendation

We recommend correcting the comment to ensure accuracy.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit 78f7085afb291ca1a2da973cf525300d237b375e.

KGE-20 | Missing Validation Of Authorization Time Window

Category	Severity	Location	Status
Volatile Code	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 473~474, 532~533	● Resolved

Description

The functions `transferWithAuthorization()` and `receiveWithAuthorization()` restrict execution to the interval between `validAfter` and `validBefore`. However, they do not validate that `validAfter <= validBefore`. As a result, malformed authorizations with `validBefore < validAfter` would define an impossible time window and always revert with `AuthorizationExpired`, which may be misleading.

Recommendation

We recommend that the client add an explicit check ensuring that `validAfter <= validBefore` to prevent malformed authorization windows and improve clarity of failure conditions.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit 4e3ba8a5833a5aba742eb83b1922431b72fff724.

KGE-21 | Unnecessary `userOrders` Zero-Check In `RedeemLock()

Category	Severity	Location	Status
Coding Issue	Informational	contracts/RedeemLock/RedeemLock.sol (base): 72~75	Resolved

Description

`redeemLock()` verifies `userOrders[msg.sender][nonce] == bytes32(0)` for `nonce = userNonce[msg.sender]` , but `userNonce` is only ever incremented and nonce slots are never reused or cleared. This makes the check effectively unreachable in normal operation and adds unnecessary gas/code complexity.

Recommendation

We recommend that the client remove the `userOrders[msg.sender][nonce] == bytes32(0)` requirement and rely on the monotonic `userNonce` invariant.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit 5f57a6096e37e67fca62a8a9efcffd1f24dc6c2b.

KGE-22 `vaultedWeight` Is Non-Operational And Unsigned In `mintWithAuthorization()`

Category	Severity	Location	Status
Design Issue	Informational	contracts/CommodityToken/CommodityToken.sol (base): 282~283	Resolved

Description

Function `mintWithAuthorization(...)` expects a `vaultedWeight` parameter but does not use it in any state transition or validation; it is only emitted as an event. Additionally, `vaultedWeight` is not included in the EIP-712 signed payload, so the `MINT_APPROVER_ROLE` signature does not attest to its value, limiting its usefulness as an authoritative on-chain record.

Recommendation

We kindly request that the client clarify whether `vaultedWeight` is intended to be purely informational for off-chain reporting, or whether it should be a verifiable/authoritative datum tied to the mint approval (i.e., included in the signed data and/or stored on-chain). In general, we recommend that the client enforce an on-chain implementation of `vaultedWeight` compatible with its intended specification and guarantees.

Alleviation

[CertiK, 04/07/2026]: The team heeded the advice and removed the `vaultedWeight` parameter in commit dce4e529e9b223c3997aa46f84382d73cbf284f9.

KGE-24 | Invalid Use Of Access Control Modifier

Category	Severity	Location	Status
Logical Issue	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 142~143, 337~338, 341~342, 345~346, 366~369, 372~373, 376~380	● Resolved

Description

The functions marked as `view` or `pure` are unnecessarily restricted by the `onlyProxy` modifier. These functions are designed to be read-only, meaning they do not modify the state on the blockchain. However, they are restricted so that only a specific address can call them.

It is important to note that even private state variables can be read off-chain, rendering the access restriction on these functions ineffective.

Recommendation

We recommend that access restrictions are not used on `view` or `pure` functions, as they do not improve security for read-only operations. Instead, these getter functions should be made public to allow transparency and follow best practice. If there is sensitive information that should not be disclosed, the way in which this data is managed and stored should be reconsidered, as restricting access in this way does not provide effective security.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue in commit cc248971d05983b660b3c450ee81246c923efa38.

KGE-25 | Redundant `ERC20Upgradeable` And `ERC20PermitUpgradeable` Inheritance

Category	Severity	Location	Status
Coding Issue	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 18~19	● Resolved

Description

`CommodityToken` inherits both `ERC20Upgradeable` and `ERC20PermitUpgradeable`, but in OpenZeppelin `ERC20PermitUpgradeable` already builds on top of `ERC20Upgradeable`. Keeping both in the inheritance list is redundant and can add confusion/maintenance overhead around inheritance linearization and overrides

Recommendation

We recommend that the client remove the direct `ERC20Upgradeable` inheritance and keep only `ERC20PermitUpgradeable`.

Alleviation

[CertiK, 04/08/2026]: The team heeded the advice and resolved the issue by deleting redundant `ERC20Upgradeable` import in commit a5f9fbf8536a4046afb01bd0cdeba604634ce075.

KGE-26 | Minor Typos

Category	Severity	Location	Status
Coding Issue	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 116	● Resolved

Description

The codebase contains several minor typos that may impact readability:

1. `"implemeted"` instead of `"implemented"` (twice) in comments within `CommodityToken`

- `"recieve"` instead of `"receive"` in `CommodityTokenIssuer`
- The `OrderStatus` struct uses `Canceled` (single "l") instead of `Cancelled` (double "l"), leading to inconsistency with common spelling conventions

While these issues do not affect functionality, they may reduce code clarity and *perceived* quality.

Recommendation

We recommend that the client review and correct spelling mistakes and standardize naming conventions across the codebase to improve readability and maintain consistency.

Alleviation

[CertiK, 04/07/2026]: The team heeded the advice and resolved the issue in commit f64923bc3e6c1911efaf58fe7056eed0d989630a.

KGE-27 | AuthorizationUsed Event Misattributes Nonce To Caller (Msg.Sender) Instead Of Authorizer (Signer)

Category	Severity	Location	Status
Inconsistency	● Informational	contracts/CommodityToken/CommodityToken.sol (base): 331~332	● Resolved

Description

The `mintWithAuthorization()` flow records nonce usage under the recovered approver (`signer`) but emits an event with `msg.sender` . As a result, the `AuthorizationUsed` event does not reflect the address whose nonce was actually consumed. This event/state mismatch can break off-chain tracking of replay protection and misattribute who approved a mint. It also diverges from other authorization flows in the contract that emit the real authorizer.

Recommendation

We recommend that the client emit `AuthorizationUsed` with the recovered authorizer (signer) rather than `msg.sender` so the event matches the replay-protection key used in `usedNonces[signer][nonce]` . In general, please align all authorization/mint flows to consistently emit the actual authorizer in their events.

Alleviation

[CertiK, 05/06/2026]: The client has applied the suggested changes in commit hash 4845130046735d7acc06a348da7e2a4fa230cd05, thus resolving the finding.

KGE-32 | Inclusive Time Bounds Deviate From EIP-3009 Reference

Category	Severity	Location	Status
Design Issue	● Informational	contracts/CommodityToken/CommodityToken.sol (update_20260408): 483~484	● Resolved

Description

Functions `transferWithAuthorization()` and `receiveWithAuthorization()` check time validity as:

```
if (block.timestamp < validAfter || block.timestamp > validBefore) revert ...
```

This makes both bounds inclusive. The EIP-3009 reference implementation (Circle's USDC, `EIP3009.sol`) uses strict inequalities (`now > validAfter` , `now < validBefore`), treating both bounds as exclusive. As a result, an authorization with `validAfter == validBefore` is executable for one block rather than being invalid, and transfers can execute at the exact boundary timestamps the signer intended as limits.

Recommendation

We recommend that the client replace the condition with strict inequalities: `if (block.timestamp <= validAfter || block.timestamp >= validBefore) revert ...`.

Alleviation

[CertiK, 04/09/2026]: The client has applied the suggested changes in commit hash d07666abc29433cc7c398525797ec2edc9d57af9 , thus resolving the finding.

KGE-34 | Issuer Validates Whitelist But Not Freeze Status, Granting Users A Costless Unilateral Abort On Signed Orders

Category	Severity	Location	Status
Volatile Code	● Informational	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (update_2026_04_24): 310~311	● Acknowledged

Description

`issue()` is gated to `OPERATION_MANAGER_ROLE` (Korea Gold's KMS-managed server) and acts as the protocol's primary swap primitive: the user signs a `permit` committing to a slippage-bounded exchange at an operator-quoted rate, and the operator submits and pays gas.

The issuer pre-validates user state via `checkIsWhitelisted(owner)` but does not check `isFrozen(owner)` . Meanwhile `CommodityToken._update` reverts unconditionally on frozen accounts, and `selfFreeze()` is permissionless by design (per the inline comment *"Users can self-assign FROZEN_ROLE via selfFreeze() without RISK_MANAGER_ROLE"*) — meaning any holder, not just privileged actors, can trigger the abort path. A user who has signed an `issue()` order can therefore call `selfFreeze()` between submission and inclusion to force the operator's transaction to revert at `safeTransferFrom` / `safeTransfer` .

The concern is not gas loss alone but the asymmetric optionality: the operator commits to an exchange rate at submission, and `selfFreeze()` becomes a costless unilateral abort if the gold price moves against the user before inclusion (free-option / last-look pattern). Off-chain mitigation is available — the operator can blacklist users post-hoc, and the whitelist caps Sybil exploitation — but the race window is too short to react to the first occurrence per address.

This finding is raised for completeness given the bounded practical impact under the operational mitigations described above.

Recommendation

We recommend that the client either add an `isFrozen(owner)` check in `issue()` mirroring `checkIsWhitelisted(owner)` to close the validation asymmetry, or confirm that the order-routing pipeline blacklists users who self-freeze with pending orders. Please document the chosen approach.

KGE-35 | Withdraw Paths Trust Nominal Amounts: Token Onboarding Must Exclude Asymmetric-Debit Semantics

Category	Severity	Location	Status
Volatile Code	Informational	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (update_20260409): 459~460, 481~482	Partially Resolved

Description

`issue()` rejects fee-on-transfer tokens via balance-delta checks against `NotAllowedFeeOnTransfer`. The same check is absent in `withdrawFees()` and `withdrawReserve()`, which trust the nominal `_amt` when updating `cumulatedFees[_ta]`. For tokens whose FOT behavior changes post-deployment, or tokens entering the contract outside `issue()`, withdrawals can debit the issuer by more than the recorded amount, drifting `cumulatedFees[_ta]` from the actual balance.

Replicating the `issue()` check on the withdraw paths is not a clean fix: a mismatch would revert the withdrawal, stranding the affected funds rather than reconciling them. The contract therefore relies on token onboarding to exclude FOT and asymmetric-debit semantics in the first place.

Recommendation

Please confirm that the absence of balance-delta checks on the withdraw paths is intentional, and that token onboarding (`setAvailablePair()`, direct deposits) excludes any token with fee-on-transfer, deflationary, or other asymmetric-debit semantics — including tokens whose behavior could activate post-deployment -, and not just by the operational safeguard in `issue()`.

Alleviation

[Korea Gold Exchange Digital Asset , 04/30/2026]:

- The absence of a balance-delta check in withdrawFees() was not intentional. To address this, we updated the logic so that fee accounting is based on the actual balance delta. This ensures that withdrawals remain functional even if a token introduces fee-on-transfer (FOT) behavior after fees have already been accumulated through issue().

- Regarding token onboarding, it is intentional that tokens with fee-on-transfer, deflationary, or other asymmetric-debit characteristics are excluded. This restriction is applied at the onboarding stage to prevent inconsistencies in accounting and reserve management.

https://github.com/CrederLabs/KGLD_contract/pull/1/changes/d865a3d72d7526aa8003eaba2999ed6a7aaaf02c

---

[CertiK, 04/30/2026]: The **client has confirmed that non-standard ERC20 tokens are excluded and we have updated the audit assumptions accordingly**. However, the revised implementation introduces concerns more significant than the original gap. When `balanceDelta < _amt` (e.g., rebase-up during the transfer, or any post-transfer balance read inflated

relative to the outflow), `cumulatedFees` is debited by less than `_amt` while the recipient still received `_amt` — over time, this inflates the fee bucket against unbacked reserve, silently cannibalizing reserve under fee-claim cover. The saturating `sub256(feeAmt, balanceDelta, false)` compounds this by absorbing the worst case (`balanceDelta > feeAmt`) without any revert or event. We recommend:

1. Reverting to the simple `cumulatedFees -= _amt` decrement and adding the `issue()`-style check `if (balanceDelta != _amt) revert NotAllowedFeeOnTransfer(_ta)` — rejecting non-standard tokens at the withdrawal boundary, consistent with the deposit boundary,
2. And **relying on the token on-boarding** process, with a the delta-check in `issue()` used as a **safeguard**. Adding a recipient-balance delta check is implementable but partially redundant with onboarding discipline (it is unsound for rebasing tokens and incomplete against hook-based or whitelist-gated variants), so onboarding must remain the primary defense.

[Korea Gold Exchange Digital Asset , 05/06/2026]: The fee withdrawal logic has been updated to decrement cumulatedFees based on `_amt` (the requested withdrawal amount), and to revert if the balance delta before/after withdrawal does not exactly match `_amt` , treating such cases as non-standard token behavior.

Non-standard ERC20 tokens (e.g., fee-on-transfer tokens) are rejected during the normal `issue()` and withdrawFees() flows.

For exceptional cases where a token's behavior changes after onboarding, an `emergencyWithdrawFee()` function has been introduced, which can only be executed by `DEFAULT_ADMIN_ROLE` as a higher-privileged recovery path.

https://github.com/CrederLabs/KGLD_contract/pull/1/commits/9f6ca4ee6dd935d58b4aa434410be84478aa503d

---

[CertiK, 05/06/2026]: The migration of `withdrawFees()` to the `issue()` -style balance-delta pattern is correctly implemented and resolves the original concern; that path also incidentally protects against `_to == address(this)` because `balanceDelta == 0 != _amt` reverts.

Two related issues remain in adjacent functions.

1. `RedeemLock.withdrawCosts(address(this), amount)` decrements `cumulatedCosts` before a self-transfer that leaves balance unchanged, erasing the accounting while custody remains in the contract.
2. `CommodityTokenIssuer.emergencyWithdrawFee()` sets `cumulatedFees[_ta] = 0` unconditionally and computes `balanceDelta` only for the event, never validating it; any residual from a non-standard transfer (or a self-withdrawal) is silently reclassified as reserve via `getReserve = balance - cumulatedFees`, which is inconsistent.

We recommend rejecting `to == address(this)` in `withdrawCosts` , and in `emergencyWithdrawFee` replacing `cumulatedFees[_ta] = 0` with `cumulatedFees[_ta] = sub256(feeAmt, balanceDelta)` so the fee accounting tracks the actual transfer outcome — preserving any residual as fees and reverting on unreconcilable states.

[CertiK, 05/07/2026]: The team has explicitly documented `emergencyWithdrawFee()` as an exceptional-use function that intentionally does not enforce balance-delta equality and resets `cumulatedFees[_ta]` to zero unconditionally, with off-chain monitoring of the emitted balanceDelta as the reconciliation safeguard. Combined with the intended DEFAULT_ADMIN_ROLE multisig gating, the residual-reclassification behavior is acceptable on record. We note one residual: calling with `_to == address(this)` zeroes the fee bucket without moving any tokens, which contradicts the

comment's claim that the function "withdraw[s] all accumulated fees". We recommend either rejecting `_to == address(this)` or updating the comment to acknowledge this fee→reserve reclassification path. We thus mark the finding as *Partially* Resolved.

KGE-36 | `renounceRole()` Override Does Not Protect `DEFAULT_ADMIN_ROLE`

Category	Severity	Location	Status
Volatile Code	Informational	contracts/CommodityToken/CommodityToken.sol (update_20260409): 147~148	Resolved

Description

The override blocks renouncing `FROZEN_ROLE` but not `DEFAULT_ADMIN_ROLE`. The sole admin (or last remaining admin) can therefore renounce, permanently orphaning role administration. The risk is purely a self-inflicted footgun — the multisig managing `DEFAULT_ADMIN_ROLE` would have to deliberately call `renounceRole(DEFAULT_ADMIN_ROLE, ...)` — but the existing `FROZEN_ROLE` carveout pattern is trivially extensible.

Recommendation

We recommend that the client extend the override to also revert on `DEFAULT_ADMIN_ROLE`.

Alleviation

[CertiK, 04/30/2026]: The client has applied the suggested changes, thus resolving the finding.

KGE-37 | `mintWithAuthorization()` Shares The EIP-3009 Nonce Bucket

Category	Severity	Location	Status
Volatile Code	● Informational	contracts/CommodityToken/CommodityToken.sol (update_2026_04_24): 285~286	● Acknowledged

Description

`mintWithAuthorization()` consumes `usedNonces[signer][nonce]` for the recovered `MINT_APPROVER_ROLE` signer — the same storage shared with `transferWithAuthorization()` and `receiveWithAuthorization()`. If a `MINT_APPROVER_ROLE` key is ever also used to sign public EIP-3009 authorizations and reuses a nonce across the two flows, whichever authorization executes second reverts as already used.

Recommendation

We recommend that the client operationally enforce that `MINT_APPROVER_ROLE` keys are dedicated to mint approvals and not reused for public EIP-3009 authorizations. Alternatively, give `mintWithAuthorization()` its own nonce mapping to remove the coupling.

Alleviation

[Korea Gold Exchange Digital Asset , 04/30/2026]: To mitigate potential risks, we will explicitly define an operational policy stating that any account holding the VAULT_MINTER_ROLE, which is authorized to execute `mintWithAuthorization()`, must not use EIP-3009 related functions.

KGE-38 | `sub256()` Helper Refactor Reduces Clarity And Reintroduces Avoidable Cost

Category	Severity	Location	Status
Coding Issue, Design Issue	● Informational	contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol (update_2026_04_30): 522~523	● Resolved

Description

The latest remediation introduced a behavioral regression in the `sub256()` helper. This raises several concerns:

1. **The change came late in the remediation cycle and reduces overall code quality.** Refactors of widely-used helpers at finalization are higher-risk than the defect they address.

2. **The boolean flag is semantically ambiguous and error-prone.** Each call site requires the reader to recall whether `true` means "revert on underflow" or its opposite; a future edit flipping or omitting the flag silently changes accounting behavior. Boolean parameters gating fundamentally different control flows are a known anti-pattern.

3. **The reverting branch is gas-suboptimal.** It performs a manual comparison and custom-error revert before subtracting, whereas Solidity ^0.8 native subtraction already reverts on underflow — the branch reimplements compiler behavior at higher cost.

4. **The previous helper had clearer semantics.** The original `sub256()` was unambiguously saturating, distinguishing it from native subtraction at every call site. The cleaner and clearer refactor would have been to *keep* a saturating helper, **rename** it (e.g., `subOrZero()` , `satSub()` ...), and replace each "reverting" call site with native subtraction.

Clarity in widely-used arithmetic helpers is a core component of maintainability and safety.

Recommendation

We recommend that the client replace the boolean-flagged `sub256()` with two distinct primitives: a clearly named saturating helper (e.g., `subOrZero(a, b)`) used only where saturation is intended, and native `a - b` everywhere else. When migrating, **verify that each call site's chosen primitive matches the original codebase's intent to avoid silent semantic drift**.

Alleviation

[CertiK, 05/06/2026]: The revised `sub256()` performs the same operation as Solidity ^0.8's native subtraction — revert on underflow, otherwise return `a - b` — at strictly higher gas cost (the manual comparison runs in addition to the compiler's underflow check). If the goal is a project-specific error rather than the default `Panic(0x11)` , the idiomatic version is:

```
function sub256(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a < b) revert NotAllowedNegativeResult();
    unchecked { return a - b; }
}
```

Otherwise, please just use `a - b`.

We also note that the migration from the previous boolean-flagged `sub256()` to this version may have changed behavior at call sites that relied on the saturation-to-zero semantics in commit 4f23f4d5f83c08e9302b9c6277e69887abeb94c4. A **concrete instance has surfaced in `getAmountOut()`**: `sub256(dOut, _retainingDecimals)` previously saturated to 0 when `_retainingDecimals > dOut` (no truncation), but now reverts — breaking quotes for low-decimal output tokens with high retention values. We kindly ask the team to:

1. Clarify the intent of retaining `sub256()` over native subtraction
2. And re-review every call site to confirm the reverting semantics is intended and that no site silently changed from saturating to reverting.

[Korea Gold Exchange Digital Asset , 05/07/2026]: Initially, `sub256()` was introduced to avoid reverts when `a < b`. However, after reviewing the risks and maintainability concerns of this pattern, we concluded that such helper usage was unnecessary.

Accordingly, all usages of `sub256()` have been removed and replaced with Solidity's native subtraction behavior.

For cases where saturation semantics are intentionally required, explicit logic has been implemented separately at the relevant call sites.

https://github.com/CrederLabs/KGLD_contract/pull/1/commits/8aec9777ddff981fe40e7f9e252fe65135b8f36c

KGE-39 | `wipeFrozenAccount()` Cannot Execute During Pause

Category	Severity	Location	Status
Volatile Code	Informational	contracts/CommodityToken/CommodityToken.sol (update_2026_05_06): 195~196	Resolved

Description

`wipeFrozenAccount()` routes through `_updateForWipe()` , which carries the `whenNotPaused` modifier. During an incident requiring both a paused contract and account confiscation, operators must sequence `unpause → wipe → pause` , briefly restoring transfer capability for all other holders.

Recommendation

We recommend that the client remove `whenNotPaused` from `_updateForWipe()` . The wipe path is administratively gated by `RISK_MANAGER_ROLE` and is precisely the operation that should remain available during incident response.

Alleviation

[CertiK, 05/07/2026]: The client has applied the suggested changes in commit hash 6042b60a26caaec20bda5707e91de5f3908dc2e4, thus resolving the finding.

FORMAL VERIFICATION | KOREA GOLD EXCHANGE DIGITAL ASSET

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

Considered Functions And Scope

Verification of contracts derived from AccessControl v4.4

We verified properties of the public interface of contracts that provide an AccessControl-v4.4 compatible API. This involves:

- The `hasRole` function, which returns `true` if an account has been granted a specific `role`.
- The `getRoleAdmin` function, which returns the admin role that controls a specific `role`.
- The `grantRole` and `revokeRole` functions, which are used for granting a `role` to an account and revoking a `role` from an `account`, respectively.
- The `renounceRole` function, which allows the calling account to revoke a `role` from itself.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
accesscontrol-default-admin-role	AccessControl Default Admin Role Invariance
accesscontrol-getroleadmin-change-state	`getRoleAdmin` Function Does Not Change State
accesscontrol-renouncerole-succeed-role-renouncing	`renounceRole` Successfully Renounces Role
accesscontrol-hasrole-change-state	`hasRole` Function Does Not Change State
accesscontrol-renouncerole-revert-not-sender	`renounceRole` Reverts When Caller Is Not the Confirmation Address
accesscontrol-getroleadmin-succeed-always	`getRoleAdmin` Function Always Succeeds
accesscontrol-grantrole-correct-role-granting	`grantRole` Correctly Grants Role
accesscontrol-hasrole-succeed-always	`hasRole` Function Always Succeeds
accesscontrol-revokerole-correct-role-revoking	`revokeRole` Correctly Revokes Role

Verification Results

Verification Results

Detailed Results For Contract CommodityTokenIssuer (contracts/CommodityTokenIssuer/CommodityTokenIssuer.sol) In Commit dd87c3cf71930c1c946682cba12e092ef9d23710

Verification of contracts derived from AccessControl v4.4

Detailed Results for Function `DEFAULT_ADMIN_ROLE`

Property Name	Final Result	Remarks
accesscontrol-default-admin-role	● True	

Detailed Results for Function `getRoleAdmin`

Property Name	Final Result	Remarks
accesscontrol-getroleadmin-change-state	● True	
accesscontrol-getroleadmin-succeed-always	● True	

Detailed Results for Function `renounceRole`

Property Name	Final Result	Remarks
accesscontrol-renouncerole-succeed-role-renouncing	● True	
accesscontrol-renouncerole-revert-not-sender	● True	

Detailed Results for Function `hasRole`

Property Name	Final Result	Remarks
accesscontrol-hasrole-change-state	● True	
accesscontrol-hasrole-succeed-always	● True	

Detailed Results for Function `grantRole`

Property Name	Final Result	Remarks
accesscontrol-grantrole-correct-role-granting	● True	

Detailed Results for Function `revokeRole`

Property Name	Final Result	Remarks
accesscontrol-revokerole-correct-role-revoking	● True	

Detailed Results For Contract RedeemLock (contracts/RedeemLock/RedeemLock.sol) In Commit dd87c3cf71930c1c946682cba12e092ef9d23710

Verification of contracts derived from AccessControl v4.4

Detailed Results for Function `renounceRole`

Property Name	Final Result	Remarks
accesscontrol-renouncerole-revert-not-sender	● True	
accesscontrol-renouncerole-succeed-role-renouncing	● True	

Detailed Results for Function `DEFAULT_ADMIN_ROLE`

Property Name	Final Result	Remarks
accesscontrol-default-admin-role	● True	

Detailed Results for Function `hasRole`

Property Name	Final Result	Remarks
accesscontrol-hasrole-succeed-always	● True	
accesscontrol-hasrole-change-state	● True	

Detailed Results for Function `getRoleAdmin`

Property Name	Final Result	Remarks
accesscontrol-getroleadmin-change-state	● True	
accesscontrol-getroleadmin-succeed-always	● True	

Detailed Results for Function `revokeRole`

Property Name	Final Result	Remarks
accesscontrol-revokerole-correct-role-revoking	● True	

Detailed Results for Function `grantRole`

Property Name	Final Result	Remarks
accesscontrol-grantrole-correct-role-granting	● True	

APPENDIX | KOREA GOLD EXCHANGE DIGITAL ASSET

Finding Categories

Categories	Description
Governance	Governance findings indicate issues related to the management of the code.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the

semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed AccessControl-v4.4 Properties

Properties related to function `DEFAULT_ADMIN_ROLE`

accesscontrol-default-admin-role

The default admin role must be invariant, ensuring consistent access control management.

Specification:

```
invariant DEFAULT_ADMIN_ROLE() == 0x00;
```

Properties related to function `getRoleAdmin`

accesscontrol-getroleadmin-change-state

The `getRoleAdmin` function must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrol-getroleadmin-succeed-always

The `getRoleAdmin` function must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `renounceRole`

accesscontrol-renouncerole-revert-not-sender

The `renounceRole` function must revert if the caller is not the same as `account`.

Specification:

```
reverts_when account != msg.sender;
```

accesscontrol-renouncerole-succeed-role-renouncing

After execution, `renounceRole` must ensure the caller no longer has the renounced role.

Specification:

```
ensures !hasRole(role, account);
```

Properties related to function `hasRole`

accesscontrol-hasrole-change-state

The `hasRole` function must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrol-hasrole-succeed-always

The `hasRole` function must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `grantRole`

accesscontrol-grantrole-correct-role-granting

After execution, `grantRole` must ensure the specified account has the granted role.

Specification:

```
ensures hasRole(role, account);
```

Properties related to function `revokeRole`

accesscontrol-revokerole-correct-role-revoking

After execution, `revokeRole` must ensure the specified account no longer has the revoked role.

Specification:

```
ensures !hasRole(role, account);
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.

 Copyright © CertiK