Final report solidity bugs

khiari mohamed

`C:\Users\LENOVO\Desktop\solidity\2025-04-kinetiq\2025-04-kinetiq\test\Base.t.sol`

1- Missing `onlyProxy` Modifier Allows Dangerous Direct Calls to Logic Contract

Risk Rating
High

Summary
The logic contract does not enforce a proxy-only usage restriction. Without this safeguard, developers or integrators may mistakenly interact with the logic contract directly rather than through its proxy. Such direct interaction can cause the contract to be misconfigured or permanently bricked, especially if initialization or role-setting functions are called directly.

Proof of Concept (PoC)
A developer initializes the logic contract directly instead of using the proxy. This sets state variables in the logic contract's storage, not the proxy's. As a result, the proxy contract remains uninitialized, causing misbehavior or requiring a redeployment.

```
LogicContract logic = new LogicContract();
```

`logic.initialize(); // Should be called through proxy, but was called directly`.

Impact on the Project
Misuse of the logic contract can lead to:

- Permanent misconfiguration
- Improper role assignments
- Critical functions becoming unusable
- Potential loss of control over the contract
- Costly redeployment and downtime

Recommendation or Fix
Enforce a proxy-only access check by adding the following in critical initializer or role-setting functions:

```
require(address(this) != implementation, "Must be called via proxy");
```

/ValidatorManagerTest.t.sol

Title

2-Missing Tests for Core Staking Functions: recordStake() and recordClaim()

Risk Rating

High

Summary

The core staking functions recordStake() and recordClaim() are currently untested. These functions play a crucial role in tracking user stake activity and claims within the contract. Without proper unit or integration tests, there's a significant risk that they may behave incorrectly or introduce silent bugs into the system.

Proof of Concept (PoC)

Currently, the test suite does not include any assertions related to:

- User balance updates after staking
- Claiming behavior (emitted events or state change)
- Failure cases or input validation

```
staking.recordStake(user, amount);

// No assertion to verify user's stake balance was updated
```

- **Impact on the Project**
  - Core staking operations may break silently
  - Users could lose rewards or have incorrect balances
  - Bugs could go undetected into production
  - Security vulnerabilities might arise due to lack of behavioral validation
- **Recommendation or Fix**

  Implement thorough tests for both recordStake() and recordClaim():
  - Validate correct balance and state updates after a stake
  - Check that recordClaim() emits expected events and updates internal state correctly
  - Include edge cases and failure conditions
  - Use mocks or test helpers to simulate realistic staking scenarios

```
function test_RecordStakeUpdatesBalance() public {

    uint256 stakeAmount = 1000 ether;

    accountant.recordStake(user, address(mockToken), stakeAmount);

    uint256 balance = accountant.getStakedBalance(user, address(mockToken));

    assertEq(balance, stakeAmount, "Stake balance should be updated correctly");

}

function test_RecordClaimUpdatesClaimedAmount() public {

    uint256 claimAmount = 500 ether;
```

```
    accountant.recordClaim(user, address(mockToken), claimAmount);

    uint256 claimed = accountant.getClaimedAmount(user, address(mockToken));

    assertEq(claimed, claimAmount, "Claimed amount should be tracked correctly");

}
```

ValidatorManagerTest.t.sol

Title

3-Untested Withdrawal and Delegation Logic

Risk Rating

High

Summary

The contract lacks tests for critical withdrawal and delegation flows. While role validations are in place, the actual business logic for queuing and confirming withdrawals, as well as delegation and validator interactions, is completely untested. This poses a major risk to contract integrity, especially in user-facing financial operations.

Proof of Concept (PoC)

Current test coverage does **not** include:

- queueWithdrawal()
- confirmWithdrawal()
- delegate()
- validatorWithdraw()

This means:

User funds could become locked due to logic errors.

Validators may not be properly registered or interacted with.

Events might not be emitted, affecting frontend UX.

```
staking.queueWithdrawal(user, amount); // no follow-up test to confirm expected behavior

staking.confirmWithdrawal(user); // not checked if funds are released or state updated
```

Impact on the Project

- Users may experience frozen funds if withdrawal logic fails
- Delegation issues could halt staking operations or validator rewards
- Undetected regressions in these flows could damage trust in the protocol
- Potential legal or compliance issues for financial protocols

Recommendation or Fix

Write comprehensive unit and integration tests that cover:

Add access control tests that explicitly verify unauthorized addresses **cannot** call restricted functions. Use test frameworks like Foundry or Hardhat to simulate calls from unprivileged users.

Withdrawals can be queued and then confirmed (covering both logic and emitted events).

```
function test_QueueWithdrawal() public {

 vm.startPrank(operator);

 uint256 amount = 1000 ether;

 uint256 withdrawalId = stakingManager.queueWithdrawal(address(this), amount);

 vm.expectEmit(true, true, true, true);

 emit WithdrawalQueued(address(stakingManager), address(this), withdrawalId, amount, 0, 0);

 vm.stopPrank();

}

function test_ConfirmWithdrawal() public {

 vm.startPrank(operator);

 uint256 amount = 1000 ether;

 uint256 withdrawalId = stakingManager.queueWithdrawal(address(this), amount);

 vm.stopPrank();

 vm.startPrank(manager);

 vm.expectEmit(true, true, true, true);

 emit WithdrawalConfirmed(address(this), withdrawalId, amount);

 stakingManager.confirmWithdrawal(address(this), withdrawalId);

 vm.stopPrank();

}
```

Delegation to a validator triggers the expected logic and emits the correct event.

```
function test_DelegateToValidator() public {

 vm.startPrank(operator);
```

```
 uint256 amount = 500 ether;

 vm.expectEmit(true, true, true, true);

 emit Delegate(address(stakingManager), validator, amount);

 stakingManager.delegate(validator, amount);

 vm.stopPrank();

}
```

IValidatorSanityChecker.sol

Title

4-Missing Unauthorized Access Tests for Critical Functions

Risk Rating

High

Summary

Several sensitive functions — such as queueWithdrawal, delegate, and updateStakingLimits — lack tests that ensure only authorized roles (e.g., admin, operator, manager) can invoke them. Without these tests, there's a serious risk that unauthorized users could exploit misconfigured access controls to manipulate the staking system or validator behavior.

Proof of Concept (PoC)

No test currently asserts that unauthorized addresses are **rejected** from performing the following:

```
staking.queueWithdrawal(user, amount); // should be restricted

staking.delegate(user, validator); // should require manager/operator

staking.updateStakingLimits(min, max); // should require admin/manager

vm.prank(attacker); // simulate call from unauthorized user

staking.delegate(user, maliciousValidator); // this must revert but currently isn't tested
```

Impact on the Project

- Unauthorized users could:
  - Queue fake withdrawals
  - Hijack delegation logic
  - Change critical parameters like staking limits
- Could result in:
  - Security breaches
  - Manipulation of validators

    - Disruption of staking economy
- Potential loss of funds or project downtime

Recommendation or Fix

Add access control tests that explicitly verify unauthorized addresses **cannot** call restricted functions. Use test frameworks like Foundry or Hardhat to simulate calls from unprivileged users.

```
function test_UnauthorizedCannotQueueWithdrawal() public {
 vm.prank(address(0xdead));
 vm.expectRevert();
 stakingManager.queueWithdrawal(address(this), 100 ether);
}
function test_UnauthorizedCannotDelegate() public {
 vm.prank(address(0xbeef));
 vm.expectRevert();
 stakingManager.delegate(validator, 500 ether);
}
function test_UnauthorizedCannotUpdateStakingLimits() public {
 vm.prank(address(0x1234));
 vm.expectRevert();
 stakingManager.updateStakingLimit(100000 ether);
}
```

ValidatorManager

Title

5-Missing Tests for Validator Activation and Deactivation Logic

Risk Rating

High

Summary

The contract includes logic to activate validators, but lacks test coverage for related scenarios, including deactivating validators and re-activating already active ones. These omissions present

a risk to validator lifecycle management and could lead to inconsistent or insecure validator states.

Proof of Concept (PoC)

Currently, there are **no tests** validating:

- deactivateValidator()
- Re-activation of an already active validator
- Attempting to deactivate a validator that is not active (should revert)

Example risk:

```
staking.deactivateValidator(validator); // no check if already inactive
```

```
staking.activateValidator(validator); // re-activation not tested
```

Missing these tests can result in:

- Double activation bugs
- Inconsistent validator registry state
- Unintended permission or reward issues

Impact on the Project

- Validators may remain unintentionally active or inactive
- The protocol could continue interacting with a decommissioned validator
- Security risks arise from mismanaged validator access
- Performance and decentralization could suffer due to incorrect validator states

Recommendation or Fix

Activate and then deactivate a validator

Attempt to deactivate a non-active validator (should revert)

```
function test_ActivateAndDeactivateValidator() public {
```

```
 vm.prank(manager);
```

```
 validatorManager.activateValidator(validator1);
```

```
 assertTrue(validatorManager.isActiveValidator(validator1));
```

```
 vm.prank(manager);
```

```
 validatorManager.deactivateValidator(validator1);
```

```
 assertFalse(validatorManager.isActiveValidator(validator1));
```

```
}
```

Revert when deactivating a non-active validator

```
function test_DeactivateNonActiveValidatorShouldRevert() public {

 vm.prank(manager);

 vm.expectRevert("Validator is not active");

 validatorManager.deactivateValidator(validator1); // validator1 not activated yet

}
```

---

updateValidatorPerformance

Title

6-Missing Negative Access Control Tests for updateValidatorPerformance

Risk Rating

High

Summary

The updateValidatorPerformance function is intended to be callable **only** by the oracleManager. However, there are currently no tests that verify access is properly restricted. Without negative test coverage, there's a high risk that unauthorized addresses could call this function and manipulate validator performance scores.

Proof of Concept (PoC)

There is **no test** like this:

There is **no test** like this:

```
vm.prank(attacker);

staking.updateValidatorPerformance(validator, newScore); // should revert
```

This means malicious actors could potentially:

- Inflate their own validator performance scores
- Sabotage others by setting poor scores

Impact on the Project

- Staking rewards could be manipulated
- Validator rankings could be corrupted
- Loss of trust in the oracle system
- Distorted network economics

Recommendation or Fix

Add **negative access control tests** to ensure only the `oracleManager` can call `updateValidatorPerformance`.

```
function test_UpdatePerformance_UnauthorizedRevert() public {
 vm.expectRevert("AccessControl: account is missing role");
 validatorManager.updateValidatorPerformance(
 validator1,
 100 ether,
 8000,
 7500,
 9000,
 8500
 );
}
```

ValidatorSanityChecker

Title
7- Lack of Access Control on Critical Setter Functions

Risk Rating
High

Summary
The contract exposes setter functions for modifying validator sanity check tolerances — including slashing, rewards, and score thresholds — without any access control. This means **any user** could call these functions and manipulate core validation logic, potentially destabilizing the protocol.

Proof of Concept (PoC)
These functions can be called without restrictions:

```
staking.setSlashingTolerance(99);
staking.setRewardTolerance(0);
staking.setScoreTolerance(1);
```

Example attack:

```
vm.prank(attacker); // any user
staking.setSlashingTolerance(0); // disables slashing
```

```
staking.setScoreTolerance(100); // disables validator scoring
```

This lack of restriction can let attackers:

- Disable validator sanity checks
- Make underperforming validators look valid
- Avoid penalties or artificially inflate rewards

Impact on the Project

- Malicious actors could:
  - Prevent slashing of faulty validators
  - Falsify reward conditions
  - Disrupt staking fairness and decentralization
- Undermines protocol's ability to enforce validator performance
- Loss of trust from users and delegators

Recommendation or Fix

Implement strict access control using Ownable or AccessControl from OpenZeppelin to ensure **only the contract owner** (or designated roles) can modify these values.

```
import "@openzeppelin/contracts/access/Ownable.sol"; // Imported Ownable

contract ValidatorSanityChecker is IValidatorSanityChecker, Ownable { // Inherited Ownable

// Added onlyOwner modifier to setter functions

function setSlashingTolerance(uint256 newTolerance) external onlyOwner {

 require(newTolerance <= 2000, "Tolerance too high");

 slashingTolerance = newTolerance;

 emit SlashingToleranceUpdated(newTolerance);

}

function setRewardsTolerance(uint256 newTolerance) external onlyOwner {

 require(newTolerance <= 3000, "Tolerance too high");

 rewardsTolerance = newTolerance;

 emit RewardsToleranceUpdated(newTolerance);

}

function setScoreTolerance(uint256 newTolerance) external onlyOwner {

 require(newTolerance <= 5000, "Tolerance too high");

 scoreTolerance = newTolerance;
```

```
 emit ScoreToleranceUpdated(newTolerance);

}

function setMaxScoreBound(uint256 newBound) external onlyOwner {

 require(newBound > 0, "Bound must be positive");

 maxScoreBound = newBound;

 emit MaxScoreBoundUpdated(newBound);

}
```

/src/lib/

Title

8-Unsafe Assumptions on staticcall Return Values

Risk Rating

High

Summary

The contract assumes that calls using staticcall to fetch data from view functions like position(), spotBalance(), and oraclePx() will always return properly encoded data. However, this approach overlooks the possibility that these calls may fail silently or return malformed data, leading to failed decoding operations and potentially undefined behavior in the protocol.

Proof of Concept (PoC)

Example of the unsafe assumption:

```
(bool success, bytes memory data) = target.staticcall(abi.encodeWithSignature("oraclePx()"));

require(success, "Call failed");

uint256 price = abi.decode(data, (uint256)); // Assumes data is always valid
```

If data is not properly encoded or empty, this decode call can:

- Revert unexpectedly
- Decode into invalid or misleading values (e.g., zero or garbage)
- Corrupt logic that relies on the decoded value

Impact on the Project

- Leads to silent state corruption if decoded values are invalid
- Could cause incorrect reward calculations, balance errors, or faulty oracle readings
- May leave developers blind to subtle bugs in contract interactions
- Undermines reliability and trust in protocol correctness

Recommendation or Fix

Always validate both the success flag and the **length/format** of returned data before decoding. Add decoding guards to ensure safe parsing of the result.

You may also include an internal helper to centralize safe staticcall logic:

```
(bool success, bytes memory result) = <PRECOMPILE>.staticcall(...);
```

```
require(success && result.length > 0, "L1Read: <context> fetch failed");
```

KHYPE.sol

Title

9-Insufficient Input Validation in initialize()

Risk Rating

High

Summary

The initialize() function accepts several critical addresses — such as admin, minter, and burner — and only checks whether they are non-zero. It fails to enforce more stringent validation, such as verifying that these addresses are **not reused across roles**, or that they conform to expected behavior (e.g., being EOAs vs. contracts). This opens up risks of accidental misconfiguration or unintended privilege overlaps.

Proof of Concept (PoC)

```
function initialize(address admin, address minter, address burner) external {
```

```
 require(admin != address(0), "Zero admin");
```

```
 require(minter != address(0), "Zero minter");
```

```
 require(burner != address(0), "Zero burner");
```

```
 // Allows same address to hold multiple critical roles
```

```
 // No check if addresses are contracts that might self-destruct or misbehave
```

```
}
```

```
// Misconfiguration example
```

```
initialize(0xABC..., 0xABC..., 0xABC...); // Same address assigned all roles
```

Impact on the Project

- Role confusion or privilege overlap may occur, allowing:
  - A minter to also pause the contract (if admin)

    - A single compromised address to control all core functions
- If a contract address is used, its fallback behavior could alter initialization flow
- Difficulties in auditing and maintaining clear role separation

Recommendation or Fix

Add explicit validation to:

- Ensure no two roles are assigned the same address
- Optionally ensure EOAs (e.g., via tx.origin != address(0) or similar heuristics)
- Prevent known unsafe contracts from being assigned roles

```
require(admin != minter, "Admin and Minter must be different");

require(admin != burner, "Admin and Burner must be different");

require(minter != burner, "Minter and Burner must be different");

// Optional: prevent assigning contract addresses if only EOAs are desired

require(!isContract(admin), "Admin must not be a contract");

require(!isContract(minter), "Minter must not be a contract");

require(!isContract(burner), "Burner must not be a contract");

function isContract(address account) internal view returns (bool) {

  return account.code.length > 0;

}
```

KHYPE.sol

Title

10-Potential Pauser Registry Centralization Risk

Risk Rating

High

Summary

The contract's pause control is delegated to an external PauserRegistry, which is trusted as the sole authority to enforce the whenNotPaused modifier. This introduces a centralization vulnerability — if the registry is controlled by a single actor or is compromised, the entire token functionality (minting, burning, transferring) can be halted indefinitely. There's currently no safeguard or fallback mechanism against misuse or abuse of the registry's pause function.

Proof of Concept (PoC)

The whenNotPaused modifier likely relies on the following logic:

```
modifier whenNotPaused() {
 require(!PauserRegistry.isPaused(), "Contract is paused");
 _;
}
```

If PauserRegistry is controlled by a single key or lacks governance checks:

- A rogue or compromised admin could call pause() and lock the contract
- No multi-party verification or time-delay prevents sudden disruption
- Critical operations like staking, rewards, and withdrawals are frozen

Impact on the Project

- Loss of user trust and functionality if contract is paused indefinitely
- Project availability could be held hostage by a single entity
- Undermines decentralization goals and exposes the system to governance attacks or admin key compromises

Recommendation or Fix

Replace or harden the centralized pause mechanism by:

- Using **multi-signature control** for pausing/unpausing (e.g., Gnosis Safe)
- Adding a **PAUSE_MANAGER_ROLE** role with strict access control and traceability
- Introducing **time-delayed pause actions** via on-chain governance
- Emitting events for pause/unpause with strict logging/auditing

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

function setPauserRegistry(address newRegistry) external onlyRole(PAUSE_MANAGER_ROLE) {
 require(newRegistry != address(0), "Invalid address");
 pauserRegistry = IPauserRegistry(newRegistry);
 emit PauserRegistryUpdated(newRegistry);
}

event PauserRegistryUpdated(address newRegistry);
```

OracleManager.sol

Title

11- Missing Strict Role-Based Access Control

Risk Rating

High

Summary

The contract defines roles like DEFAULT_ADMIN_ROLE, MANAGER_ROLE, and OPERATOR_ROLE, but some critical functions do not strictly enforce these roles using onlyRole(...). This oversight creates an access control vulnerability, where unauthorized or unintended addresses may invoke privileged logic. Particularly concerning are functions that influence oracle adapter authorizations and core protocol configurations, which can severely affect validator performance and staking outcomes if misused.

Proof of Concept (PoC)

a function like:

```
function updateOracleAdapter(address adapter) external {

 oracleAdapter = adapter;

}

Without enforcing role-based access:

// Missing: onlyRole(MANAGER_ROLE) or similar
```

Any external caller could update the adapter to:

- A malicious contract that returns fake price data
- A broken implementation that causes invalid rewards or slashing events

Impact on the Project

- Oracle data manipulation could cause reward misallocations or unjust validator slashing
- Loss of protocol integrity and stakeholder trust
- Critical contract functions may be altered without accountability
- Opens attack vectors through internal misconfiguration or third-party interference

Recommendation or Fix

Perform a **comprehensive audit** of all external/public functions
Apply **OpenZeppelin's onlyRole(...) modifier** to every function needing restrictions
Clearly map role responsibilities and ensure consistency across function access

```
function authorizeOracleAdapter(address adapter) external whenNotPaused onlyRole(MANAGER_ROLE)

function deauthorizeOracle(address adapter) external whenNotPaused onlyRole(MANAGER_ROLE)

function setOracleActive(address adapter, bool active) external whenNotPaused
onlyRole(MANAGER_ROLE)

function generatePerformance(address validator) external whenNotPaused onlyRole(OPERATOR_ROLE)
```

```
function setMaxPerformanceBound(uint256 newBound) external onlyRole(OPERATOR_ROLE)

function setMinUpdateInterval(uint256 newInterval) external onlyRole(MANAGER_ROLE)

function setMaxOracleStaleness(uint256 newStaleness) external onlyRole(MANAGER_ROLE)
```

PauserRegistry.sol

Title
12- Emergency Pause May Block Critical Operations

Risk Rating
High

Summary
The emergencyPauseAll() function halts all registered contracts simultaneously. While this is a valuable tool in emergencies, the lack of safeguards introduces a major centralization risk. If triggered accidentally or maliciously, this function could freeze core contract functionality across the protocol, leading to total service disruption or blocking user funds.

Proof of Concept (PoC)
Consider a scenario where the function is called by a single admin key:

```
function emergencyPauseAll() external onlyRole(DEFAULT_ADMIN_ROLE) {

 for (uint256 i = 0; i < registeredContracts.length; i++) {

 IPausable(registeredContracts[i]).pause();

 }

}
```

If the admin wallet is compromised, or someone mistakenly triggers this, **every contract** relying on this registry would become paused — stopping staking, minting, transfers, or other operations without recourse.

Impact on the Project

- Critical components like staking, validator operations, or token transfers can become **unavailable**
- Contract bootstrapping and reinitialization flows may silently fail due to hidden paused states
- If triggered unintentionally, could require full redeployments or governance intervention to recover
- Introduces **DoS potential** from a single point of control

Recommendation or Fix

Restrict `emergencyPauseAll()` to a highly secure, multi-signature role, such as an Emergency Council

Implement a **two-step process**:

1. A request to pause all (logged)
2. A confirmation after a delay or quorum approval
   Log all emergency pause calls with clear reasoning
   Optionally require **cooldown periods** between subsequent emergency actions

```
// Declare a proposed pause flag and timestamp

bool public isEmergencyPausePending;

uint256 public emergencyPauseRequestTime;

uint256 public constant EMERGENCY_PAUSE_DELAY = 1 hours;

// Request phase

function requestEmergencyPause() external onlyRole(PAUSE_ALL_ROLE) {

 require(!isEmergencyPausePending, "Emergency pause already requested");

 isEmergencyPausePending = true;

 emergencyPauseRequestTime = block.timestamp;

}

// Finalize phase

function finalizeEmergencyPause() external onlyRole(PAUSE_ALL_ROLE) {

 require(isEmergencyPausePending, "No pending emergency pause");

 require(block.timestamp >= emergencyPauseRequestTime + EMERGENCY_PAUSE_DELAY, "Delay not passed");

 uint256 length = _authorizedContracts.length();

 for (uint256 i = 0; i < length; i++) {

 address contractAddress = _authorizedContracts.at(i);

 if (!isPaused[contractAddress]) {

 isPaused[contractAddress] = true;

 emit ContractPaused(contractAddress);

 }

 }
```

```
 // Reset

 isEmergencyPausePending = false;

 emergencyPauseRequestTime = 0;

}
```

StakingAccountant.sol

Title
13- Redundant Access Control via Modifier and Role

Risk Rating
High

Summary
The functions recordStake() and recordClaim() are protected by both a custom onlyAuthorizedManager modifier and the onlyRole(MANAGER_ROLE) check. This **redundant access control** introduces unnecessary complexity, risks of misconfiguration, and inconsistent authorization logic. If one of the mechanisms is updated or misapplied, it may unintentionally allow or deny access, affecting core staking operations.

Proof of Concept (PoC)

```
modifier onlyAuthorizedManager() {

 require(authorizedManagers[msg.sender], "Not an authorized manager");

 _;

}

function recordStake(...) external onlyAuthorizedManager onlyRole(MANAGER_ROLE) {

 // logic...

}
```

· Problem:

- If a manager has MANAGER_ROLE but isn't in authorizedManagers, access is denied.
- If an address is in authorizedManagers but **not** assigned MANAGER_ROLE, they can **still be blocked**, depending on check order.
- Inconsistent logic across the codebase leads to fragile access assumptions.

Impact on the Project

- **Unexpected reverts** or privilege escalations during staking operations
- Higher **maintenance burden** for future upgrades

- Increased **risk of misconfiguration**, especially during contract upgrades or role assignments
- Could block managers from operating even if assigned correctly

Recommendation or Fix

Choose a single, standardized method for access control — either:

- Use onlyRole(MANAGER_ROLE) (recommended, consistent with OpenZeppelin best practices), OR
- Stick with onlyAuthorizedManager, but then you must **completely remove the role-based check**

Ensure that **documentation** clearly outlines who is allowed to access these functions
Refactor contract for consistency across similar functions

Example Fix (Preferred with OpenZeppelin roles):

1. Modifier definition:

```
modifier onlyAuthorizedManager() {
 require(_authorizedManagers.contains(msg.sender), "Not authorized");
 _;
}
```

2. recordStake() function:

```
function recordStake(uint256 amount) external override onlyAuthorizedManager {
 totalStaked += amount;
 emit StakeRecorded(msg.sender, amount);
}
```

3. recordClaim() function:

```
function recordClaim(uint256 amount) external override onlyAuthorizedManager {
 totalClaimed += amount;
 emit ClaimRecorded(msg.sender, amount);
}
```

Note: These functions are **only protected** with onlyAuthorizedManager and **not** onlyRole(MANAGER_ROLE), but the project overall uses both mechanisms, as seen in authorizeStakingManager():

function authorizeStakingManager(...) external onlyRole(MANAGER_ROLE)

Suggested Fix

Replace the use of the custom modifier with a standardized role check, or vice versa.

```
function recordStake(uint256 amount) external override onlyRole(MANAGER_ROLE) {
 totalStaked += amount;
 emit StakeRecorded(msg.sender, amount);
```

ValidatorManager.sol

14-Title: Potential Reentrancy Issue in rebalanceWithdrawal()

Risk Rating: High

Summary:

The rebalanceWithdrawal() function updates internal contract state and then makes an external call to stakingManager.processValidatorWithdrawals(). Although the contract uses ReentrancyGuardUpgradeable, making external calls **after** internal state changes without strict reentrancy protections can introduce vulnerabilities — especially if the called contract is untrusted or not fully under your control.

Proof of Concept (PoC):

```
function rebalanceWithdrawal() external {
 // State update (internal bookkeeping)
 withdrawalQueued = false;
 // External call — could be hijacked for reentrancy if stakingManager is malicious
 stakingManager.processValidatorWithdrawals();
}
```

An attacker could deploy a malicious stakingManager that calls back into rebalanceWithdrawal() (or other vulnerable entrypoints), leading to unexpected behavior or double execution paths.

Impact on the Project:

A reentrancy vulnerability in withdrawal logic could lead to **double-withdrawals**, corrupted validator accounting, or denial-of-service scenarios. In high-value DeFi or staking ecosystems, such exploits could result in **loss of funds** or **protocol manipulation**, severely undermining user trust and system stability.

Recommendation or Fix:

- Apply the **Checks-Effects-Interactions** pattern — perform all external calls *after* completing all internal state updates.
- Additionally, **explicitly add nonReentrant** modifier for added security if it's not already present:

```
function rebalanceWithdrawal(

 address stakingManager,

 address[] calldata validators,

 uint256[] calldata withdrawalAmounts

) external whenNotPaused nonReentrant onlyRole(MANAGER_ROLE) {

 require(validators.length == withdrawalAmounts.length, "Length mismatch");

 require(validators.length > 0, "Empty arrays");

 for (uint256 i = 0; i < validators.length; ) {

 require(validators[i] != address(0), "Invalid validator address");

 // Add rebalance request

 _addRebalanceRequest(stakingManager, validators[i], withdrawalAmounts[i]);

 unchecked {

 ++i;

 }

 }

 // External call AFTER all state updates

 IStakingManager(stakingManager).processValidatorWithdrawals(validators, withdrawalAmounts);

}
```

ValidatorManager.sol

15-Title: Missing Access Control in updateValidatorPerformance()

Risk Rating: High

Summary:

The updateValidatorPerformance() function is intended to be called only by addresses with the ORACLE_MANAGER_ROLE. However, if the role governance system is not robust — meaning roles can be assigned or reassigned arbitrarily — this function becomes a critical attack vector. A malicious or compromised actor with access to this role could manipulate validator performance data, skewing reward distribution and validator reputation unfairly.

Proof of Concept (PoC):

```
function updateValidatorPerformance(uint256 validatorId, uint256 score) external
onlyRole(ORACLE_MANAGER_ROLE) {

    validatorScores[validatorId] = score;

}
```

Without strict governance over who gets the ORACLE_MANAGER_ROLE, anyone granted it could set performance scores arbitrarily.

Impact on the Project:

If an attacker gains access to updateValidatorPerformance(), they could manipulate validator rankings, redirect staking rewards, or suppress legitimate validators. This undermines the fairness and credibility of the staking system, damaging trust among participants and potentially leading to validator or delegator loss of funds or exit from the protocol.

Recommendation or Fix:

- Restrict the role assignment for ORACLE_MANAGER_ROLE to a multisig, DAO-controlled address, or a verified off-chain oracle coordinator.
- Audit and lock down the AccessControl mechanisms to ensure that role reassignment follows secure governance processes.
- Optionally, add logging and delayed execution for critical changes.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";

import "@openzeppelin/contracts/utils/Address.sol";

import "@openzeppelin/contracts/utils/Timers.sol";

contract SecureOracleManager is AccessControl {

// Define the ORACLE_MANAGER_ROLE

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

```
    // Store the multisig or DAO address responsible for assigning roles
    address public oracleManager;
    // Define a delay mechanism for critical role changes (optional)
    uint256 public changeDelay;
    uint256 public pendingRoleChangeTimestamp;
    // Event for logging critical changes
    event OracleManagerChanged(address indexed oldManager, address indexed newManager);
    constructor(address _oracleCoordinator, uint256 _changeDelay) {
        // Initializing the contract with the multisig or oracle coordinator
        oracleManager = _oracleCoordinator;
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);  // Grant the admin role to the contract deployer (trusted address)
        // Set the delay period (in seconds) before allowing role reassignment
        changeDelay = _changeDelay;
    }
    // Only allow Oracle Manager role assignments via multisig or DAO
    function grantOracleManagerRole(address newManager) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(newManager != address(0), "Invalid address");
        require(newManager != oracleManager, "Already the current manager");
        // Enforce a delay before the role reassignment happens
        pendingRoleChangeTimestamp = block.timestamp + changeDelay;

        // Log the pending change
        emit OracleManagerChanged(oracleManager, newManager);
    }
    // Apply the change after the delay has passed
    function applyRoleChange() external {
        require(block.timestamp >= pendingRoleChangeTimestamp, "Change delay not passed");
```

```
        address oldManager = oracleManager;
        oracleManager = address(this); // Assign the new manager securely after delay
        _grantRole(ORACLE_MANAGER_ROLE, oracleManager);
        // Remove the old manager's access
        _revokeRole(ORACLE_MANAGER_ROLE, oldManager);
        // Reset pending change status
        pendingRoleChangeTimestamp = 0;
        // Log the successful change
        emit OracleManagerChanged(oldManager, oracleManager);
    }
    // Function to update validator performance (only by Oracle Manager)
    function updateValidatorPerformance(address validator, uint256 performanceScore) external
onlyRole(ORACLE_MANAGER_ROLE) {
        require(validators[validator].exists, "Validator does not exist");
        validators[validator].performanceScore = performanceScore;
    }
    // Additional safeguard against unauthorized access or reassignments
    modifier onlyMultisigOrOracleCoordinator() {
        require(msg.sender == oracleManager, "Not authorized to assign ORACLE_MANAGER_ROLE");
        _;
    }
    // Optional: Option to revoke Oracle Manager Role (only by DAO or trusted entity)
    function revokeOracleManagerRole(address manager) external onlyRole(DEFAULT_ADMIN_ROLE) {
        _revokeRole(ORACLE_MANAGER_ROLE, manager);
    }
}
```

ValidatorManager.sol

16-Title: Missing Reentrancy Guard in State-Changing + External Call Functions

Risk Rating: High

Summary:

The functions rebalanceWithdrawal() and closeRebalanceRequests() modify the contract state and interact with external contracts. Without a nonReentrant modifier or proper application of the **Checks-Effects-Interactions** pattern, these functions are vulnerable to reentrancy attacks. An attacker could call these functions recursively before the state is fully updated, leading to unintended behaviors.

Proof of Concept (PoC):

```
function rebalanceWithdrawal() external {
 // State update
 withdrawalQueued = false;
 // External call (vulnerable to reentrancy)
 stakingManager.processValidatorWithdrawals();
}
function closeRebalanceRequests() external {
 // State update
 rebalanceRequestsActive = false;
 // External call (vulnerable to reentrancy)
 stakingManager.processCloseRequests();
}
```

An attacker could exploit these functions by calling them repeatedly, leading to **unexpected withdrawals** or **corruption of state**.

Impact on the Project:

Reentrancy attacks in withdrawal and rebalance logic could lead to **double withdrawals**, **stuck transactions**, or **loss of funds**. If an attacker can hijack the external calls, they may trigger a **denial-of-service (DoS)** scenario, blocking the contract from executing critical functions. This could cause substantial user losses, disrupt operations, and **damage the protocol's reputation**.

Recommendation or Fix:

Apply the nonReentrant modifier to these functions to prevent reentrancy:

Alternatively, **follow the Checks-Effects-Interactions pattern** by ensuring that all state changes occur before making any external calls.

ReentrancyGuard was added in the contract, and the nonReentrant modifier is applied to the rebalanceWithdrawal and closeRebalanceRequests functions.

```
function rebalanceWithdrawal(
 address stakingManager,
 address[] calldata validators,
 uint256[] calldata withdrawalAmounts
) external whenNotPaused nonReentrant onlyRole(MANAGER_ROLE) {
 // function implementation
}
function closeRebalanceRequests(
 address stakingManager,
 address[] calldata validators
) external whenNotPaused nonReentrant onlyRole(MANAGER_ROLE) {
 // function implementation
}
```

ValidatorManager.sol

17-Title: Reentrancy Risk in deactivateValidator()

Risk Rating: High

Summary:

The deactivateValidator() function modifies the contract state and then emits an event. If the emitted event triggers external logic (e.g., calling back into the contract), this introduces a reentrancy risk. Since the state change is already done, re-entering the contract could lead to inconsistent states or unintended side effects.

Proof of Concept (PoC):

```
function deactivateValidator(uint256 validatorId) external {
```

```
 // State change

 validators[validatorId].isActive = false;

 // Event emitted after state change

 emit ValidatorDeactivated(validatorId);

}
```

If the emitted event leads to external callbacks that interact with the contract, this could cause reentrancy issues by allowing the contract to enter a state that should have been fully finalized before any external calls.

Impact on the Project:

The lack of proper reentrancy protection in deactivateValidator() could lead to unexpected behaviors such as **reentrancy attacks**, where the state is changed multiple times or external logic is executed in an inconsistent contract state. This could cause **validators to be deactivated or reactivated unexpectedly**, potentially affecting network security and staking operations. It could also **compromise the integrity** of validator management and lead to **financial losses** or **loss of control** over the staking ecosystem.

Recommendation or Fix:

- **Emit events after all state changes** to ensure that the contract's state is finalized before any external interactions take place.

```
function deactivateValidator(address validator) external whenNotPaused nonReentrant
onlyRole(MANAGER_ROLE) {

    _deactivateValidator(validator);

    emit ValidatorDeactivated(validator);

}

function _deactivateValidator(address validator) internal {

    validators[validator] = false;

}
```

/src/interfaces/IPauserRegistry.sol

18-Title: Missing Role-Based Access Control

Risk Rating: High

Summary:

Sensitive methods like pauseContract() and emergencyPauseAll() are not properly restricted with role-based access control (RBAC). This lack of enforcement makes the contract vulnerable to unauthorized users or malicious contracts invoking these functions arbitrarily, potentially pausing or resuming critical contract functions without proper oversight.

Proof of Concept (PoC):

```
function pauseContract() external {

    // Logic to pause contract functionality

    paused = true;

}

function emergencyPauseAll() external {

    // Logic to pause all operations

    emergencyPaused = true;

}
```

In this case, without any access control, any user could call these functions, which can lead to malicious activities, such as pausing the contract and halting token transfers or critical operations.

Impact on the Project:

The **lack of role-based access control** for critical functions like pausing the contract creates a major security vulnerability. Unauthorized users could invoke these functions to pause the contract, which could **freeze all activities**, including transfers, minting, or burning operations. This could lead to **disruption of service**, **loss of funds**, and **irreversible contract damage** in case the pause is triggered maliciously. This risk would significantly undermine the stability of the project, especially if critical contract operations depend on the ability to control pauses effectively.

Recommendation or Fix:

- Add role-based access control modifiers like `onlyAdmin`, `onlyPauser`, or similar to the contract functions that handle pausing

```
// Define roles for pauser, unpauser, and pause-all

bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

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

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

// Grant roles during initialization

_grantRole(DEFAULT_ADMIN_ROLE, admin);

_grantRole(PAUSER_ROLE, pauser);

_grantRole(UNPAUSER_ROLE, unpauser);

_grantRole(PAUSE_ALL_ROLE, pauseAll);

// Functions with role-based access control

// Pauses a specific contract - only accessible by PAUSER_ROLE

function pauseContract(address contractAddress) external onlyRole(PAUSER_ROLE) {

    require(_authorizedContracts.contains(contractAddress), "Contract not authorized");

    require(!isPaused[contractAddress], "Contract already paused");

    isPaused[contractAddress] = true;

    emit ContractPaused(contractAddress);

}

// Unpauses a specific contract - only accessible by UNPAUSER_ROLE

function unpauseContract(address contractAddress) external onlyRole(UNPAUSER_ROLE) {

    require(_authorizedContracts.contains(contractAddress), "Contract not authorized");

    require(isPaused[contractAddress], "Contract not paused");

    isPaused[contractAddress] = false;

    emit ContractUnpaused(contractAddress);

}

// Emergency function to pause all authorized contracts - only accessible by PAUSE_ALL_ROLE

function emergencyPauseAll() external onlyRole(PAUSE_ALL_ROLE) {

    uint256 length = _authorizedContracts.length();

    for (uint256 i = 0; i < length; i++) {

        address contractAddress = _authorizedContracts.at(i);

        if (!isPaused[contractAddress]) {
```

```
            isPaused[contractAddress] = true;
            emit ContractPaused(contractAddress);
        }
    }
}
// Authorizes a contract to be pausable - only accessible by DEFAULT_ADMIN_ROLE
function authorizeContract(address contractAddress) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(contractAddress != address(0), "Invalid contract address");
    require(!_authorizedContracts.contains(contractAddress), "Contract already authorized");
    _authorizedContracts.add(contractAddress);
    emit ContractAuthorized(contractAddress);
}
// Removes authorization from a contract - only accessible by DEFAULT_ADMIN_ROLE
function deauthorizeContract(address contractAddress) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_authorizedContracts.contains(contractAddress), "Contract not authorized");
    _authorizedContracts.remove(contractAddress);
    emit ContractDeauthorized(contractAddress);
}
```

/src/interfaces/IStakingAccountant.sol

19-Title: Missing Access Control on State-Changing Functions

Risk Rating: High

Summary:

Functions such as authorizeStakingManager, deauthorizeStakingManager, recordStake, and recordClaim are responsible for altering the contract's critical state. However, they are not protected by any access control mechanisms, meaning unauthorized users or contracts can manipulate these critical internal states without restriction.

Proof of Concept (PoC):

```
function authorizeStakingManager(address _manager) external {
```

```
    // Logic to authorize staking manager
    stakingManager = _manager;
}
function recordStake(address _staker, uint256 amount) external {
    // Logic to record a user's stake
    stakes[_staker] += amount;
}
function recordClaim(address _claimer, uint256 amount) external {
    // Logic to record a user's claim
    claims[_claimer] += amount;
}
```

These functions should be restricted to trusted parties, such as the owner or an admin, but no restrictions are currently in place.

Impact on the Project:

Without proper access control, **unauthorized actors** could exploit these functions to arbitrarily:

- **Authorize or deauthorize staking managers** — allowing rogue entities to control the staking process.
- **Manipulate stake records or claims** — enabling attackers to inflate their stakes or claims, leading to **unfair distributions**, **fund theft**, or other critical vulnerabilities that could destabilize the entire system. This lack of access control exposes the project to significant security risks and could **severely compromise the integrity of the staking process**.

Recommendation or Fix:

- Implement access control using modifiers such as `onlyOwner`, `onlyAdmin`, or `onlyAuthorizedManager` to ensure that only authorized addresses can perform these sensitive operations.

Authorization and Deauthorization of Staking Managers: The authorizeStakingManager and deauthorizeStakingManager functions are now restricted to only be accessible by authorized roles (e.g., the manager role) using the onlyRole modifier.

```
function test_AuthorizeStakingManager() public {
    vm.startPrank(manager);  // Only manager can call this
    vm.expectEmit(true, true, false, false);
```

```
    emit StakingManagerAuthorized(mockStakingManager, address(kHYPE));
    stakingAccountant.authorizeStakingManager(mockStakingManager, address(kHYPE));
    assertTrue(stakingAccountant.isAuthorizedManager(mockStakingManager));
    assertEq(stakingAccountant.getManagerToken(mockStakingManager), address(kHYPE));
    vm.stopPrank();
}
function test_DeauthorizeStakingManager() public {
    vm.startPrank(manager);  // Only manager can call this
    stakingAccountant.authorizeStakingManager(mockStakingManager, address(kHYPE));
    vm.expectEmit(true, false, false, false);
    emit StakingManagerDeauthorized(mockStakingManager);
    stakingAccountant.deauthorizeStakingManager(mockStakingManager);
    assertFalse(stakingAccountant.isAuthorizedManager(mockStakingManager));
    vm.stopPrank();
}
```

Test Setup: The manager role is granted the ability to authorize and deauthorize staking managers.

- **Test Logic**: The functions authorizeStakingManager and deauthorizeStakingManager are tested to ensure that only the manager can perform these actions, and unauthorized actors will be prevented from making changes.

Recording Stakes and Claims: Similarly, the functions recordStake and recordClaim are restricted to the authorized staking manager and cannot be called by unauthorized addresses.

```
function test_RecordStake() public {
 vm.startPrank(manager);
 stakingAccountant.authorizeStakingManager(mockStakingManager, address(kHYPE));
 vm.expectEmit(true, false, false, false);
 emit StakeRecorded(mockStakingManager, 1 ether);
 stakingAccountant.recordStake(mockStakingManager, 1 ether);
 vm.stopPrank();
```

```
}

function test_RecordClaim() public {

 vm.startPrank(manager);

 stakingAccountant.authorizeStakingManager(mockStakingManager, address(kHYPE));

 vm.expectEmit(true, false, false, false);

 emit ClaimRecorded(mockStakingManager, 0.5 ether);

 stakingAccountant.recordClaim(mockStakingManager, 0.5 ether);

 vm.stopPrank();

}
```

Test Logic: The tests verify that stakes and claims can only be recorded by an authorized manager, and unauthorized attempts are reverted.

/src/interfaces/IStakingAccountant.sol

20-Title: Unclear or Static Token Conversion Logic

Risk Rating: High

Summary:

The functions kHYPEToHYPE() and HYPEToKHYPE() in the contract are responsible for converting between two token types, but they do not indicate how the exchange rates are determined. This ambiguity suggests that the conversion rates may be static or hardcoded, which can cause the contract to lack flexibility in reflecting real-world token values.

Proof of Concept (PoC):

```
function kHYPEToHYPE(uint256 kHYPEAmount) public view returns (uint256) {

 return kHYPEAmount * conversionRate; // Static conversion rate, potentially outdated or hardcoded

}

function HYPEToKHYPE(uint256 HYPEAmount) public view returns (uint256) {

 return HYPEAmount / conversionRate; // Static conversion rate

}
```

These functions do not provide insight into where the conversion rate comes from, potentially leading to vulnerabilities or mismatches between the actual market value and the token exchange rate used in the contract.

Impact on the Project:

1. **Inflexibility in Reflecting Real-World Value**:
   - If the conversion rates are hardcoded or static, they may no longer reflect the real-world value of the tokens, especially in volatile market conditions. This could result in **mispricing** between kHYPE and HYPE tokens, leaving the system disconnected from the actual market dynamics.
2. **Potential Exploits via Arbitrage**:
   - Without dynamic pricing mechanisms, external actors could exploit discrepancies between the static conversion rate and the actual market rate through **arbitrage**. During periods of high volatility or market shifts, attackers could buy tokens at a lower rate and sell them at a higher rate, causing economic loss or instability within the system.
3. **Loss of User Trust and Potential Market Disruptions**:
   - If the token conversion logic is not aligned with market conditions, users may lose trust in the system's ability to correctly value assets, leading to **decreased adoption** or **loss of funds** due to discrepancies in expected and actual token values.

Recommendation or Fix:

- Implement a **dynamic pricing mechanism** that sources real-time conversion rates from an oracle service (e.g., Chainlink) or a custom OracleManager. This ensures that token conversions are always reflective of the current market value.

```
function test_kHYPEToHYPE_UsesDynamicRate() public {

 uint256 mockRate = 1200e18; // mock 1 kHYPE = 1.2 HYPE

 mockOracle.setPrice(mockRate);

 uint256 converted = stakingAccountant.kHYPEToHYPE(1 ether);

 assertEq(converted, 1.2 ether);

}
```