CATALYSIS

Core Smart Contracts

Security Assessment Report

Version: 2.0

March, 2026

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Catalysis components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Catalysis components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Catalysis components in scope.

Overview

Catalysis connects restaked capital, risk underwriting, and DeFi vaults into a single onchain coverage flow.

Restakers deposit assets such as ETH, BTC, and stablecoins into restaking protocols such as EigenLayer and Symbiotic. Catalysis Core aggregates this capital and programmatically allocates it to provide coverage for participating DeFi vaults.

Coverage is vault-specific, opt-in, and enforced entirely through smart contracts.

If a predefined loss event occurs, Catalysis Core enforces slashing on delegated restaked capital. The slashed funds are then routed to compensate depositors in the covered vault, up to the defined coverage amount.

Catalysis Core also manages reward distribution across participating restaked vaults.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Catalysis Core repository.

The scope of this time-boxed review was strictly limited to files at commit 4c24694.

The specific files in scope were:

- /contracts/Adapters/BaseAdapter.sol
- /contracts/Adapters/EigenAdapter.sol
- /contracts/Adapters/SymbioticAdapter.sol
- /contracts/ChainlinkPriceFeed.sol
- /contracts/RewardsManager.sol
- /contracts/SlashingManager.sol
- /contracts/SSPRouter.sol
- /contracts/StakeManager.sol
- /contracts/libraries/SortingLib.sol
- /interfaces/*

The fixes of the identified findings were assessed at the following PRs and in a final merged commit of baf56f8.

- PR 477
- PR 478
- PR 479
- PR 480
- PR 481
- PR 482
- PR 483
- PR 484
- PR 485
- PR 486
- PR 488
- PR 489
- PR 490
- PR 491
- PR 492
- PR 493
- PR 494
- PR 495
- PR 496
- PR 497
- PR 501
- PR 502

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 23 issues during this assessment. Categorised by their severity:

- Critical: 2 issues.
- High: 1 issue.
- Medium: 1 issue.
- Low: 8 issues.
- Informational: 11 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Catalysis components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
CSC-01	Precision Loss In Slashing Calculations Reduces Slashed Amounts	Critical	Resolved
CSC-02	Raw Token Amounts Aggregated In Module-Level Reward Split	Critical	Resolved
CSC-03	Replay Protection Failure Allows Duplicate Operations	High	Resolved
CSC-04	Insufficient Cover Pool Validation Risks Cross-Committee Vault Manipulation	Medium	Resolved
CSC-05	Chainlink Price Feed Lacks Staleness Validation Checks	Low	Resolved
CSC-06	Adapter Slashing Fails To Validate Execution	Low	Resolved
CSC-07	Committee Recreation Allows State Inconsistencies Between Contracts	Low	Resolved
CSC-08	CommitteeID Truncated In EigenAdapter	Low	Resolved
CSC-09	Direct Role Management Functions Bypass State Tracking And Event Emission	Low	Resolved
CSC-10	Duplicate Network Instance Keys Across Multiple Manager Contracts	Low	Resolved
CSC-11	Operation Ordering Creates Inconsistent Vault-Operator State	Low	Resolved
CSC-12	Operator Can Be Added To Nonexistent Committee	Low	Resolved
CSC-13	Avoidable Gas Overhead In Frequently Called Functions	Informational	Closed
CSC-14	Chainlink Oracle Assumes 8 Decimal Return Format	Informational	Resolved
CSC-15	Contracts Do Not Enforce Stake Consistency For Insurance Periods	Informational	Resolved
CSC-16	Contracts Do Not Implement UUPS Upgrade Pattern	Informational	Resolved
CSC-17	Missing Code Length Check In `setCoverPoolFactory()`	Informational	Resolved
CSC-18	Mixed Token Committees Distribute Slashing Risk Across All Vault Assets	Informational	Closed
CSC-19	Shared Vault Stake Tracking Relies On Offchain Listener Coordination	Informational	Resolved
CSC-20	Silent Failure When Adapter Not Configured During Vault Addition	Informational	Resolved
CSC-21	Slashing Manager Mixes Token Amounts In Variable	Informational	Resolved
CSC-22	Upgradeable Implementation Contracts Lack Constructor Initialiser Protection	Informational	Resolved
CSC-23	Miscellaneous General Comments	Informational	Resolved

CSC-01	Precision Loss In Slashing Calculations Reduces Slashed Amounts		
Assets	`contracts/SlashingManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The slashing calculation mechanism in `SlashingManager._calculateVaultSlashings()` suffers from precision loss that reduces the actual slashed amounts, potentially to zero.

The function performs a two-stage calculation where each stage involves integer division:

SlashingManager._calculateVaultSlashings()

```
// Stage 1: Calculate slash percentage
uint256 slashPercentage = slashAmountUSD > totalUSDStake ? 10000 : (slashAmountUSD * 10000) / totalUSDStake;

// Stage 2: Apply to each vault
vaultSlashAmount = (stake * slashPercentage) / 10000;
```

Each division operation truncates the result, and when the slash amount is small relative to the total stake, the first division can round to zero or significantly underestimate the percentage.

Consider a slash of $99 on a stake of $1,000,000:

Example 1: Small slash amount calculation

```
slashAmountUSD   = 99 USD = 9,900,000,000 (8 decimals)
totalUSDStake    = 1,000,000 USD = 100,000,000,000,000 (8 decimals)

slashPercentage  = (9,900,000,000 * 10,000) / 100,000,000,000,000
                 = 99,000,000,000,000 / 100,000,000,000,000
                 = 0 basis points
```

This results in zero slashing for any slash amount under $100 (ie. less than a ten thousandth of `totalUSDStake`).

For larger amounts, the precision loss can still be substantial. Consider a slash of $333.33.. on a stake of $1,111,111:

Example 2: Precision loss demonstration

```
slashAmountUSD   = 333.33.. USD = 33,333,333,333 (8 decimals)
totalUSDStake    = 1,111,111.. USD = 111,111,111,111,111 (8 decimals)

slashPercentage  = (33,333,333,333 * 10,000) / 111,111,111,111,111
                 = 333,333,333,330,000 / 111,111,111,111,111
                 = 2.999..
                 = 2 (in Solidity)
```

The calculated percentage of 2 basis points instead of 2.999.. results in approximately 33% of the intended slash amount being lost from each vault.

This issue has a high impact as slashing operations can fail to extract the intended financial sums, reducing the economic security provided by the insurance system.

This issue has a high likelihood as it occurs in normal operation whenever slash amounts are small relative to total stake.

Recommendations

Restructure the calculation to perform all multiplications before divisions to preserve precision. Calculate the total slash amount directly for each vault without the intermediate percentage step:

Calculate vault slash amount directly

```
vaultSlashAmount = (stake * slashAmountUSD) / totalUSDStake;
```

This performs a single division operation and preserves maximum precision throughout the calculation.

Resolution

The development team has addressed this issue by restructuring the slashing calculation in `SlashingManager._calculateVaultSlashings()` to eliminate the two-stage division that was causing precision loss.

The fixes can be found in PR 477.

CSC-02	Raw Token Amounts Aggregated In Module-Level Reward Split		
Assets	`contracts/SSPRouter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The reward distribution mechanism aggregates raw token amounts from vaults containing different assets when calculating the split between Symbiotic and EigenLayer adapters, treating heterogeneous tokens as fungible units when determining module-level allocation ratios.

The `SSPRouter.distributeRewards()` function calculates reward distribution between Symbiotic and EigenLayer adapters based on stake proportions. The stake calculation in `_calculateCommitteeStakeAllocation()` sums raw token balances from different vaults:

SSPRouter._calculateCommitteeStakeAllocation()

```
function _calculateCommitteeStakeAllocation(
    uint96 committeeId,
    address[] memory vaults,
    address symbioticAdapter,
    address eigenAdapter
) internal view returns (ISSPRouter.CommitteeStakeAllocation memory breakdown) {
    uint256 vaultsLength = vaults.length;
    for (uint256 i = 0; i < vaultsLength; i++) {
        address vault = vaults[i];
        uint256 vaultStake = committeeVaultTotalStake[committeeId][vault];  // @audit Raw token amount

        if (vaultStake > 0) {
            ISSPRouter.SSPModuleType moduleType = vaultToModule[vault];

            if (moduleType == ISSPRouter.SSPModuleType.SYMBIOTIC && symbioticAdapter != address(0)) {
                breakdown.totalStake += vaultStake;      // @audit Summing different tokens
                breakdown.symbioticStake += vaultStake;
            } else if (moduleType == ISSPRouter.SSPModuleType.EIGENLAYER && eigenAdapter != address(0)) {
                breakdown.totalStake += vaultStake;      // @audit Summing different tokens
                breakdown.eigenStake += vaultStake;
            }
        }
    }
}
```

The `committeeVaultTotalStake` mapping stores raw token amounts in native token decimals for each vault. When vaults contain different assets, these amounts represent entirely different values. According to the development team, committees group related tokens together (e.g., wETH and wstETH in one committee, USDC and USDT in another), which creates two distinct potential problems:

Problem 1: Different Decimal Precision

Correlated stablecoins may have different decimal representations:

- USDC uses 6 decimals: `1000000` represents 1 USDC
- USDT uses 6 decimals: `1000000` represents 1 USDT
- DAI uses 18 decimals: `1000000000000000000` represents 1 DAI

A committee with 1,000 USDC and 1,000 DAI would sum as `1000000 + 1000000000000000000`, allocating rewards as if DAI holders have 1 trillion times more stake than USDC holders, despite equal economic value.

Problem 2: Different Economic Values

Even with identical decimals, related tokens trade at different values. Wrapped ETH derivatives were mentioned by the development team as potentially being grouped together:

- wETH might have `1000000000000000000` (1 wETH, 18 decimals, ~$3,000)
- wstETH might have `950000000000000000` (0.95 wstETH, 18 decimals, ~$3,500)

Summing these treats wstETH as having less stake despite representing higher economic value. The calculation treats token unit counts as fungible when they are not:

SSPRouter reward distribution split

```
// Distribute to Symbiotic adapter
if (stakes.symbioticStake > 0 && symbioticAdapter != address(0)) {
    uint256 symbioticReward = (rewardAmount * stakes.symbioticStake) / stakes.totalStake;  // @audit Ratio based on mixed tokens
    // ... distribute reward
}

// Distribute to Eigen adapter
if (stakes.eigenStake > 0 && eigenAdapter != address(0)) {
    uint256 eigenReward = (rewardAmount * stakes.eigenStake) / stakes.totalStake;  // @audit Ratio based on mixed tokens
    // ... distribute reward
}
```

These module-level reward distribution ratios are calculated based on the summed raw token amounts, which do not represent comparable economic units.

Scope of the Issue

It is important to note that this issue specifically affects the Symbiotic vs EigenLayer split at the committee level. Other parts of the reward distribution system mitigate the issue though USD normalisation:

- `SymbioticAdapter._getStakeDataAndVaults()` converts vault stakes to USD values when distributing rewards across Symbiotic vaults
- `StakeManager._getOperatorStake()` uses Chainlink price feeds to return USD-normalised stake values for slashing calculations

However, the `SSPRouter._calculateCommitteeStakeAllocation()` function, which determines how rewards are split between the two adapters, operates on raw token amounts without USD conversion. This creates an inconsistency where rewards are correctly allocated within each adapter but incorrectly split between adapters when committees contain vaults spanning both modules.

This issue has a high impact as reward distribution represents the core economic incentive mechanism. When the module split is incorrect, operators with vaults in one adapter receive disproportionately large or small rewards relative to their economic contribution. While the per-vault distribution within each adapter is correct, the initial split between adapters can cause significant allocation errors based on token characteristics (decimals or exchange rates) rather than actual economic value.

This issue has a high likelihood if committees commonly contain vaults spanning both Symbiotic and EigenLayer modules with heterogeneous tokens. The development team's intended operational model groups related tokens within committees, and if these tokens are distributed across both modules, the issue will manifest in the module-level split calculation.

Recommendations

Modify `SSPRouter._calculateCommitteeStakeAllocation()` to use USD-normalised stake values instead of raw token amounts when calculating the split between Symbiotic and EigenLayer adapters. This aligns the module-level split with the existing USD normalisation used within adapter-level reward distribution.

Resolution

The development team has addressed this issue by implementing changes to use USD-normalised stake values for module-level reward distribution.

The implementation resolves economic value disparities between different tokens, ensuring that reward distribution reflects the economic contribution of operators regardless of the underlying token characteristics or module distribution.

The fixes can be found in PR 478.

CSC-03	Replay Protection Failure Allows Duplicate Operations		
Assets	`contracts/SlashingManager.sol` `contracts/RewardsManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `SlashingManager` and `RewardsManager` contracts attempt to prevent duplicate operations through a counter-based replay protection mechanism, but the implementation has flaws that mean it provides no protection against duplicate slashing or reward distribution operations.

In `SlashingManager.executeSlashing()`, the counter is incremented before generating the key used for duplicate detection:

SlashingManager.executeSlashing()

```
// Increment counter for unique key generation
slashingCounter++;

// Check if slashing has already been executed for this task instance
bytes32 slashingKey = keccak256(abi.encode(committeeId, operator, slashAmount, slashingCounter));
if (slashingExecuted[slashingKey]) revert SlashingAlreadyExecuted();
```

Because `slashingCounter` is incremented before the key is generated, each call to `executeSlashing()` produces a unique key regardless of whether the same slashing parameters are submitted multiple times. The check `if (slashingExecuted[slashingKey])` cannot revert because every invocation creates a new, previously unused key.

For example, three identical slashing requests would generate three different keys:

- Call 1: `keccak256(committeeId, operator, 100, 1)`
- Call 2: `keccak256(committeeId, operator, 100, 2)`
- Call 3: `keccak256(committeeId, operator, 100, 3)`

The `RewardsManager.distributeRewards()` function contains an identical flaw with `rewardDistributionCounter`.

This issue has a high impact as operators can be slashed multiple times for the same offence, and rewards can be distributed multiple times for the same action, causing financial loss to operators and depletion of reward pools.

This issue has a medium likelihood as the claim manager or any privileged role with access to these functions can submit duplicate operations.

Recommendations

Implement proper replay protection by having the claim manager provide a unique identifier for each slashing and reward operation. Modify the functions to accept an additional parameter such as `taskId` or `eventId` that is externally generated and guaranteed to be unique per operation.

Update `SlashingManager.executeSlashing()` to:

Proposed fix with unique task ID

```
function executeSlashing(
    uint96 committeeId,
    address operator,
    uint256 slashAmount,
    address claimManager,
    bytes32 taskId  // @audit Add unique identifier
)
    external
    onlyRole(CLAIM_MANAGER_ROLE)
    returns (address[] memory collateralTokens, uint256[] memory collateralAmounts)
{
    // Check if slashing has already been executed for this task
    bytes32 slashingKey = keccak256(abi.encode(committeeId, operator, taskId));
    if (slashingExecuted[slashingKey]) revert SlashingAlreadyExecuted();

    // ... rest of the function

    slashingExecuted[slashingKey] = true;
}
```

If the `slashingCounter` is incremented after the duplication check, this will improve the duplicate protection, but will still allow any duplicate slashing call that does not match the very last slashing processed. In the case of two events in quick succession, duplicate slashings could still be processed.

Apply the same pattern to `RewardsManager.distributeRewards()`.

Resolution

The development team has addressed this issue by implementing the recommended replay protection mechanism using unique external identifiers. The changes include:

- Modified both `SlashingManager.executeSlashing()` and `RewardsManager.distributeRewards()` functions to accept an additional `taskId` parameter that serves as a unique identifier for each operation.
- Changed the key calculation to `keccak256(abi.encode(committeeId, operator, taskId))`, which creates consistent keys for identical operations regardless of when they are submitted.

The fixes can be found in PR 480.

CSC-04	Insufficient Cover Pool Validation Risks Cross-Committee Vault Manipulation		
Assets	`contracts/StakeManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `StakeManager.addCommitteeVaults()` function validates that the caller is a registered cover pool but does not verify that the caller is the appropriate cover pool for the specified committee. This allows any cover pool to add vaults to any committee.

The validation in `_checkCoverPool()` only confirms that `msg.sender` exists in the cover pool registry:

StakeManager cover pool validation

```
function _checkCoverPool() internal view {
    if (coverPoolFactory == address(0)) revert ZeroAddress();
    if (!ICoverPoolFactory(coverPoolFactory).coverPoolExists(msg.sender)) {
        revert NotCoverPool();
    }
}

function addCommitteeVaults(uint96 committeeId, address[] calldata vaults) external {
    _checkCoverPool();  // @audit Only checks caller is a cover pool
    // ... snip ...
    // No verification that msg.sender is the correct cover pool for committeeId
}
```

This design assumes that all cover pool contracts are fully secure and will not malfunction or be compromised. If a cover pool contract contains vulnerabilities that allow an attacker to call arbitrary functions, or if a cover pool implementation has logic errors, an attacker can manipulate vault configurations across multiple committees.

The protocol relies on a many-to-many relationship where multiple cover pools may interact with the staking infrastructure. Without proper authentication of which cover pool owns which committee, the security boundary between different committees is weakened.

This issue has a high impact as unauthorised vault additions would disrupt committee operations, affect slashing and reward distribution calculations, and potentially allow an attacker to redirect stake or manipulate committee state.

This issue has a low likelihood as it requires either a vulnerability in a cover pool contract or a compromised cover pool deployment. The assumption is that cover pools are expected to be secure, but this creates a single point of failure where any weakness in any cover pool affects all committees.

Recommendations

Implement a mapping that associates each committee with its authorised cover pool, and validate this relationship in `addCommitteeVaults()`:

Proposed committee-specific validation

```
mapping(uint96 => address) public committeeToAuthorisedCoverPool;

function _checkCoverPoolForCommittee(uint96 committeeId) internal view {
    if (coverPoolFactory == address(0)) revert ZeroAddress();
    if (!ICoverPoolFactory(coverPoolFactory).coverPoolExists(msg.sender)) {
        revert NotCoverPool();
    }
    if (committeeToAuthorisedCoverPool[committeeId] != msg.sender) {
        revert UnauthorisedCoverPool();
    }
}

function addCommitteeVaults(uint96 committeeId, address[] calldata vaults) external {
    _checkCoverPoolForCommittee(committeeId);
    // ... rest of function
}
```

Alternatively, if all committees should only be accessible from a single designated cover pool contract, enforce this in the contract initialisation and validate that only that specific cover pool can call committee modification functions.

Resolution

The development team has addressed this issue by implementing committee-specific cover pool validation. The changes include:

- Added a mapping `_committeeToAuthorizedCoverPool` that associates each committee with its authorised cover pool address, establishing ownership boundaries between committees and cover pools.
- Modified the `addCommitteeVaults()` function to include validation that `msg.sender` matches the authorised cover pool for the specific committee using `if (msg.sender != coverPool) revert NotCoverPool()`.
- Updated the committee creation process to set the committee-to-cover-pool association using `_committeeToAuthorizedCoverPool[committeeId] = coverPool`, so that each committee is linked to its authorised cover pool at creation time.

The fixes can be found in PR 479.

CSC-05	Chainlink Price Feed Lacks Staleness Validation Checks		
Assets	`contracts/ChainlinkPriceFeed.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `ChainlinkPriceFeed` contract retrieves price data from Chainlink oracles without verifying that the data is current, which can result in stale price information being used for USD value calculations.

The `getUSDValue()` function calls `latestRoundData()` but only uses the price value whilst discarding the other return parameters that indicate data freshness:

ChainlinkPriceFeed.getUSDValue()

```
function getUSDValue(address token, uint256 amount) external view onlyRegisteredToken(token) returns (uint256) {
    (, int256 price,,,) = AggregatorV3Interface(_priceFeeds[token]).latestRoundData();

    // Basic validation
    if (price <= 0) revert InvalidPrice(price);

    uint8 tokenDecimals = IERC20Metadata(token).decimals();

    // Calculate USD value with proper decimal handling
    return (amount * uint256(price)) / (10 ** tokenDecimals);
}
```

The `latestRoundData()` function returns five values:

AggregatorV3Interface.latestRoundData()

```
function latestRoundData() external view returns (
    uint80 roundId,
    int256 answer,
    uint256 startedAt,
    uint256 updatedAt,
    uint80 answeredInRound
);
```

The `updatedAt` timestamp indicates when the price was last updated. Without checking this value against the current block timestamp, the contract accepts price data regardless of age. During oracle outages, network disruptions, or periods when price updates are delayed, the contract continues to use outdated prices.

Chainlink price feeds operate with different heartbeat intervals depending on the asset and network, typically ranging from 1 to 24 hours. Prices older than the feed's heartbeat interval should be considered stale and rejected to prevent the use of outdated market information.

Stale prices can lead to incorrect USD value calculations used in slashing operations and reward distributions. If an asset's price has changed but the oracle has not updated, operators may be slashed for incorrect amounts, or rewards may be distributed based on outdated valuations.

This issue has a medium impact as incorrect price data affects financial calculations that determine slashing amounts and reward distributions. However, the impact is mitigated by the fact that Chainlink oracles are generally reliable and have built-in redundancy, and price deviations during staleness periods are typically limited by the deviation thresholds that trigger updates. The financial impact is further constrained by the protocol's operational parameters and the short duration of typical staleness events.

This issue has a low likelihood as it requires oracle feeds to become stale, which can occur during network congestion, oracle infrastructure issues, or periods when price deviation thresholds are not met to trigger updates. While Chainlink feeds are generally reliable, staleness validation is a standard security practice for oracle integrations.

Recommendations

Implement staleness validation by checking the `updatedAt` timestamp and reverting if the price data exceeds an acceptable age threshold:

ChainlinkPriceFeed.getUSDValue()

```
function getUSDValue(address token, uint256 amount) external view onlyRegisteredToken(token) returns (uint256) {
    (, int256 price,, uint256 updatedAt,) = AggregatorV3Interface(_priceFeeds[token]).latestRoundData();

    // Validate price is positive
    if (price <= 0) revert InvalidPrice(price);

    // Validate price is not stale
    if (block.timestamp - updatedAt > stalenessThreshold[token]) {
        revert StalePriceData(token, updatedAt);
    }

    uint8 tokenDecimals = IERC20Metadata(token).decimals();
    return (amount * uint256(price)) / (10 ** tokenDecimals);
}
```

Implement configurable staleness thresholds per token to accommodate different heartbeat intervals for different price feeds. This allows the contract to enforce appropriate freshness requirements based on each feed's update frequency.

Resolution

The development team has addressed this issue by implementing staleness validation with configurable thresholds. The changes include:

- Added a mapping to store different staleness thresholds for each token, allowing customisation based on the specific heartbeat intervals of different Chainlink price feeds.
- Modified the `getUSDValue()` function to capture the `updatedAt` timestamp from `latestRoundData()` and validate it against the current block timestamp.

The fixes can be found in PR 486.

CSC-06	Adapter Slashing Fails To Validate Execution		
Assets	`contracts/Adapters/EigenAdapter.sol` `contracts/Adapters/SymbioticAdapter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `executeSlashing` function in both `EigenAdapter.sol` and `SymbioticAdapter.sol` returns the requested slash amounts to the caller instead of the actual amounts that are slashed by the underlying protocols.

According to the interface definition comments, the array `collateralAmounts` should return "an Array of amounts slashed for each collateral token". However, the `collateralAmounts` value is derived directly from the `vaultSlashings` calldata. This means that the value that is returned is not necessarily the amount that has been slashed.

This may result in improper slashing under scenarios such as race conditions, where slashing is executed on the same operator simultaneously either through Catalysis or directly from the integrated protocol.

In addition, EigenLayer takes percentage values as the slashing amount parameters rather than direct token amounts, increasing the likelihood of perceived successful execution of the requested slash amount to Catalysis.

The definition of the `collateralAmounts` array specifies that this should be the amount slashed, and not the requested amount.

IBaseAdapter.executeSlashing()

```
/**
 * @notice Executes slashing for an operator
 * @param committeeId The committee ID
 * @param operator The address of the operator to slash
 * @param vaultSlashings Array of vault slashing information
 * @param totalSlashAmount The total slash amount across all vaults
 * @param claimManager The address of the claim manager who will receive the slashed funds
 * @return collateralTokens Array of collateral token addresses that were slashed
 * @return collateralAmounts Array of amounts slashed for each collateral token
 */
```

Within `EigenAdapter`, `executeSlashing()` calls the internal function `_executeSlashing()` which has no return value. The `collateralAmounts` return values are derived directly from the `vaultSlashings` argument.

EigenAdapter.executeSlashing()

```
    _executeSlashing(operator, vaultSlashings, committeeId, operator);

    // Collect collateral tokens (underlying tokens) and amounts
    collateralTokens = new address[](vaultSlashingsLength);
    collateralAmounts = new uint256[](vaultSlashingsLength);

    for (uint256 i = 0; i < vaultSlashingsLength; i++) {
        collateralTokens[i] = address(IStrategy(vaultSlashings[i].vault).underlyingToken());
        collateralAmounts[i] = vaultSlashings[i].slashAmount;
    }

    emit SlashingExecuted(operator, totalSlashAmount, committeeId);
}
```

`_executeSlashing()` retrieves the actual slash amount from the protocol. It does not return this information to the calling function, however, although it is emitted in an event.

EigenAdapter._executeSlashing()

```
function _executeSlashing(
        address eigenOperator,
        VaultSlashing[] memory vaultSlashings,
        uint96 committeeId,
        address operator
    ) internal {

        //...//

        (bool success, bytes memory returnData) = address(allocationManager).call(callData);

        //...//

        (uint256 slashId, uint256[] memory slashedAmounts) = abi.decode(returnData, (uint256, uint256[]));

        //...//

        emit OperatorSlashed(
            operator, eigenOperator, committeeId, vaultSlashingsLength, totalSlashedAmount, slashId, slashedAmounts
        );
    }
}
```

This issue has a low impact as the discrepancy between requested and actual slashed amounts primarily affects accounting accuracy and event reporting rather than causing direct financial loss. However, there is potential for confusion in monitoring systems if actual slashed amounts differ from reported amounts.

This issue has a low likelihood as it requires specific conditions where the actual slashed amount differs from the requested amount, such as concurrent slashing operations, insufficient operator stake, or protocol limitations on slashing percentages.

Recommendations

Utilise the values returned from the integrated protocol slashing execution message instead of the function input argument to derive the slash amount.

Resolution

The development team has addressed this issue by implementing the recommended solution to return actual slashed amounts instead of requested amounts. The changes include:

- Modified the `_executeSlashing()` function in both `EigenAdapter` and `SymbioticAdapter` to return the actual slashed amounts received from the underlying protocols.
- Updated the `executeSlashing()` function to utilise the values returned from the integrated protocol slashing execution instead of deriving amounts from the input arguments.

The fixes can be found in PR 485.

CSC-07	Committee Recreation Allows State Inconsistencies Between Contracts		
Assets	contracts/StakeManager.sol contracts/SSPRouter.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `StakeManager.createCommittee()` function does not prevent duplicate committee creation, which can lead to state inconsistencies between `StakeManager` and `SSPRouter`.

When `createCommittee()` is called, it adds the committee ID to the `_committeeIds` set and forwards the call to `SSPRouter`:

StakeManager.createCommittee()

```
function createCommittee(uint96 committeeId, uint256 maxStakeForCommittee) external onlyRole(POLICY_MANAGER_ROLE) {
    if (committeeId == 0) revert InvalidCommitteeId();
    if (router == address(0)) revert ZeroAddress();
    if (maxStakeForCommittee == 0) revert InvalidAmount();

    _committeeIds.add(uint256(committeeId));  // @audit Returns false if already exists, does not revert

    ISSPRouter(router).createCommittee(committeeId, maxStakeForCommittee);

    emit CommitteeCreated(committeeId, maxStakeForCommittee);
}
```

The `add()` function from `EnumerableSet` returns `false` if the value already exists rather than reverting. If a committee ID already exists, calling `createCommittee()` again will update the `maxStakeForCommittee` value in `SSPRouter.committeeMaxStake[committeeId]` but emit a misleading `CommitteeCreated` event.

Additionally, when vaults are added to committees through `SSPRouter.addCommitteeVaults()`, Symbiotic vaults receive the `maxStakeForCommittee` value via `ISymbioticAdapter.setMaxNetworkLimit()`. If the committee is recreated with a different maximum stake value, the call to `createCommittee()` does not update the network limits on existing Symbiotic vaults, creating a mismatch between the value stored in `SSPRouter` and the actual network limits configured in the Symbiotic protocol.

This issue has a low impact as it causes inconsistent state representation and accounting issues that may affect offchain monitoring systems, but does not directly compromise funds or core protocol functionality.

This issue has a low likelihood as it requires the privileged `POLICY_MANAGER_ROLE` to call `createCommittee()` with a duplicate committee ID, which should be prevented by correct operational procedures.

Recommendations

Add explicit validation to prevent duplicate committee creation:

StakeManager.createCommittee()

```
function createCommittee(uint96 committeeId, uint256 maxStakeForCommittee) external onlyRole(POLICY_MANAGER_ROLE) {
    if (committeeId == 0) revert InvalidCommitteeId();
    if (router == address(0)) revert ZeroAddress();
    if (maxStakeForCommittee == 0) revert InvalidAmount();

    // Revert if committee already exists
    if (!_committeeIds.add(uint256(committeeId))) {
        revert CommitteeAlreadyExists();
    }

    ISSPRouter(router).createCommittee(committeeId, maxStakeForCommittee);

    emit CommitteeCreated(committeeId, maxStakeForCommittee);
}
```

If updating committee parameters is a desired feature, create a dedicated `updateCommitteeMaxStake()` function that properly updates all affected contracts including adapter configurations.

Resolution

The development team has addressed this issue by implementing validation to prevent duplicate committee creation. Explicit validation was added in `createCommittee()` using the return value of `_committeeIds.add(uint256(committeeId))` to detect duplicate committee IDs.

The fixes can be found in PR 481.

CSC-08	CommitteeID Truncated In EigenAdapter		
Assets	`contracts/Adapters/EigenAdapter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The Catalysis core solution uses `uint96 committeeId` as the key to reference individual committees, however as part of the EigenLayer integration requirements it casts this value to a `uint32` as the unique ID to represent the operator. This type of conversion truncates the original value resulting in a risk of collision for `committeeId` values outside of the `uint32` range (0-4,294,967,295).

As an example, the following unique `committeeId` values would truncate to the value `10` when cast to `uint32` and result in colliding operator IDs when interacting with the EigenAdapter:

- 10
- 4294967306
- 8589934602

EigenAdapter.createRedistributingOperatorSet()

```
function createRedistributingOperatorSet(uint96 committeeId, address[] calldata strategies, address claimManager)
        external
        onlyRole(SSP_ROUTER_ROLE)
    {

...

        IAllocationManagerTypes.CreateSetParamsV2 memory params = IAllocationManagerTypes.CreateSetParamsV2({
            operatorSetId: uint32(committeeId),
            strategies: eigenStrategies,
            slasher: address(this)
        });
```

Observe the above code in EigenAdapter demonstrating the conversion of the `committeeId` value to `uint32`. This conversion has been identified in the following functions:

- `createRedistributingOperatorSet()`
- `addStrategiesToOperatorSet()`
- `_distributeRewards()`
- `_executeSlashing()`

The likelihood of this finding is low as creation of the `committeeId` value is not handled within the Catalysis core scripts, and it is not known if it is possible to intentionally generate a `committeeId` value that would collide with existing EigenLayer operator ID.

Recommendations

Enforce a hard bound where committees are created or where EigenAdapter is entered to revert if `committeeId > type(uint32).max`.

Resolution

The development team has addressed this issue by adding validation in the committee creation process to ensure that committee IDs do not exceed `type(uint32).max` (4,294,967,295).

The fixes can be found in PR 482.

CSC-09	Direct Role Management Functions Bypass State Tracking And Event Emission		
Assets	`contracts/SSPRouter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `SSPRouter` contract maintains storage variables tracking manager addresses alongside their corresponding access control roles, but the inherited `AccessControlUpgradeable` functions allow role modifications that bypass this tracking mechanism.

The `setManagerAddresses()` function manages both the roles and storage variables:

SSPRouter.setManagerAddresses()

```
function setManagerAddresses(address _stakeManager, address _rewardsManager, address _slashingManager)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    // ... snip ...

    _revokeRole(STAKE_MANAGER_ROLE, stakeManager);
    _revokeRole(REWARDS_MANAGER_ROLE, rewardsManager);
    _revokeRole(SLASHING_MANAGER_ROLE, slashingManager);

    // Set new addresses
    stakeManager = _stakeManager;
    rewardsManager = _rewardsManager;
    slashingManager = _slashingManager;

    // Grant new roles
    _grantRole(STAKE_MANAGER_ROLE, _stakeManager);
    _grantRole(REWARDS_MANAGER_ROLE, _rewardsManager);
    _grantRole(SLASHING_MANAGER_ROLE, _slashingManager);

    emit ManagerAddressesSet(_stakeManager, _rewardsManager, _slashingManager);
}
```

However, the public `grantRole()`, `revokeRole()`, and `renounceRole()` functions inherited from `AccessControlUpgradeable` remain accessible to addresses with `DEFAULT_ADMIN_ROLE`. These functions can modify the roles without updating the storage variables `stakeManager`, `rewardsManager`, and `slashingManager`, and without emitting the `ManagerAddressesSet` event.

This creates a situation where the storage variables and roles can become desynchronised, with the storage variables pointing to addresses that no longer have the required roles, or addresses having roles without being recorded in storage.

This issue has a low impact as it creates state inconsistencies that may affect offchain monitoring and administrative processes, but does not directly compromise the core protocol functionality since access control checks use the role system rather than the storage variables.

This issue has a low likelihood as it requires an administrator with `DEFAULT_ADMIN_ROLE` to use the inherited role management functions instead of the provided `setManagerAddresses()` function, which should be prevented by correct operational procedures.

Recommendations

Override the inherited role management functions to prevent direct role modifications for manager roles:

Proposed override implementation

```
function grantRole(bytes32 role, address account) public virtual override onlyRole(getRoleAdmin(role)) {
    if (role == STAKE_MANAGER_ROLE || role == REWARDS_MANAGER_ROLE || role == SLASHING_MANAGER_ROLE) {
        revert UseSetManagerAddresses();
    }
    super.grantRole(role, account);
}

function revokeRole(bytes32 role, address account) public virtual override onlyRole(getRoleAdmin(role)) {
    if (role == STAKE_MANAGER_ROLE || role == REWARDS_MANAGER_ROLE || role == SLASHING_MANAGER_ROLE) {
        revert UseSetManagerAddresses();
    }
    super.revokeRole(role, account);
}

function renounceRole(bytes32 role, address account) public virtual override {
    if (role == STAKE_MANAGER_ROLE || role == REWARDS_MANAGER_ROLE || role == SLASHING_MANAGER_ROLE) {
        revert UseSetManagerAddresses();
    }
    super.renounceRole(role, account);
}
```

This ensures all manager role changes flow through `setManagerAddresses()`, maintaining consistency between roles and storage variables.

Resolution

The development team has addressed this issue by overriding the inherited `grantRole()`, `revokeRole()`, and `renounceRole()` functions in `SSPRouter` to add validation for manager roles.

The fixes can be found in PR 484.

CSC-10	Duplicate Network Instance Keys Across Multiple Manager Contracts		
Assets	`contracts/SlashingManager.sol` `contracts/RewardsManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `SlashingManager` and `RewardsManager` contracts generate network instance keys using the same format, which COULD create collisions when the same committee ID and counter values are used across different managers or network instances.

Both contracts use identical key generation:

Network instance key generation

```
// SlashingManager.sol
bytes32 networkInstanceKey = keccak256(abi.encode(committeeId, slashingCounter));

// RewardsManager.sol
bytes32 networkInstanceKey = keccak256(abi.encode(committeeId, rewardDistributionCounter));
```

The keys are used to track accounting information in mappings:

Key usage in mappings

```
// SlashingManager
totalSlashingPerNetworkInstance[networkInstanceKey] += totalSlashAmount;

// RewardsManager
totalRewardsPerNetworkInstance[address(rewardToken)][networkInstanceKey] += netAmount;
```

According to the development team, these keys are used for accounting across multiple network instances of Eigen and Symbiotic. When multiple instances exist, or when both managers process operations for the same committee, the keys will collide:

- SlashingManager processes committee 1, counter reaches 5: `keccak256(1, 5)`
- RewardsManager processes committee 1, counter reaches 5: `keccak256(1, 5)`

These produce identical keys despite representing different operations in different contexts.

While the mappings are in separate contracts and do not directly interfere with each other, the collision affects offchain accounting systems that use these keys to aggregate data across contracts. If the accounting system queries both contracts and uses the keys as unique identifiers, it will incorrectly merge slashing and reward data.

This issue has a low impact as it affects offchain accounting and monitoring systems rather than onchain protocol functionality, but may cause confusion in financial reporting and auditing.

This issue has a low likelihood as it requires specific operational conditions where committee IDs and counter values align across different contracts, but this becomes more probable as the system scales and processes more operations.

Recommendations

Include a unique identifier in the key generation to differentiate between managers:

Proposed unique key generation

```
// SlashingManager
bytes32 networkInstanceKey = keccak256(abi.encode(address(this), committeeId, slashingCounter));

// RewardsManager
bytes32 networkInstanceKey = keccak256(abi.encode(address(this), committeeId, rewardDistributionCounter));
```

This ensures keys are globally unique across all contracts.

Resolution

The development team has addressed this issue by modifying the network instance key generation in both `SlashingManager` and `RewardsManager` to include the contract address as a unique identifier.

The fixes can be found in PR 483.

CSC-11	Operation Ordering Creates Inconsistent Vault-Operator State		
Assets	`contracts/StakeManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `StakeManager` contract maintains relationships between operators, committees, and vaults through multiple mappings. The implementation of `addCommitteeVaults()` and `_addOperatorToCommittee()` creates a dependency on operation ordering, where adding vaults before or after an operator produces different internal state.

When `_addOperatorToCommittee()` is called, it retrieves all vaults for the committee and updates the operator-vault relationships:

StakeManager._addOperatorToCommittee()

```
function _addOperatorToCommittee(address operator, uint96 committeeId) internal {
    address[] memory vaults = _committeeVaults[committeeId].values();  // @audit Gets current vaults
    uint256 vaultsLength = vaults.length;

    _operatorCommitteeId[operator] = committeeId;
    _committeeOperator[committeeId] = operator;

    for (uint256 i = 0; i < vaultsLength; i++) {
        _vaultToOperators[vaults[i]].add(operator);
        // ... snip ...
        _operators[operator].delegatedVaults.push(vaults[i]);
    }
}
```

However, when `addCommitteeVaults()` is called, it only updates `_committeeVaults` and does not update `_vaultToOperators` or `_operators[operator].delegatedVaults`:

StakeManager.addCommitteeVaults()

```
function addCommitteeVaults(uint96 committeeId, address[] calldata vaults) external {
    _checkCoverPool();
    // ... snip ...
    for (uint256 i = 0; i < vaultsLength; i++) {
        // ... validation ...
        _committeeVaults[committeeId].add(vault);  // @audit Only updates committee vaults
    }
    // @audit Does NOT update _vaultToOperators or _operators[operator].delegatedVaults
}
```

This creates two distinct execution paths:

Path A: Add operator, then add vaults

- `_vaultToOperators[vault]` does not contain operator
- `_operators[operator].delegatedVaults` does not contain vault

Path B: Add vaults, then add operator

- `_vaultToOperators[vault]` contains operator
- `_operators[operator].delegatedVaults` contains vault

The inconsistency primarily affects the `getOperatorInfo()` function, which returns operator information including delegated vaults. Offchain monitoring systems that rely on this function receive different data depending on the operation ordering, which may cause confusion or errors in accounting and dashboard displays.

This issue has a low impact as it affects the completeness of data returned by view functions used for monitoring and accounting, but does not compromise the core protocol functionality of staking, slashing, or rewards.

This issue has a low likelihood as typical operational procedures would establish committees fully before assigning operators, but the protocol does not enforce this ordering.

Recommendations

Update `addCommitteeVaults()` to synchronise the operator-vault relationships when a committee already has an assigned operator:

Proposed fix

```
function addCommitteeVaults(uint96 committeeId, address[] calldata vaults) external {
    _checkCoverPool();
    // ... existing validation ...

    // Get operator if one is assigned to this committee
    address operator = _committeeOperator[committeeId];

    for (uint256 i = 0; i < vaultsLength; i++) {
        address vault = vaults[i];
        // ... existing validation ...

        _committeeVaults[committeeId].add(vault);

        // Update operator-vault relationships if operator exists
        if (operator != address(0)) {
            _vaultToOperators[vault].add(operator);
            _operators[operator].delegatedVaults.push(vault);
        }
    }

    // ... rest of function
}
```

Resolution

The development team has addressed this issue with an architectural solution. The changes include:

- Modified `addCommitteeVaults()` to require that an operator must be assigned to the committee before vaults can be added, enforcing a specific operational sequence through `if (operator == address(0)) revert OperatorNotInCommittee()`.
- Updated the function to maintain state consistency by calling `_vaultToOperators[vault].add(operator)` when adding vaults, ensuring that operator-vault relationships are properly established regardless of the previous state.

The fixes can be found in PR 497.

CSC-12	Operator Can Be Added To Nonexistent Committee		
Assets	`contracts/StakeManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `StakeManager.addOperatorToCommittee()` function does not verify that the specified committee exists before assigning an operator to it. This allows operators to be assigned to committee IDs that have never been created through `createCommittee()`.

The function validates the operator and committee ID but does not check the `_committeeIds` set:

StakeManager.addOperatorToCommittee()

```
function addOperatorToCommittee(address operator, uint96 committeeId) external onlyRole(POLICY_MANAGER_ROLE) {
    if (operator == address(0)) revert ZeroAddress();
    if (committeeId == 0) revert InvalidCommitteeId();  // @audit Only checks for zero, not existence
    address currentOperator = _committeeOperator[committeeId];
    if (currentOperator != address(0)) {
        revert CommitteeAlreadyHasOperator();
    }
    // ... snip ...
    _addOperatorToCommittee(operator, committeeId);

    emit OperatorAddedToCommittee(operator, committeeId);
}
```

When an operator is added to a nonexistent committee, the operator becomes associated with a committee ID that does not exist in `_committeeIds`. This creates inconsistencies where:

- `isOperatorInCommittee()` and `getOperatorInfo()` return that the operator is in the committee.
- `getAllCommitteeIds()` does not include the committee ID.
- The committee will have no vaults associated with it, since vaults can only be added through `addCommitteeVaults()` which requires a call to `SSPRouter.addCommitteeVaults()`, and that function validates that the committee exists.

This inconsistency primarily affects offchain monitoring systems that rely on the committee enumeration and operator information queries to track the state of the protocol.

This issue has a low impact as it creates data inconsistencies that affect monitoring and accounting systems, but does not directly compromise funds or enable attacks since committees without vaults cannot participate in staking operations.

This issue has a low likelihood as it requires the privileged `POLICY_MANAGER_ROLE` to add an operator to a committee ID that was never created, which should be prevented by correct operational procedures.

Recommendations

Add validation to ensure the committee exists before adding an operator:

Proposed fix

```
function addOperatorToCommittee(address operator, uint96 committeeId) external onlyRole(POLICY_MANAGER_ROLE) {
    if (operator == address(0)) revert ZeroAddress();
    if (committeeId == 0) revert InvalidCommitteeId();

    // Verify committee exists
    if (!_committeeIds.contains(uint256(committeeId))) {
        revert CommitteeDoesNotExist();
    }

    address currentOperator = _committeeOperator[committeeId];
    if (currentOperator != address(0)) {
        revert CommitteeAlreadyHasOperator();
    }
    // ... rest of function
}
```

Resolution

The development team has addressed this issue by implementing the recommended validation in the `addOperatorToCommittee()` function. Committee existence is now checked to verify that the committee was properly created before allowing operator assignment.

The fixes can be found in PR 489.

CSC-13	Avoidable Gas Overhead In Frequently Called Functions
Assets	`contracts/StakeManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `StakeManager` contract does not initialise the configuration values `chainlinkPriceFeed`, `router`, and `coverPoolFactory`. The contract design has numerous functions to validate that `router` is configured before proceeding, adding gas overhead to frequently executed operations.

Multiple functions check if `router` is set:

StakeManager

```
function setOperatorCommitteeVaultStake(address operator, uint96 committeeId, address vault, uint256 stake)
    external
    onlyRole(LISTENER_ROLE)
{
    // ... snip ...
    if (router == address(0)) revert ZeroAddress();  // @audit Gas overhead
    // ... rest of function
}

function createCommittee(uint96 committeeId, uint256 maxStakeForCommittee) external onlyRole(POLICY_MANAGER_ROLE) {
    if (committeeId == 0) revert InvalidCommitteeId();
    if (router == address(0)) revert ZeroAddress();  // @audit Gas overhead
    // ... rest of function
}

function addCommitteeVaults(uint96 committeeId, address[] calldata vaults) external {
    _checkCoverPool();
    // ... snip ...
    if (router == address(0)) revert ZeroAddress();  // @audit Gas overhead
    // ... rest of function
}
```

Each check adds a storage read and conditional branch to the function execution. Functions like `setOperatorCommitteeVaultStake()` are called by listeners that monitor external protocol state and may be invoked frequently, making the cumulative gas cost noticeable.

The setter pattern provides operational flexibility to change the router address after deployment, but this flexibility comes at the cost of increased gas consumption for every function call that uses the router. If the router address is not expected to change after initial configuration, this overhead provides no benefit.

Recommendations

Consider setting the `router` address in the `initialize()` function and altering the setter function to disallow invalid router addresses. This eliminates the need for validation checks in every function that uses the router:

StakeManager.initialize()

```
function initialize(
    address _router,
    address admin,
    address policyManager
) external initializer {
    if (_router == address(0)) revert ZeroAddress();
    router = _router;
    // ... rest of initialisation
}
```

Resolution

The development team has acknowledged this finding but has chosen to maintain the current architecture due to deployment constraints. Their response states:

> We understand the finding regarding the repeated router == address(0) checks and the resulting gas overhead in high-frequency functions like setOperatorCommitteeVaultStake, createCommittee, and addCommitteeVaults.
>
> However, the suggestion to assign the router within the StakeManager.initialize() function and eliminate these runtime checks cannot be implemented due to a fundamental circular dependency between the StakeManager and the SSPRouter contracts.
>
> This architectural constraint is defined as follows: The StakeManager relies on a valid router address being set. The SSPRouter requires the StakeManager address to be deployed and registered. Because both contracts are mutually dependent during deployment, it is impossible to set the router address deterministically within StakeManager.initialize() without disrupting the deployment sequence. Consequently, the router must be assigned in a distinct configuration step executed after both contracts have been deployed.

While alternative deployment strategies exist (such as using CREATE2 for address prediction or a deployment contract that handles both contracts in a single transaction), the development team has chosen to accept the gas overhead associated with the runtime validation checks in favour of maintaining their current deployment approach.

CSC-14	Chainlink Oracle Assumes 8 Decimal Return Format
Assets	`contracts/ChainlinkPriceFeed.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The Catalysis core solution utilises a Chainlink integration to get the price of a token in USD, however assumes that the token price is returned in 8 decimal format. This results in severely inflated price calculations for token/USD pairings that return a value in 18 decimal format, for example eth+/USD at contract address `0x66Ab900341C98de949EC31d11b3C4180743844F1`.

ChainlinkPriceFeed

```
function registerPriceFeed(address token, address aggregator) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (token == address(0) || aggregator == address(0)) revert ZeroAddress();
    _priceFeeds[token] = aggregator;
    emit PriceFeedRegistered(token, aggregator);
}

...

function getUSDValue(address token, uint256 amount) external view onlyRegisteredToken(token) returns (uint256) {
    (, int256 price,,,) = AggregatorV3Interface(_priceFeeds[token]).latestRoundData();

    // Basic validation
    if (price <= 0) revert InvalidPrice(price);

    uint8 tokenDecimals = IERC20Metadata(token).decimals();

    // Calculate USD value with proper decimal handling
    // price = USD value of 1 token in 8 decimals
    // amount has tokenDecimals
    // Divide by 10^tokenDecimals to convert amount into whole tokens,
    // final result will be in 8 decimal USD.
    return (amount * uint256(price)) / (10 ** tokenDecimals);
}
```

As noted in the code comments, it is assumed in calculations that price feeds are returned in 8 decimal format, however no validation is performed at time of registration of the price feed or within this function when the price value is retrieved.

This finding is informational as registration of price feeds can only be performed by the administrator.

Recommendations

Implement a check to confirm that the price feed returns the expected decimal format in the `registerPriceFeed()` function.

Resolution

The development team has addressed this issue by implementing the recommended decimal validation in the `registerPriceFeed()` function. The changes include:

- Added decimal format validation that retrieves the aggregator's decimal precision using `AggregatorV3Interface(aggregator).decimals()`.
- Implemented a check that compares the retrieved decimal count against the expected 8-decimal format required by the system's calculations.

The fixes can be found in PR 490.

CSC-15	Contracts Do Not Enforce Stake Consistency For Insurance Periods
Assets	contracts/SSPRouter.sol contracts/StakeManager.sol contracts/SlashingManager.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

The protocol relies on operational discipline to ensure reward and slashing calculations use stake values consistent with defined insurance periods, but does not enforce this alignment through onchain mechanisms such as snapshots or time gating.

The contracts store current vault stake values in `SSPRouter.committeeVaultTotalStake`, which are updated via `StakeManager.setOperatorCommitteeVaultStake()` (role: `POLICY_MANAGER_ROLE`) and forwarded to `SSPRouter.setOperatorCommitteeVaultStake()` (role: `STAKE_MANAGER_ROLE`). All reward distribution and slashing calculations query these live values:

SSPRouter and SlashingManager stake queries

```
// SSPRouter.sol - stake used for reward distribution
function _calculateCommitteeStakeAllocation(...) internal view returns (...) {
    for (uint256 i = 0; i < vaultsLength; i++) {
        address vault = vaults[i];
        uint256 vaultStake = committeeVaultTotalStake[committeeId][vault];  // @audit Live value, not snapshot
        // ... uses vaultStake for reward split calculations
    }
}

// SlashingManager.sol - stake used for slashing calculations
function _calculateVaultSlashings(...) internal view returns (...) {
    for (uint256 i = 0; i < allVaultsLength; i++) {
        uint256 stake = ISSPRouter(sspRouter).committeeVaultTotalStake(committeeId, allVaults[i]);  // @audit Live value
        // ... calculates slash amounts based on this stake
    }
}
```

Operational Context

Insurance periods define windows during which operators provide coverage. Reward and slashing amounts should be calculated based on stake committed during these periods. However, vault stakes can change during and after insurance periods through:

1. **Symbiotic**: Deposits can occur within the current epoch; withdrawals become claimable after the epoch ends
2. **EigenLayer**: Deposits are accepted until a DurationVault is locked; withdrawals become available after the unlock timestamp

According to the development team, insurance periods are intended to align with or be shorter than Symbiotic epochs and EigenLayer vault durations. This alignment provides operational protection, but the contracts themselves do not enforce it.

Where the Issue Manifests

The reliance on live stake values creates operational risks in several scenarios:

1. **Late deposits after insurance period start**: If relayers update stake values to reflect deposits that occurred after the insurance period began, rewards may be allocated to stake that was not committed for the full period.
2. **Withdrawals during insurance periods**: Users can initiate withdrawals that will be claimable after the epoch/unlock timestamp. If stake values are updated before the insurance period ends, slashing calculations may use reduced stakes that do not reflect the coverage actually provided.
3. **Committees remaining active after maturity**: Nothing prevents reward distribution or slashing operations on committees after their insurance periods have ended, when vault balances may have changed significantly.
4. **Misaligned timing between protocols**: If a committee contains both Symbiotic vaults (epoch-based) and EigenLayer vaults (unlock timestamp-based) with different timing parameters, stake changes in one protocol may not correspond to changes in the other, creating inconsistencies in calculations.

Safe Operational Path

The protocol does provide a safe path through careful operational discipline:

- Relayers maintain consistent stake snapshots aligned to insurance period boundaries
- Stake values are only updated at insurance period starts, not during the period
- Committees are deactivated or removed after their insurance periods end
- Insurance period timing is coordinated with Symbiotic epochs and EigenLayer vault durations

If these practices are followed, the live stake values effectively function as snapshots for each insurance period, and the development team have clearly stated their intention to follow these practices. However, the contracts do not enforce them and it would be possible to add greater support.

This issue is rated as informational as it represents a design choice that places operational responsibility on relayers and administrators rather than a protocol vulnerability. The contracts function correctly if operated according to intended procedures, but lack guardrails that would prevent incorrect usage.

Recommendations

Consider implementing onchain mechanisms to enforce stake consistency aligned with insurance periods. Several approaches could strengthen the protocol's robustness:

Snapshot-Based Approach:

Proposed snapshot-based approach

```
// Store stake snapshots per insurance period
mapping(uint96 => mapping(uint256 => mapping(address => uint256))) public committeeInsurancePeriodVaultStake;

function snapshotStakeForInsurancePeriod(uint96 committeeId, uint256 insurancePeriodId) external {
    // Lock stake values for this insurance period
    // Reward/slashing calculations reference this snapshot rather than live values
}
```

Time Gating Approach:

Proposed time gating approach

```
mapping(uint96 => uint256) public committeeMaturityTimestamp;

function distributeRewards(...) external {
    require(block.timestamp <= committeeMaturityTimestamp[committeeId], "Committee matured");
    // ... proceed with reward distribution
}
```

Insurance Period Tracking:

Proposed insurance period tracking

```
struct InsurancePeriod {
    uint256 startTimestamp;
    uint256 endTimestamp;
    bool active;
}

mapping(uint96 => InsurancePeriod) public committeeInsurancePeriods;

function setInsurancePeriod(uint96 committeeId, uint256 start, uint256 end) external {
    // Define period boundaries
    // Enforce that rewards/slashing only occur during active periods
}
```

These mechanisms would codify the intended operational model, reducing reliance on offchain discipline and providing clearer guarantees about when stake values are used for calculations.

Resolution

The development team has addressed this issue through the implementation of the Duration Vault mechanism. Their response states:

> Please note that this finding has already been resolved through the Duration Vault implementation PR #469. The protocol now ensures that stake consistency is enforced at the vault level, aligning with the policy (insurance) duration for both supported restaking systems: EigenLayer and Symbiotic.

The implementation includes:

- **Symbiotic Integration**: Stake eligibility is determined by the vault's epoch duration, with insurance policies structured to align with Symbiotic epochs. This ensures that stake included in reward and slashing calculations consists exclusively of funds locked for the entire policy period, excluding deposits and withdrawals occurring outside the relevant epoch boundary.
- **EigenLayer Integration**: The DurationVault mechanism enforces stake eligibility by directly verifying the lock duration from the DurationVault, ensuring that only stake locked for at least the full policy duration is incorporated into committee stake calculations.

This approach addresses the core concern by implementing onchain mechanisms that enforce stake consistency aligned with insurance periods.

CSC-16	Contracts Do Not Implement UUPS Upgrade Pattern
Assets	contracts/Adapters/BaseAdapter.sol contracts/Adapters/EigenAdapter.sol contracts/Adapters/SymbioticAdapter.sol contracts/ChainlinkPriceFeed.sol contracts/RewardsManager.sol contracts/SlashingManager.sol contracts/SSPRouter.sol contracts/StakeManager.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract implementations do not include the functionality required for UUPS (Universal Upgradeable Proxy Standard) upgrades, despite deployment scripts and documentation referring to "UUPS proxy" addresses in documentation comments.

The contracts are designed as upgradeable implementations with `initialize()` functions and inherit from OpenZeppelin's upgradeable contract variants. The development team have confirmed their intention to use the UUPS upgrade pattern.

DeployedContracts struct

```
struct DeployedContracts {
    address chainlinkPriceFeedProxy;
    address chainlinkPriceFeedImpl;
    address eigenAdapterProxy;       // @audit Comment says "UUPS proxy"
    address eigenAdapterImpl;
    // ... additional contracts
}
```

However, the implementation contracts do not inherit from `UUPSUpgradeable` and do not implement the `_authorizeUpgrade()` function required for UUPS upgrades. The UUPS pattern allows the implementation contract to control upgrade authorisation, whereas the deployed `ERC1967Proxy` is a basic proxy that delegates all calls to the implementation.

The `ERC1967Proxy` used in deployment supports the ERC1967 storage slot standard for proxies but does not enforce any particular upgrade mechanism. Contracts can be upgraded through this proxy using either:

- Transparent proxy pattern with a separate `ProxyAdmin` contract
- UUPS pattern where the implementation contains upgrade logic

Without `UUPSUpgradeable` inheritance, upgrades must be performed using the transparent proxy pattern.

This issue is rated as informational as it represents a discrepancy between documentation and implementation rather than a functional defect. The contracts remain upgradeable through alternative proxy patterns.

Recommendations

Consider either:

1. Update documentation to accurately reflect the upgrade pattern being used. If transparent proxies with `ProxyAdmin` contracts are intended, remove references to "UUPS proxy" from comments and documentation.
2. Implement the UUPS pattern by inheriting from `UUPSUpgradeable` and implementing `_authorizeUpgrade()` functions in all upgradeable contracts:

Proposed UUPS implementation

```
import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";

contract StakeManager is UUPSUpgradeable, ... {
    function _authorizeUpgrade(address newImplementation) internal override onlyRole(DEFAULT_ADMIN_ROLE) {}
}
```

This would enable self-contained upgrade authorisation within the implementation contracts as suggested by the deployment script comments.

Resolution

The development team has addressed this issue by implementing the recommended UUPS upgrade pattern across all upgradeable contracts.

The fixes can be found in PR 491.

CSC-17	Missing Code Length Check In `setCoverPoolFactory()`
Assets	`contracts/StakeManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `StakeManager.setCoverPoolFactory()` function validates that the provided address is not zero but does not verify that the address contains contract code. This is inconsistent with the other setter functions in the contract that perform both checks.

The three configuration setters show inconsistent validation patterns:

StakeManager configuration setters

```
function setChainlinkPriceFeed(address _chainlinkPriceFeed) external onlyRole(STAKE_MANAGER_CONFIG_ROLE) {
    if (_chainlinkPriceFeed == address(0)) revert ZeroAddress();
    if (_chainlinkPriceFeed.code.length == 0) revert InvalidChainlinkPriceFeed();  // @audit Has code check
    chainlinkPriceFeed = _chainlinkPriceFeed;
    emit ChainlinkPriceFeedSet(_chainlinkPriceFeed);
}

function setRouter(address _router) external onlyRole(STAKE_MANAGER_CONFIG_ROLE) {
    if (_router == address(0)) revert ZeroAddress();
    if (_router.code.length == 0) revert InvalidRouter();  // @audit Has code check
    router = _router;
    emit RouterSet(_router);
}

function setCoverPoolFactory(address _coverPoolFactory) external onlyRole(STAKE_MANAGER_CONFIG_ROLE) {
    if (_coverPoolFactory == address(0)) revert ZeroAddress();
    // @audit Missing code length check
    coverPoolFactory = _coverPoolFactory;
    emit CoverPoolFactorySet(_coverPoolFactory);
}
```

The code length check prevents accidental configuration errors where an externally owned account address or nonexistent contract address is set instead of a deployed contract. While this is a minor issue that would be caught during testing, including the check maintains consistency and provides defence in depth.

This issue is rated as informational as it represents inconsistent validation patterns rather than a security vulnerability, and the privileged role required to call this function should prevent accidental misconfiguration.

Recommendations

Consider adding a code length check to `setCoverPoolFactory()` for consistency with the other setter functions:

Proposed fix

```
function setCoverPoolFactory(address _coverPoolFactory) external onlyRole(STAKE_MANAGER_CONFIG_ROLE) {
    if (_coverPoolFactory == address(0)) revert ZeroAddress();
    if (_coverPoolFactory.code.length == 0) revert InvalidCoverPoolFactory();
    coverPoolFactory = _coverPoolFactory;
    emit CoverPoolFactorySet(_coverPoolFactory);
}
```

Resolution

The development team has addressed this issue by implementing the recommended code length check in the `setCoverPoolFactory()` function in `StakeManager`, matching the validation pattern used in `setChainlinkPriceFeed()` and `setRouter()`.

The fixes can be found in PR 492.

CSC-18	Mixed Token Committees Distribute Slashing Risk Across All Vault Assets
Assets	contracts/SlashingManager.sol contracts/StakeManager.sol
Status	**Closed:** See Resolution
Rating	Informational

Description

Committees containing vaults with different collateral tokens distribute slashing risk across all vault assets based on a uniform percentage, exposing vaults to losses from token-specific price movements affecting other tokens in the committee.

The slashing mechanism calculates a single slash percentage based on the total USD value of all committee stakes, then applies this percentage uniformly to all vaults regardless of their underlying collateral token. This design creates cross-collateral risk exposure within mixed-token committees.

The slashing calculation process operates as follows:

SlashingManager._calculateVaultSlashings()

```
// Get total USD stake across all vaults
uint256 totalUSDStake = IStakeManager(stakeManager).getOperatorCommitteeStake(operator, committeeId);

// ... //

// Calculate uniform slash percentage
uint256 slashPercentage = slashAmountUSD > totalUSDStake ? 10000 : (slashAmountUSD * 10000) / totalUSDStake;

// ... //

// Apply same percentage to all vaults
for (uint256 i = 0; i < allVaultsLength; i++) {
    uint256 stake = ISSPRouter(sspRouter).committeeVaultTotalStake(committeeId, allVaults[i]);
    if (stake > 0) {
        vaultSlashAmount = (stake * slashPercentage) / 10000;
        totalSlashAmount += vaultSlashAmount;
    }
}
```

The total USD stake aggregates values across all committee vaults, with each vault's token amount converted to USD at current prices:

StakeManager._getOperatorStake()

```
function _getOperatorStake(uint96 committeeId) internal view returns (uint256) {
    if (router == address(0)) revert ZeroAddress();
    address[] memory vaults = _committeeVaults[committeeId].values();
    if (vaults.length == 0) revert EmptyVaults();
    if (chainlinkPriceFeed == address(0)) revert ZeroAddress();

    uint256 vaultsLength = vaults.length;
    uint256 totalStakeUSD = 0;
    for (uint256 i = 0; i < vaultsLength; i++) {
        address vault = vaults[i];
        uint256 operatorStake = ISSPRouter(router).committeeVaultTotalStake(committeeId, vault);

        if (operatorStake > 0) {
            // Get the vault's underlying token based on module type
            address token = _getVaultToken(vault);
            if (token == address(0)) {
                // Revert if vault has unknown module type
                revert VaultModuleNotSet(vault);
            }

            // Convert stake to USD value using price feed
            uint256 usdValue = IChainlinkPriceFeed(chainlinkPriceFeed).getUSDValue(token, operatorStake);
            totalStakeUSD += usdValue;
        }
    }
    return totalStakeUSD;
}
```

Consider a committee with two vaults: 1% WETH (valued at $1,000) and 99% WstETH (valued at $99,000), totalling $100,000. If WstETH experiences an idiosyncratic price decline of 10% (dropping the WstETH vault value to $89,100), the total committee stake becomes $90,100.

For a fixed slash amount of $5,000 USD:

- **Before price decline**: Slash percentage = $5,000 / $100,000 = 5%
  - WETH vault slashed: 5% of 1 WETH tokens = 0.05 WETH
  - WstETH vault slashed: 5% of 99 WstETH tokens = 4.95 WstETH
- **After WstETH price decline**: Slash percentage = $5,000 / $90,100 = 5.55%
  - WETH vault slashed: 5.55% of 1 WETH tokens = 0.0555 WETH (11% increase)
  - WstETH vault slashed: 5.55% of 99 WstETH tokens = 5.49 WstETH (11% increase)

The WETH vault, despite maintaining its token quantity and USD value, experiences an 11% increase in token slashing purely due to WstETH's price movement. This occurs because the slash percentage increases when total committee USD value decreases, and this higher percentage applies uniformly to all token quantities.

This cross-collateral risk exposure is an inherent characteristic of the committee-based slashing design. It is highlighted here as a matter for consideration because of the potential risk of vault curators joining committees without a full appreciation of this risk, and any resultant reputational damage that could result.

Recommendations

Ensure comprehensive documentation clearly communicates the cross-collateral risk characteristics of mixed-token committees to vault operators and liquidity providers. Documentation should explicitly state that:

- Token-specific price movements affecting any committee member impact slashing percentages for all committee vaults.
- The magnitude of cross-collateral impact scales with the proportion of committee stake held in the price-volatile token.

The development team have expressed their intention to implement committee compositions that group tokens with correlated price behaviour. This will mitigate this issue to some degree.

Resolution

The development team has acknowledged this design characteristic and provided the following response:

> The proposed solution focuses on committee-specific vault configuration to ensure all tokens within a committee's vaults exhibit correlated price behavior. Implementing this at the smart contract level is complex. Instead, we can manage this during the vault addition process by providing appropriately correlated vaults for each committee in `StakeManager.addCommitteeVaults`.

CSC-19	Shared Vault Stake Tracking Relies On Offchain Listener Coordination
Assets	contracts/StakeManager.sol contracts/SSPRouter.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

Vaults can participate in multiple committees simultaneously, but their token balances are not partitioned onchain across committees, creating a dependency on offchain listener infrastructure to correctly attribute stake changes to specific committees following slashing events.

The protocol permits the same vault to be registered with multiple committees without enforcing balance segregation. Committee stake tracking occurs through the `committeeVaultTotalStake` mapping in `SSPRouter`, which maintains independent stake values per committee-vault pair.

Consequently, a vault with 200 tokens can be registered as providing 100 tokens to Committee A and 100 tokens to Committee B, with both allocations tracked independently in `committeeVaultTotalStake[A][vaultX]` and `committeeVaultTotalStake[B][vaultX]`.

Slashing operations execute against vault token balances without modifying the onchain stake tracking mappings. The protocol relies on the `LISTENER_ROLE` to observe slashing events and vault balance changes, then update committee stake allocations by calling `setOperatorCommitteeVaultStake()`:

StakeManager.setOperatorCommitteeVaultStake()

```
function setOperatorCommitteeVaultStake(address operator, uint96 committeeId, address vault, uint256 stake)
    external
    onlyRole(LISTENER_ROLE)
{
        // ... //
        if (_committeeOperator[committeeId] != operator) {
            revert OperatorNotInCommittee();
        }
        if (!_committeeVaults[committeeId].contains(vault)) {
            revert VaultNotInCommittee();
        }

        ISSPRouter(router).setOperatorCommitteeVaultStake(operator, committeeId, vault, stake);

        emit OperatorCommitteeVaultStakeSet(operator, committeeId, vault, stake);
}
```

This function permits arbitrary stake value assignment without validating against actual vault balances or coordinating with other committee allocations for the same vault.

Consider the following scenario illustrating the listener coordination requirement:

Initial State:

- Committee A: Vault X allocated 100 tokens
- Committee B: Vault X allocated 100 tokens, Vault Y allocated 100 tokens
- Actual Vault X balance: 200 tokens

- `committeeVaultTotalStake[A][X] = 100`
- `committeeVaultTotalStake[B][X] = 100`
- `committeeVaultTotalStake[B][Y] = 100`

Slashing Event: Committee A experiences a slashing event that removes 50 tokens from Vault X.

Post-Slashing Vault State:

- Actual Vault X balance: 150 tokens (reduced from 200)

Listener Behaviour Option 1 (Committee-Specific Attribution): The listener correctly attributes the 50-token reduction to Committee A only:

- `committeeVaultTotalStake[A][X] = 50`
- `committeeVaultTotalStake[B][X] = 100` (unchanged)
- `committeeVaultTotalStake[B][Y] = 100` (unchanged)

Listener Behaviour Option 2 (Proportional Distribution): The listener observes a 50-token reduction (25% of 200) and distributes this proportionally:

- `committeeVaultTotalStake[A][X] = 75` (25% reduction)
- `committeeVaultTotalStake[B][X] = 75` (25% reduction)
- `committeeVaultTotalStake[B][Y] = 100` (unchanged)

The protocol provides no onchain enforcement mechanism to ensure the listener implements Option 1 (correct committee-specific attribution) rather than Option 2 (incorrect proportional distribution) or other arbitrary allocation strategies. The listener must correctly correlate slashing events with their originating committees and update only the affected committee's stake allocation.

This architectural characteristic requires robust offchain infrastructure to maintain accurate committee stake tracking. The listener component must monitor slashing events, identify which committee triggered each event, query actual vault balances, and calculate appropriate per-committee stake updates whilst preserving allocations for unaffected committees.

This issue is rated as informational as the described behaviour represents a known architectural dependency on offchain listener infrastructure rather than an onchain vulnerability. It is highlighted here as a potential consideration for the design and review of the listener software.

Recommendations

Ensure the offchain listener implementation correctly handles shared vault scenarios by attributing slashing events to specific committees rather than distributing balance changes proportionally across all committees sharing the affected vault.

Resolution

The development team restructured the code to significantly reduce reliance on offchain listeners and instead utilise onchain value assessment in USD for vault accounting.

Checks were also added to require correct durations and stake valuations. As part of integrating these changes, a requirement was also added to add all vaults to a committee in a single transaction.

The fixes can be found in PR 497, PR 501 and PR 502.

CSC-20	Silent Failure When Adapter Not Configured During Vault Addition
Assets	`contracts/SSPRouter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `SSPRouter.addCommitteeVaults()` function silently skips adapter configuration when an adapter address is not set, rather than reverting or ensuring proper configuration.

After categorising vaults by module type, the function checks if adapters exist before calling them:

SSPRouter.addCommitteeVaults()

```
// Process EigenLayer vaults
if (eigenCount > 0) {
    address eigenAdapter = adapters[ISSPRouter.SSPModuleType.EIGENLAYER];
    if (eigenAdapter != address(0)) {  // @audit Silent skip if adapter not configured
        // ... configure vaults with adapter ...
        IEigenAdapter(eigenAdapter).addStrategiesToOperatorSet(committeeId, eigenStrategies);
    }
}

// Process Symbiotic vaults
if (symbioticCount > 0) {
    address symbioticAdapter = adapters[ISSPRouter.SSPModuleType.SYMBIOTIC];
    if (symbioticAdapter != address(0)) {  // @audit Silent skip if adapter not configured
        for (uint256 i = 0; i < symbioticCount; i++) {
            ISymbioticAdapter(symbioticAdapter).setMaxNetworkLimit(
                symbioticVaults[i], committeeId, maxStakeForCommittee
            );
        }
    }
}

emit CommitteeVaultsAdded(committeeId, vaults);
```

If an adapter is not configured, the function completes successfully and emits the `CommitteeVaultsAdded` event, but the vaults are not properly configured in the external protocols. The vaults are added to `committeeModuleVaults` mapping, but the corresponding protocol-specific setup is skipped.

Testing confirms that this scenario cannot occur during normal operation because `registerVaultModule()` requires an adapter to be set before a vault can be registered with a module type. However, the defensive checks suggest the code is attempting to handle this case, and the current implementation handles it by silently failing rather than reverting.

This design creates ambiguity about the intended behaviour. If adapters are guaranteed to exist due to the `registerVaultModule()` validation, these checks are unnecessary overhead. If there are scenarios where adapters might not be configured, the silent failure could cause operational issues.

This issue is rated as informational as it represents unclear defensive programming patterns rather than a functional issue under normal operation.

Recommendations

Consider removing the adapter checks entirely since `registerVaultModule()` ensures adapters exist before vaults can be registered:

Proposed fix - remove defensive checks

```
// Process EigenLayer vaults
if (eigenCount > 0) {
    address eigenAdapter = adapters[ISSPRouter.SSPModuleType.EIGENLAYER];
    // ... configure vaults with adapter ...
}
```

Alternatively, if defensive checks are desired for safety, revert when an adapter is not found:

Alternative fix - explicit revert

```
// Process EigenLayer vaults
if (eigenCount > 0) {
    address eigenAdapter = adapters[ISSPRouter.SSPModuleType.EIGENLAYER];
    if (eigenAdapter == address(0)) revert AdapterNotConfigured();
    // ... configure vaults with adapter ...
}
```

This makes the expected behaviour explicit and prevents silent failures.

Resolution

The development team has addressed this issue by implementing the recommended explicit revert behaviour. The changes include:

- Modified the adapter validation logic to use `==` comparison instead of `!=` to check for unconfigured adapters.
- Added explicit revert statements when an adapter address equals `address(0)`, preventing silent failures and ensuring that vault addition operations only complete when adapters are properly configured.

The fixes can be found in PR 493.

CSC-21	Slashing Manager Mixes Token Amounts In Variable
Assets	`contracts/SlashingManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_calculateVaultSlashings` function sums slash amounts across vaults containing different tokens into a single `totalSlashAmount`. This value is considered meaningless and at worst misleading as there is limited use to combining counts of tokens with different values.

Currently the variable is not used for any calculation other than a non-zero check that can be accomplished through other means.

SlashingManager._calculateVaultSlashings()

```
// ============ Internal Functions ============

function _calculateVaultSlashings(uint96 committeeId, address operator, uint256 slashAmountUSD)
    internal
    view
    returns (IBaseAdapter.VaultSlashing[] memory vaultSlashings, uint256 totalSlashAmount)
{

...

      if (stake > 0) {
            if (moduleType == ISSPRouter.SSPModuleType.SYMBIOTIC && slasherContract != address(0)) {
                // Symbiotic vault with valid slasher
                vaultSlashAmount = (stake * slashPercentage) / 10000;
                totalSlashAmount += vaultSlashAmount;
            } else if (moduleType == ISSPRouter.SSPModuleType.EIGENLAYER) {
                // EigenLayer vault - calculate slash amount (executed by EigenAdapter)
                vaultSlashAmount = (stake * slashPercentage) / 10000;
                totalSlashAmount += vaultSlashAmount;
```

Observe above, `totalSlashAmount` is the sum of vault tokens to be slashed, across vaults that may contain different collateral.

This finding is informational as Catalysis is enforcing through administration that only tokens and their derivatives are allowed in the same committee.

Recommendations

Remove the `totalSlashAmount` if there is no plan to utilise this value in future.

Resolution

The development team has addressed this issue by removing the problematic `totalSlashAmount` variable completely. The core slashing functionality is maintained through direct use of the provided slash amount parameter.

The fixes can be found in PR 494.

CSC-22	Upgradeable Implementation Contracts Lack Constructor Initialiser Protection
Assets	contracts/StakeManager.sol contracts/RewardsManager.sol contracts/SlashingManager.sol contracts/SSPRouter.sol contracts/ChainlinkPriceFeed.sol contracts/Adapters/EigenAdapter.sol contracts/Adapters/SymbioticAdapter.sol contracts/Adapters/BaseAdapter.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

The upgradeable implementation contracts do not include constructor protection to prevent uninitialised implementation contracts, creating a potential attack surface where malicious actors could initialise the implementation contracts directly.

All upgradeable contracts in the protocol inherit from OpenZeppelin's upgradeable base contracts and use the `initialize()` pattern with the `initializer` modifier. However, none of these contracts include a constructor with `_disableInitializers()` to prevent third parties from calling `initialize()` on the implementation contract itself.

StakeManager

```
contract StakeManager is IStakeManager, AccessControlUpgradeable {
    // ... state variables ...

    function initialize(address _admin, address _stakeManagerConfigRoleHolder, address _policyManager)
        external
        initializer
    {
        // ... initialisation logic ...
    }
}
```

Without a constructor calling `_disableInitializers()`, the implementation contract remains in an uninitialised state and can be initialised by anyone. Whilst initialising the implementation contract does not directly affect the proxy contracts that users interact with, it creates operational and security concerns, as well as potential reputational damage.

According to OpenZeppelin's documentation and security best practices, implementation contracts should include the following constructor:

Recommended constructor pattern

```
constructor() {
    _disableInitializers();
}
```

This pattern prevents the implementation contract from being initialised by calling `_disableInitializers()` in the constructor, which sets the initialisation status to the maximum value, effectively locking the implementation contract.

The same issue affects all eight upgradeable contracts in the protocol:

- `StakeManager`

- `RewardsManager`
- `SlashingManager`
- `SSPRouter`
- `ChainlinkPriceFeed`
- `EigenAdapter`
- `SymbioticAdapter`
- `BaseAdapter`

Recommendations

Add a constructor with `_disableInitializers()` to all upgradeable implementation contracts to prevent them from being initialised. Apply this pattern to all eight upgradeable contracts in the protocol.

Resolution

The development team has addressed this issue by adding constructors with `_disableInitializers()` to all upgradeable implementation contracts.

The fixes can be found in PR 495.

CSC-23	Miscellaneous General Comments
Assets	/*
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Incorrect Constructor Documentation In `SlashingManager`**

   Related Asset(s): contracts/SlashingManager.sol

   The `SlashingManager` contract contains a documentation comment indicating a constructor section, but the function is an `initialize()` function for an upgradeable contract pattern, not a constructor.

```
// ============ Constructor ============

/// @notice Constructor for SlashingManager
/// @param _admin The address that will be granted admin role
/// @param _slashingManagerConfigRole The address that will be granted the slashing manager config role
/// @param _stakeManager The address of the StakeManager contract
function initialize(address _admin, address _slashingManagerConfigRole, address _stakeManager)
    external
    initializer
{
    // ... implementation
}
```

   Update the documentation to reflect the upgradeable pattern:

```
// ============ Initialiser ============

/// @notice Initialises the SlashingManager contract
/// @param _admin The address that will be granted admin role
/// @param _slashingManagerConfigRole The address that will be granted the slashing manager config role
/// @param _stakeManager The address of the StakeManager contract
function initialize(address _admin, address _slashingManagerConfigRole, address _stakeManager)
    external
    initializer
{
    // ... implementation
}
```

2. **Event Emitted When No State Change Occurs In `removeCommitteeVaultStake()`**

   Related Asset(s): contracts/SSPRouter.sol

   The `SSPRouter.removeCommitteeVaultStake()` function emits a `CommitteeVaultStakeRemoved` event even when no state modification occurs, which can create misleading event logs.

   The function retrieves the operator stake and only deletes the mapping entry if the stake is nonzero, but the event is emitted unconditionally:

```
function removeCommitteeVaultStake(uint96 committeeId, address vault, address operator)
    external
    onlyRole(STAKE_MANAGER_ROLE)
{
    if (committeeId == 0) revert InvalidCommitteeId();
    uint256 operatorStake = committeeVaultTotalStake[committeeId][vault];

    if (operatorStake > 0) {
        delete committeeVaultTotalStake[committeeId][vault];
    }

    emit CommitteeVaultStakeRemoved(committeeId, vault, operator, operatorStake);  // @audit Emitted even when operatorStake
        ↪  is 0
}
```

If `operatorStake` is zero before the function is called, the function performs no state changes but still emits an event indicating that stake was removed.

Consider moving the event emission inside the conditional block so it only fires when state is actually modified:

```
function removeCommitteeVaultStake(uint96 committeeId, address vault, address operator)
    external
    onlyRole(STAKE_MANAGER_ROLE)
{
    if (committeeId == 0) revert InvalidCommitteeId();
    uint256 operatorStake = committeeVaultTotalStake[committeeId][vault];

    if (operatorStake > 0) {
        delete committeeVaultTotalStake[committeeId][vault];
        emit CommitteeVaultStakeRemoved(committeeId, vault, operator, operatorStake);
    }
}
```

3. **Event Emitted When No Vault Removal Occurs In `removeCommitteeVault()`**

   Related Asset(s): contracts/SSPRouter.sol

   The `SSPRouter.removeCommitteeVault()` function emits a `CommitteeVaultRemoved` event even when no state modification occurs, similar to the issue in `removeCommitteeVaultStake()`.

   The function retrieves the module type and conditionally removes the vault from the mapping, but emits the event unconditionally:

```
function removeCommitteeVault(uint96 committeeId, address vault) external onlyRole(STAKE_MANAGER_ROLE) {
    if (committeeId == 0) revert InvalidCommitteeId();
    if (vault == address(0)) revert ZeroAddress();

    // Get module type before removing vault
    ISSPRouter.SSPModuleType moduleType = vaultToModule[vault];

    // Remove from module-specific mapping
    if (moduleType != ISSPRouter.SSPModuleType.NONE) {
        committeeModuleVaults[committeeId][moduleType].remove(vault);
    }

    emit CommitteeVaultRemoved(committeeId, vault);  // @audit Emitted even when moduleType is NONE
}
```

If `moduleType` is `NONE`, the function performs no state changes but still emits an event indicating a vault was removed.

Consider moving the event emission inside the conditional block so it only fires when state is actually modified:

```
function removeCommitteeVault(uint96 committeeId, address vault) external onlyRole(STAKE_MANAGER_ROLE) {
    if (committeeId == 0) revert InvalidCommitteeId();
    if (vault == address(0)) revert ZeroAddress();

    // Get module type before removing vault
    ISSPRouter.SSPModuleType moduleType = vaultToModule[vault];

    // Remove from module-specific mapping
    if (moduleType != ISSPRouter.SSPModuleType.NONE) {
        committeeModuleVaults[committeeId][moduleType].remove(vault);
        emit CommitteeVaultRemoved(committeeId, vault);
    }
}
```

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team have made the following changes.

- Updated the constructor documentation in `SlashingManager` to correctly reflect the upgradeable pattern by changing the comment from “Constructor” to “Initialiser” and updating the function description accordingly.
- Modified `removeCommitteeVaultStake()` in `SSPRouter` to only emit the `CommitteeVaultStakeRemoved` event when state is actually modified (i.e., when `operatorStake > 0`).
- Modified `removeCommitteeVault()` in `SSPRouter` to only emit the `CommitteeVaultRemoved` event when state is actually modified (i.e., when `moduleType != NONE`).

The fixes can be found in PR 496.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

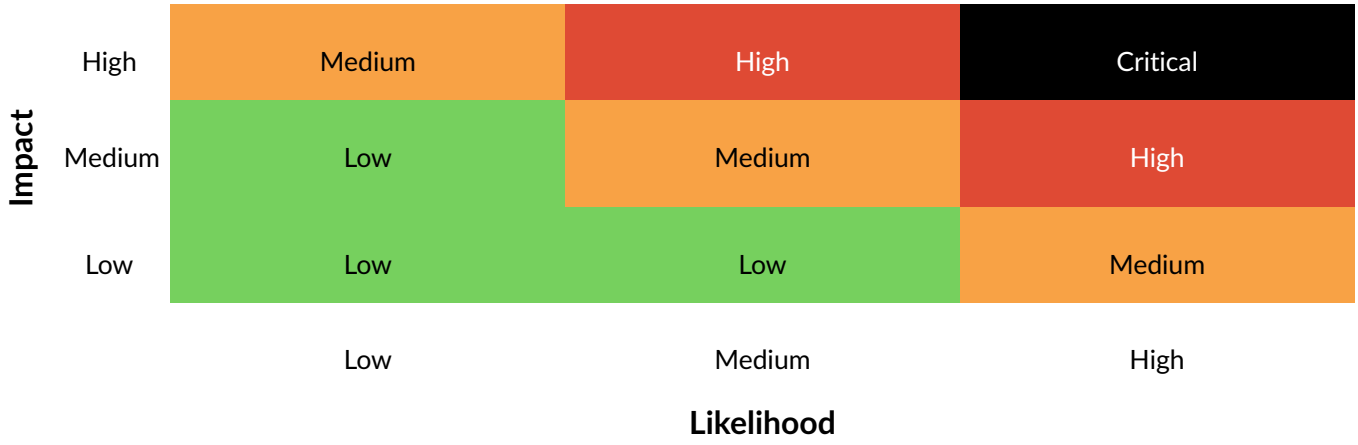

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].