CATALYSIS

Core and Coverage Smart Contracts

Security Assessment Report

Version: 2.0

May, 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 contracts, 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.

This review focusses on recent updates, especially relating to removing the USD denomination of coverage and claim amounts.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Catalysis core and coverage repositories.

The scope of this time-boxed review was strictly limited to changes to `src/contracts` in core diff 9e1c5bc...a3b70c5 and to `src/*` in coverage diff d8d492a...d5e2c9d, excluding `src/defi/*`, `src/interfaces/*` and `src/mocks/*`.

The fixes of the identified issues were assessed at core commit 85e4165 and coverage commit a3a2a7e.

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

Approach

The security assessment covered components written in Solidity.

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.

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 12 issues during this assessment. Categorised by their severity:

- Critical: 2 issues.
- High: 1 issue.

- Medium: 3 issues.
- Low: 3 issues.
- Informational: 3 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 finding 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 finding has not been addressed by the project team.
- ***Resolved:*** the finding was acknowledged by the project team and updates to the affected components have been made to mitigate the related risk.
- ***Closed:*** the project team has reviewed and acknowledged the finding, but the recommended remediation has not been implemented. The project team has typically provided a documented rationale supporting this decision, including the approach taken and any associated risk considerations.

Summary of Findings

ID	Description	Severity	Status
CYS3-01	Raw `coverageLimit` Values Misconfigure Eigen Vault Limits	Critical	Resolved
CYS3-02	Upgradeable Storage Layout Changes Can Corrupt Live Proxies	Critical	Resolved
CYS3-03	DEX Quotes Are A Single Point Of Failure For Claim Settlement	High	Resolved
CYS3-04	Claim Suspension Can Make Valid Claims Expire	Medium	Closed
CYS3-05	Mutable Module Adapters Can Strand Protocol State	Medium	Resolved
CYS3-06	Policies Can Be Backed By Unswappable Collateral	Medium	Resolved
CYS3-07	Buyers Do Not Lock Accepted Collateral Tokens	Low	Resolved
CYS3-08	Slashed Collateral Can Remain Stranded After Payout Is Filled	Low	Resolved
CYS3-09	Oracle-Based Slash Sizing Temporarily Reduces Effective Coverage	Low	Closed
CYS3-10	Any Account Can Fund Policy Premiums	Informational	Resolved
CYS3-11	Curator Next-Claim Approval Does Not Bind Claim Details	Informational	Resolved
CYS3-12	Miscellaneous General Comments	Informational	Resolved

CYS3-01	Raw `coverageLimit` Values Misconfigure Eigen Vault Limits		

Assets	`coverage/src/interfaces/ICoverPool.sol`
	`coverage/src/CoverPool.sol`
	`coverage/src/PolicyManager.sol`
	`core/src/contracts/SSPRouter.sol`
	`core/src/contracts/OraclePriceFeed.sol`		

Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Eigen duration vault limits can be configured orders of magnitude above or below the intended coverage notional because coverage passes a payout-token amount into the core code which then interprets the value as 8-decimal USD.

Coverage quotes define `coverageLimit` as a payout-token amount. For example, a policy paying USDC is expected to express `coverageLimit` with 6 USDC decimals, while a policy paying DAI is expected to express it with 18 DAI decimals.

coverage/src/interfaces/ICoverPool.sol::Quote

```
struct Quote {
    address pool;
    uint96 policyId;
    // @audit This value is documented as payout-token units.
    uint256 coverageLimit;
    bytes32 specId;
    bytes32 termsHash;
    address beneficiary;
    uint32 expirationTime;
}
```

When a policy is bound, `CoverPool.bindPolicyForRequest()` passes this raw payout-token amount through `StakeManager` into `SSPRouter`. The payout token is not passed with it.

coverage/src/CoverPool.sol::bindPolicyForRequest()

```
// @audit quote.coverageLimit is still in the policy payout token's native decimals.
IStakeManager(stakeManager).setCommitteeVaults(draft.policyId, vaults, quote.coverageLimit);
```

`SSPRouter` then uses the same raw value as a USD amount when configuring EigenLayer duration vault limits. The helper called here expects `amountUSD`, and `OraclePriceFeed.getTokenAmount()` documents that this value is 8-decimal USD.

core/src/contracts/SSPRouter.sol::_configureEigenStrategies()

```
address vaultUnderlyingToken = address(IStrategy(eigenVaults[i]).underlyingToken());
// @audit coverageLimit is treated as an 8-decimal USD value.
uint256 coverageLimitTokenAmount = _convertUSDToTokenAmount(vaultUnderlyingToken, coverageLimit);
if (coverageLimitTokenAmount == 0) revert InvalidAmount();
IEigenAdapter(eigenAdapter).updateTVLLimits(
    eigenVaults[i], coverageLimitTokenAmount, coverageLimitTokenAmount
);
```

core/src/contracts/OraclePriceFeed.sol::getTokenAmount()

```
// Convert USD value to token amount
// usdValue is in 8 decimals (Chainlink standard)
// price is in 8 decimals (USD per token)
// Result will be in tokenDecimals
// Formula: tokenAmount = (usdValue * 10^tokenDecimals) / price
return (usdValue * (10 ** tokenDecimals)) / uint256(price);
```

This is inconsistent with the later policy sufficiency check, which correctly converts the payout-token `coverageLimit` to USD by using the policy's `payoutToken`. That conversion happens after the core vault configuration has already received the mis-scaled raw amount.

coverage/src/PolicyManager.sol::bindPolicy()

```
uint256 coverageLimitUSD =
    // @audit PolicyManager correctly values the payout-token amount only after core has used
    //        the raw value for Eigen vault limits.
    IChainlinkPriceFeed(chainlinkPriceFeed).getUSDValue(request.payoutToken, boundPolicy.coverageLimit);
require(coverageLimitUSD > 0, ZeroCoverageLimitUSD());
require(totalStakeUSD >= coverageLimitUSD, InsufficientStake(policyId, coverageLimitUSD, totalStakeUSD));
```

The resulting scale error depends on the payout token decimals. A `coverageLimit` of `1_000e6` for a 1,000 USDC policy is treated as `1_000e6` units of 8-decimal USD, or only 10 USD. A `coverageLimit` of `1_000e18` for a 1,000 DAI policy is treated as 10,000,000,000,000 USD. Both values are detached from the intended policy notional.

This issue has a high impact because Eigen duration vault limits are core risk controls for the collateral backing coverage. Under-scaled limits can prevent policies from being backed as intended or lock Eigen vaults with unusably small capacity. Over-scaled limits can permit far more Eigen stake exposure than the policy notional and the pool's risk configuration intended.

This issue has a high likelihood because it affects every Eigen-backed policy whose payout token does not use 8 decimals. Common payout tokens such as USDC and DAI use 6 and 18 decimals respectively, so ordinary policy creation can trigger the bug without unusual conditions.

Recommendations

Pass a correctly normalised value into core.

One fix is to compute `coverageLimitUSD` in `PolicyManager` or `CoverPool` using the policy payout token, then pass the 8-decimal USD value to `StakeManager.setCommitteeVaults()` and `SSPRouter.setCommitteeVaults()`.

Alternatively, change the core interface to accept both `payoutToken` and `coverageLimit`, and modify the internal function `_configureEigenStrategies()` to convert from the payout token to each vault's underlying token.

Resolution

The coverage limit value is now correctly normalised to 8-decimal USD before being passed to core.

Resolved in coverage PR 99.

Resolved in core PR 538.

CYS3-02	Upgradeable Storage Layout Changes Can Corrupt Live Proxies		
Assets	core/src/contracts/extensions/RoleActivationTimelock.sol core/src/contracts/StakeManager.sol core/src/contracts/RewardsManager.sol core/src/contracts/SlashingManager.sol core/src/contracts/SSPRouter.sol core/src/contracts/OraclePriceFeed.sol core/src/contracts/Adapters/BaseAdapter.sol core/src/contracts/Adapters/EigenAdapter.sol core/src/contracts/Adapters/SymbioticAdapter.sol coverage/src/ClaimManager.sol coverage/src/PremiumManager.sol coverage/src/CoverPool.sol coverage/src/PolicyManager.sol coverage/src/interfaces/IPolicyManager.sol coverage/src/interfaces/ICoverPool.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Several upgradeable Core and Coverage contracts change their storage layouts in ways that can corrupt live proxy state during an in-place upgrade.

Upgradeable implementations must preserve storage slot order and type compatibility across versions. The new implementations instead insert storage-bearing base contracts, remove state variables, reuse existing slots for different meanings, change mapping value types, and modify structs that are already stored inside mappings. Existing proxy state would therefore be read as different variables or become inaccessible after the upgrade.

The largest break in core is the replacement of direct `AccessControlUpgradeable` inheritance with the wrapping contract `RoleActivationTimelock`. `RoleActivationTimelock` stores `delay` and `_roleGrantedAt` in ordinary Solidity storage. Because it is inherited before each affected contract's existing state variables, those fields are inserted before the child contract storage.

extensions/RoleActivationTimelock.sol

```
abstract contract RoleActivationTimelock is AccessControlUpgradeable {
    // @audit These variables are inserted before child contract storage.
    uint256 public delay;
    mapping(bytes32 => mapping(address => uint256)) private _roleGrantedAt;
}
```

The storage layout breaks at the following sites:

- `StakeManager`, `RewardsManager`, `SlashingManager`, `SSPRouter`, `OraclePriceFeed`, and all adapter contracts: inserting `RoleActivationTimelock` into the inheritance chain shifts all child contract state by two slots.
- `RewardsManager` removes `totalRewardsPerNetworkInstance` and `rewardDistributionCounter` without deprecated placeholders.
- `SlashingManager` removes `slashingCounter` without a deprecated placeholder.
- `SSPRouter` removes `committeeStakeRequirement` without a deprecated placeholder.

- `StakeManager` replaces a `uint96` mapping with an `EnumerableSet.UintSet` at the same logical position, making existing data undecodable.
- `ClaimManager`: the slot previously storing `chainlinkPriceFeed` is reused for `premiumManager`, causing the proxy to misinterpret live state.
- `PremiumManager`: `_approvedPremiumTokens` is inserted before `_distributionNonce`, shifting the nonce into a later slot.
- `CoverPool`: `poolFeeBps` is moved from before the bound-policy mappings to after them, changing the seed slots of those mappings. Struct fields are also removed from stored `BoundPolicy` records, leaving stale data in previously occupied slots.
- `PolicyManager`: fields are removed from and inserted into structs stored in mappings (`PolicyDraft` as well as `PolicyMetadata`), causing all subsequent struct members to be read from incorrect storage positions.

This issue has a high impact because a live in-place upgrade can cause significant operational disruption and moderate financial loss. Existing manager addresses, price feed registrations, policy records, committee associations, pool settings, bound policy mappings, reward replay state, and slashing replay state may be lost, misread, or stranded in stale slots. The likely outcomes are disabled policy binding or claim settlement, incorrect fee and reward configuration, lost access to historical policy state, and failed administrative recovery paths.

This issue has a high likelihood because the incompatible storage changes are unconditional in multiple in-scope upgradeable implementations. Any existing proxy or upgradeable clone moved to these implementations without a layout-preserving migration would encounter the mismatch.

However, these ratings are in the context of upgrades to deployed contracts. If deployed contracts are not going to be upgraded with these changes, then the severity is greatly reduced.

Recommendations

Add storage gaps at the end of storage variable lists for all contracts that may possibly be upgraded in future to allow for new variables to be added without significant technical obstacles.

If deployed contracts are being upgraded, all of the following points are recommended:

- Restore storage compatibility for every upgradeable contract before upgrading live proxies. Do not insert storage-bearing base contracts before existing child storage. For the role activation delay, use ERC-7201 namespaced storage.
- Keep removed variables as deprecated placeholders with their original types and positions, perhaps prepending "_deprecated" or "_legacy" to their old variable names.
- Do not reuse an existing slot for a different semantic purpose, such as replacing `chainlinkPriceFeed` with `premiumManager`. Append new variables instead.
- Keep the old `StakeManager` operator committee mapping in place and append the new multi-committee mapping. Provide an explicit, idempotent migration path to populate the new representation from the old representation where live data must be retained.
- Do not change the order or contents of structs already stored inside mappings. Append new struct fields to the end only, or deploy new storage and migrate every live record through a controlled one-time process. For `CoverPool`, preserve the original `poolFeeBps`, `boundPolicyCommits`, `boundPolicies`, and `BoundPolicy` storage positions, then append any pending-fee timelock state.
- Add storage-layout validation and fork tests that upgrade populated old deployments and assert that roles, manager addresses, price feeds, policies, committees, counters, mappings, and pool settings are preserved.

Resolution

The concerns about potential storage collisions during upgrades have been acknowledged and will be managed by admin where necessary.

Storage layout validation tests have been added to ensure predictable storage updates whenever contracts are extended.

Resolved in coverage PR 107.

Resolved in core PR 542.

CYS3-03	DEX Quotes Are A Single Point Of Failure For Claim Settlement		
Assets	coverage/src/ClaimManager.sol coverage/src/Swapper.sol coverage/src/swappers/UniswapV3Adapter.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Cross-token claim settlement uses the configured DEX route as the only source of truth for collateral valuation, making payout execution dependent on a single manipulable or unavailable market.

`ClaimManager` must convert slashed collateral into the policy payout token when the collateral token differs from `metadata.payoutToken`. The contract does not use the Chainlink price feeds that are used during policy binding to sanity-check the claim-time value. Instead, it asks `Swapper.quoteSwap()` for the value of each full vault stake, then uses those DEX quotes to decide how much of each collateral token should be slashed.

coverage/src/ClaimManager.sol::_computeVaultSlashes()

```
if (tokens[i] == payoutToken) {
    payoutEquivalents[i] = tokenStakes[i];
} else {
    // @audit The DEX quote is the only valuation source used to size cross-token slashes.
    try ISwapper(swapper).quoteSwap(tokens[i], payoutToken, tokenStakes[i]) returns (uint256 quoted) {
        payoutEquivalents[i] = quoted;
    } catch {}
}

// ...

uint256 baseSlash = (tokenStakes[i] * requestedAmount) / totalPayoutEquivalent;
```

The same quote path is used again immediately before execution to derive the minimum accepted swap output. If the quote is manipulated, stale, based on thin liquidity, or unavailable, the claim path either accepts a bad execution bound or reverts.

coverage/src/ClaimManager.sol::_computeAmountOutMin()

```
try ISwapper(swapper).quoteSwap(tokenIn, tokenOut, amountIn) returns (uint256 quoted) {
    if (quoted == 0) revert QuoteUnavailable(tokenIn, tokenOut);
    // @audit The minimum output is derived only from the DEX quote.
    uint256 amountOutMin = (quoted * (uint256(_MAX_SLIPPAGE_BPS) - maxSwapSlippageBps)) / _MAX_SLIPPAGE_BPS;
    return amountOutMin == 0 ? 1 : amountOutMin;
} catch {
    revert QuoteUnavailable(tokenIn, tokenOut);
}
```

`Swapper.quoteSwap()` simply delegates to the configured route target. For the included Uniswap V3 adapter, the target reads the current Uniswap pool quote through `QuoterV2`. There is no independent oracle bound, time-weighted average price check, liquidity floor, or comparison to the bind-time Chainlink valuation.

coverage/src/Swapper.sol::quoteSwap()

```
SwapRoute storage route = _swapRoutes[routeKey];

if (!route.enabled || route.swapTarget == address(0)) return 0;

// @audit The configured route target is the sole pricing source for this pair.
try IQuotableAdapter(route.swapTarget).quote(amountIn, route.swapCalldata) returns (uint256 quoted) {
    return quoted;
} catch {
    return 0;
}
```

coverage/src/swappers/UniswapV3Adapter.sol::quote()

```
IQuoterV2.QuoteExactInputSingleParams memory params = IQuoterV2.QuoteExactInputSingleParams({
    tokenIn: tokenIn, tokenOut: tokenOut, amountIn: amountIn, fee: fee, sqrtPriceLimitX96: 0
});

// @audit Current pool state is used directly as the claim settlement oracle.
try QUOTER.quoteExactInputSingle(params) returns (uint256 out, uint160, uint32, uint256) {
    return out;
} catch {
    return 0;
}
```

This creates two failure modes.

First, if the route or its underlying pool is unavailable, the claim path reverts for policies that require cross-token conversion.

Second, and more importantly, if the route's spot price is manipulated, claim settlement can over-slash collateral, under-slash collateral, or accept settlement bounds that do not reflect the broader market value of the assets. The issue is most severe for thin pools, exotic collateral, or payout pairs where the configured DEX route is the only liquid market.

This issue has a high impact because DEX quote failure or manipulation can block valid claims or cause economically incorrect slashing and payout behaviour for policies that were sold with a fixed payout token. The affected path is part of the core claim settlement flow and directly controls beneficiary payment and restaker losses.

This issue has a medium likelihood because the attacker or adverse market condition must affect the configured DEX route for the collateral-to-payout pair.

Recommendations

Do not use a DEX quote as the sole source of truth for claim settlement valuation. Apply an independent oracle or time-weighted price bound to every cross-token quote before it is used to size slashes or set `amountOutMin`.

Require configured routes to satisfy minimum liquidity and freshness constraints. For Uniswap V3 routes, compare the spot quote against Chainlink-derived fair value within a bounded tolerance. Alternatively, but less ideally, a TWAP can also be used as a fair value reference.

If no reliable independent price is available for a collateral-to-payout pair, prevent that collateral from backing policies that promise payouts in a different token.

Resolution

An independent oracle price bound is now applied to cross-token DEX quotes during claim settlement.

Resolved in coverage PR 101.

CYS3-04	Claim Suspension Can Make Valid Claims Expire		
Assets	coverage/src/ClaimManager.sol coverage/src/SpecRegistry.sol coverage/src/Swapper.sol core/src/contracts/SlashingManager.sol		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Temporary suspension mechanisms can permanently prevent otherwise valid policy claims from being paid if the suspension lasts until after the policy maturity time.

The coverage claim flow combines claim filing, spec evaluation, slashing, swapping, and payout in one atomic `ClaimManager.fileClaim()` call. Before any claim can be resolved, `ClaimManager` requires the transaction to execute while the policy is inside its active coverage window.

coverage/src/ClaimManager.sol::_fileClaimWithMetadata()

```
require(
    block.timestamp >= metadata.startTime && block.timestamp < metadata.maturityTime,
    ClaimOutsideCoverageWindow(policyId, metadata.startTime, metadata.maturityTime, uint32(block.timestamp))
);
```

The same transaction then resolves the registered spec and proceeds to settlement. If the spec has been revoked in `SpecRegistry`, `resolveSpec()` reverts and the whole claim transaction is rolled back. No claim record is preserved for later evaluation.

coverage/src/ClaimManager.sol::_evaluateSpec()

```
ISpec spec = ISpecRegistry(specRegistry).resolveSpec(metadata.pool, metadata.specId);
```

coverage/src/SpecRegistry.sol::resolveSpec()

```
spec = _specs[coverPool][specId];
require(address(spec) != address(0), SpecNotRegistered(coverPool, specId));
require(_approvedSpecs[address(spec)], SpecNotApproved(address(spec)));
```

This means an emergency action intended to be temporary can become permanent for near-expiry policies. For example, if a valid loss occurs before maturity but the relevant spec is revoked until after maturity, the claimer cannot successfully file during the revocation period. Once the spec is re-approved, the same claim can no longer be filed because the policy is outside the coverage window.

The issue is not limited to spec revocation. Any mechanism or dependency that suspends the atomic claim path can have the same effect if it lasts past maturity:

- `ClaimManager.pause()` blocks `fileClaim()` directly through `whenNotPaused`.
- `SpecRegistry.revokeSpec()` blocks spec resolution through `resolveSpec()`.
- A paused or unavailable core `SlashingManager` blocks settlement after a payable spec result.
- A paused or unavailable `Swapper`, or a missing swap route, blocks settlement for claims that need collateral conversion.

This issue has a high impact because a policyholder can lose the practical ability to claim for a covered loss, despite the loss occurring during the policy window.

This issue has a low likelihood because it requires a suspension period to overlap with a valid claim opportunity and continue until maturity. It is most likely during emergency response, governance delay, operational misconfiguration, or dependency outage rather than ordinary operation.

Recommendations

Split claim filing from claim resolution. Let claims be filed and durably recorded before maturity, then evaluated and settled later once the relevant suspension is lifted, such as after the spec is re-approved.

Resolution

The development team has closed the issue with the following comment:

> *This breaks our atomic claim filing requirement and it affects only some emergency or misconfiguration cases so we just acknowledge it.*

CYS3-05	Mutable Module Adapters Can Strand Protocol State		
Assets	core/src/contracts/SSPRouter.sol core/src/contracts/Adapters/EigenAdapter.sol core/src/contracts/Adapters/SymbioticAdapter.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Privileged adapter replacement can disconnect existing committees and vaults from the protocol state that was created for the previous adapter, causing rewards, slashing, stake accounting, and committee configuration to fail or operate against the wrong module identity.

The `SSPRouter` contract stores the active adapter address for each module type in `adapters`. The `setAdapter()` function can be called by an account with `DEFAULT_ADMIN_ROLE` at any time and only rejects `address(0)` and the `NONE` module type. It does not check that the new adapter is a contract, that it exposes the expected interface, that it has granted `SSP_ROUTER_ROLE` to the router, that it has the current oracle configured, or that existing committee and vault state can be migrated to the new adapter.

core/src/contracts/SSPRouter.sol::setAdapter()

```
function setAdapter(ISSPRouter.SSPModuleType moduleType, address adapter)
    external
    onlyActiveRole(DEFAULT_ADMIN_ROLE)
{
    if (adapter == address(0)) revert ZeroAddress();
    if (moduleType == ISSPRouter.SSPModuleType.NONE) revert InvalidModuleAddress();

    // @audit The adapter for an active module can be replaced with any non-zero address.
    //        Existing vault and committee state is not migrated or validated.
    adapters[moduleType] = adapter;

    emit AdapterSet(moduleType, adapter);
}
```

This is especially problematic for EigenLayer because the adapter address is used as the AVS identity. When a committee is created, `EigenAdapter` creates an operator set for `address(this)`. When duration vault strategies are deployed, their admin, arbitrator, and `OperatorSet` are also bound to `address(this)`.

core/src/contracts/Adapters/EigenAdapter.sol::createRedistributingOperatorSet()

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

// ...

// Use this EigenAdapter as the AVS and set claim manager as redistribution recipient
allocationManager.createRedistributingOperatorSets(address(this), paramsArray, redistributionRecipients);
// @audit The Eigen adapter address becomes the AVS address for this operator set.
```

core/src/contracts/Adapters/EigenAdapter.sol::deployDurationVaultStrategy()

```
IDurationVaultStrategyTypes.VaultConfig memory vaultConfig = IDurationVaultStrategyTypes.VaultConfig({
    underlyingToken: IERC20_V4(underlyingToken),
    // @audit These privileged roles are bound to the adapter that deployed the vault.
    vaultAdmin: address(this),
    arbitrator: address(this),
    duration: duration,
    maxPerDeposit: type(uint256).max,
    stakeCap: type(uint256).max,
    metadataURI: "",
    // @audit The vault is bound to the old adapter's AVS identity.
    operatorSet: OperatorSet({avs: address(this), id: uint32(committeeId)}),
    operatorSetRegistrationData: "",
    delegationApprover: address(0),
    operatorMetadataURI: ""
});
```

Later EigenLayer operations continue to use the current adapter's own address as the AVS. After replacing the router's Eigen adapter, the new adapter will submit rewards and slash against `OperatorSet({avs: address(this), id: committeeId})` for the new adapter address, while existing vaults and operator sets were created for the old adapter address.

core/src/contracts/Adapters/EigenAdapter.sol::_submitEigenRewards()

```
try rewardsCoordinator.createOperatorDirectedOperatorSetRewardsSubmission(
    // @audit This points to the new adapter after replacement, not the adapter that created
    //        the existing vaults and operator set.
    OperatorSet({avs: address(this), id: uint32(committeeId)}), submissions
) {} catch {
    revert RewardDistributionFailed(address(rewardsCoordinator));
}
```

core/src/contracts/Adapters/EigenAdapter.sol::_executeSlashing()

```
bytes memory callData = abi.encodeWithSelector(
    allocationManager.slashOperator.selector,
    // @audit Slashing is authorised for the current adapter's AVS identity.
    address(this), // avs
    slashParams
);

// ...

OperatorSet memory operatorSet = OperatorSet({avs: address(this), id: uint32(committeeId)});
strategyManager.clearBurnOrRedistributableShares(operatorSet, slashId);
```

Symbiotic has the same class of problem. The adapter address is encoded into the subnetwork key used for stake accounting and slashing. If the router is changed to a replacement `SymbioticAdapter`, the new adapter queries a different subnetwork than the one configured by the old adapter. In addition, rewarder mappings are stored inside the adapter and are not migrated.

core/src/contracts/Adapters/SymbioticAdapter.sol::getOperatorStakeUSD()

```
bytes32 subnetwork = bytes32((uint256(uint160(address(this))) << 96) | uint256(committeeId));
// @audit Replacing the adapter changes address(this), and therefore changes the
//        Symbiotic subnetwork used for stake accounting.
vaultStakesUSD = new uint256[](vaults.length);
```

core/src/contracts/Adapters/SymbioticAdapter.sol::registerVaultRewards()

```
// @audit Vault rewarder configuration is local to a single adapter instance.
vaultToRewardsContract[vault] = rewardsContract;
```

This issue has a high impact because replacing an adapter after committees and vaults exist can break core safety-critical module operations. Rewards may be submitted against the wrong EigenLayer AVS or fail entirely, slashing may execute against a non-existent or unrelated operator set, Symbiotic stake accounting may query a fresh subnetwork with no configured limits or stake, and existing rewarder mappings may be lost. These outcomes can prevent the system from enforcing coverage, distributing rewards, or recovering slashed collateral correctly.

This issue has a low likelihood because exploiting it requires privileged access to `DEFAULT_ADMIN_ROLE`, a governance or operational error, or an unsafe upgrade or migration. Nevertheless, the affected function is an intended administrative control, and the current implementation does not provide guardrails that prevent a well-intentioned adapter upgrade from stranding live module state.

Recommendations

Make module adapter replacement an explicit migration flow rather than a simple setter. At minimum, prevent replacing an adapter once any vault has been registered or any committee has been created for that module until all coverage periods have expired for a given cooldown period.

If adapter replacement is required, implement a dedicated migration procedure that validates the new adapter before activation. This validation procedure can be executed offchain if performed by a trusted role, but ideally would be onchain. The migration should verify that the new adapter is a contract, has granted `SSP_ROUTER_ROLE` to the router, has the current oracle configured, is registered with the relevant EigenLayer or Symbiotic protocol state, and can operate on every existing committee and vault. For EigenLayer, the migration must account for AVS identity, operator sets, duration vault admin and arbitrator roles, and redistribution recipients. For Symbiotic, the migration must account for subnetwork identity, network limits, middleware registration, and vault rewarder mappings.

Resolution

Adapter replacement now includes validation guards to prevent stranding live module state.

Resolved in core PR 541.

CYS3-06	Policies Can Be Backed By Unswappable Collateral		
Assets	coverage/src/PolicyManager.sol coverage/src/ClaimManager.sol coverage/src/Swapper.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Policies can potentially pass the bind-time collateral sufficiency check using collateral that cannot be converted into the policy payout token, causing claims to exclude that collateral or fail entirely.

`PolicyManager.bindPolicy()` validates backing by asking core for the committee's vault tokens and raw token stakes, then pricing each token through the Chainlink price feed. This confirms that the selected collateral has sufficient USD value at bind time, but it does not verify that any non-payout collateral can be converted into the policy's `payoutToken`.

coverage/src/PolicyManager.sol::bindPolicy()

```
(, address[] memory tokens, uint256[] memory tokenStakes) =
    IStakeManager(stakeManager).getCommitteeTokenStakes(policyId);

uint256 totalStakeUSD = 0;
for (uint256 i = 0; i < tokens.length; ++i) {
    if (tokenStakes[i] > 0) {
        // @audit Any token with a Chainlink price feed can count as backing.
        //        No payout-token swap route is checked here.
        totalStakeUSD += IChainlinkPriceFeed(chainlinkPriceFeed).getUSDValue(tokens[i], tokenStakes[i]);
    }
}
uint256 coverageLimitUSD =
    IChainlinkPriceFeed(chainlinkPriceFeed).getUSDValue(request.payoutToken, boundPolicy.coverageLimit);
require(totalStakeUSD >= coverageLimitUSD, InsufficientStake(policyId, coverageLimitUSD, totalStakeUSD));
```

At claim time, however, cross-token collateral is only usable if the configured `Swapper` can quote the collateral-to-payout pair. If the quote reverts or returns zero, the vault is excluded from slashing.

coverage/src/ClaimManager.sol::_computeVaultSlashes()

```
if (tokens[i] == payoutToken) {
    payoutEquivalents[i] = tokenStakes[i];
} else {
    // @audit A vault that counted as backing at bind time is excluded if it has no
    //        functioning route to the payout token at claim time.
    try ISwapper(swapper).quoteSwap(tokens[i], payoutToken, tokenStakes[i]) returns (uint256 quoted) {
        payoutEquivalents[i] = quoted;
    } catch {}

    if (payoutEquivalents[i] == 0) {
        emit VaultExcludedFromSlashing(vaults[i], tokens[i]);
    }
}
```

If every cross-token backing vault is unquotable, `_computeVaultSlashes()` returns an empty array and the payable claim reverts before settlement.

coverage/src/ClaimManager.sol::_slash()

```
ISlashingManager.VaultSlash[] memory vaultSlashes =
    _computeVaultSlashes(vaults, tokens, tokenStakes, metadata.payoutToken, requestedAmount);

// @audit A policy can be fully backed by USD value but have no slashable collateral
//        once swap route availability is considered.
require(vaultSlashes.length > 0, NoCollateralSlashed(metadata.policyId));
```

`Swapper.quoteSwap()` returns zero when the route is disabled or missing. The route can therefore be absent at binding, removed before a claim, or become unusable due to the configured target reverting.

coverage/src/Swapper.sol::quoteSwap()

```
bytes32 routeKey = _getRouteKey(actualTokenIn, actualTokenOut);
SwapRoute storage route = _swapRoutes[routeKey];

// @audit Missing or disabled routes make the collateral unusable for claim settlement.
if (!route.enabled || route.swapTarget == address(0)) return 0;

try IQuotableAdapter(route.swapTarget).quote(amountIn, route.swapCalldata) returns (uint256 quoted) {
    return quoted;
} catch {
    return 0;
}
```

This issue has a high impact because a policyholder can hold apparently valid, fully collateralised coverage but receive no payout when the only backing collateral cannot be swapped to the promised payout token. Partial route failure can also shift the whole claim burden onto the remaining quotable vaults, unfairly slashing one subset of restakers while ignoring collateral that was counted during policy binding.

This issue has a low likelihood because it requires the curator to bind a policy with collateral that lacks a usable payout route, or requires the route to fail after binding. The condition is nevertheless realistic during new token onboarding, route migration, DEX outage, or configuration error.

Recommendations

Require every non-payout collateral token selected for a policy to have an enabled, non-zero quote route to the policy payout token before binding succeeds. Make route availability part of the same invariant as stake sufficiency, so collateral cannot count as policy backing unless it can be used for claim settlement.

Prevent route removal or target de-whitelisting while active policies depend on the route, or add a timelocked migration flow that replaces the route before disabling the old one. If route availability cannot be guaranteed, store the unsupported collateral as explicitly excluded from coverage backing and do not count it in `totalStakeUSD`.

Resolution

Swap route availability is now verified at bind time for all non-payout collateral tokens.

Resolved in coverage PR 106.

CYS3-07	Buyers Do Not Lock Accepted Collateral Tokens		
Assets	coverage/src/PolicyManager.sol coverage/src/interfaces/ICoverPool.sol coverage/src/CoverPool.sol core/src/contracts/SSPRouter.sol core/src/contracts/OraclePriceFeed.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Coverage buyers can commit to a payout token and payout amount without locking the collateral tokens that will back the policy onchain, creating a potential mismatch between the purchased payout asset and the actual collateral risk accepted by the policy.

`PolicyManager.requestCoverage()` records the payout token requested by the buyer, but it does not receive or store a list of acceptable collateral tokens or backing vaults. The policy draft therefore captures the token in which claims should be paid, but not the token risk that will support those claims.

coverage/src/PolicyManager.sol::requestCoverage()

```
function requestCoverage(
    address pool,
    address beneficiary,
    address claimer,
    address payoutToken,
    uint32 duration,
    address bindPolicyHook
) external returns (uint96 policyId) {
    require(supportedPayoutTokens[payoutToken], UnsupportedPayoutToken(payoutToken));

    _policyDrafts[policyId] = PolicyDraft({
        pool: pool,
        buyer: msg.sender,
        beneficiary: beneficiary,
        claimer: claimer,
        // @audit The buyer commits to the payout token, but no accepted
        //        collateral-token list is committed here.
        payoutToken: payoutToken,
        curator: curator,
        duration: duration,
        policyId: policyId,
        status: PolicyStatus.Created,
        bindPolicyHook: bindPolicyHook
    });
}
```

The signed quote also commits to the policy notional, spec and terms hash, but does not include the backing vaults or the collateral tokens. The backing vaults are supplied later by the pool curator during `bindPolicyForRequest()`.

coverage/src/interfaces/ICoverPool.sol::Quote

```
struct Quote {
    address pool;
    uint96 policyId;
    uint256 coverageLimit;
    bytes32 specId;
    bytes32 termsHash;
    address beneficiary;
    uint32 expirationTime;
    // @audit No vault list or accepted collateral-token list is included.
}
```

coverage/src/CoverPool.sol::bindPolicyForRequest()

```
function bindPolicyForRequest(
    uint96 policyId,
    Quote calldata quote,
    address spec,
    address[] calldata vaults,
    bytes calldata signature
)
// ...
{
// ...
    bytes32 policyCommit = _verifyQuote(quote, signature);
    BoundPolicy memory boundPolicy = _recordBoundPolicy(policyId, policyCommit, quote);

    ISpecRegistry(specRegistry).registerSpec(quote.specId, ISpec(spec));

    // Add vaults to committee in core contracts using policyId as committeeId (1:1 mapping).
    IStakeManager(stakeManager).setCommitteeVaults(draft.policyId, vaults, quote.coverageLimit);
    // @audit The curator chooses the backing vaults during binding, after the
    //        buyer's policy draft has already been created.

    IPolicyManager(policyManager)
        .bindPolicy(
            IPolicyManager.BindPolicyRequest({
                policyId: policyId,
                buyer: draft.buyer,
                beneficiary: draft.beneficiary,
                claimer: draft.claimer,
                payoutToken: draft.payoutToken,
                quote: quote,
                signature: signature
            }),
            boundPolicy
        );
}
```

This creates two related problems.

First, a buyer approves the payout token amount, but does not directly approve the collateral-token list for the cover being purchased. For example, a buyer may believe coverage denominated in token A is economically backed by token A, while the final policy is backed by token A1, a restaked or derivative version of token A. The assets may be highly correlated, but token A1 can still have a different liquidity, could depeg, be slashed, or otherwise differ in a manner that affects its risk profile.

Second, the collateral is governed by mutable admin-controlled configuration rather than a per-policy buyer-approved snapshot. The router's acceptable token set, vault module registry and oracle feed registry can be changed by privileged roles. There is no policy-level invariant that the final vault tokens must match a collateral list accepted by the buyer when the policy was requested.

This issue is potentially mitigated by `termsHash`. The offchain terms referenced by `termsHash` could specify the exact collateral tokens or backing vaults that the buyer has accepted. However, the contracts treat `termsHash` as an opaque value and do not enforce that the bound vaults match those terms.

This issue is also mitigated by the admin trust model. Admin is trusted not to introduce malicious collateral tokens, malicious oracle feeds, or unsuitable vaults. The residual risk is therefore primarily an expectation and documentation risk rather than a direct unauthorised fund-drain path.

This issue has a low impact because claim payouts remain denominated in the selected `payoutToken`, and the binding flow still requires the selected collateral to satisfy the protocol's USD-value check. The practical harm is that a buyer may accept a different collateral-risk profile than expected.

This issue has a low likelihood because it requires buyer misunderstanding, incomplete offchain terms, or privileged configuration changes around the request-to-bind lifecycle. It is nevertheless plausible during new collateral onboarding, restaked-token integrations, or product flows that present payout denomination more prominently than backing collateral.

Recommendations

Require the coverage purchase transaction to include the buyer-approved collateral-token list, then enforce that list when the committee is created and when vaults are bound. The policy should reject any vault whose underlying collateral token is not in the buyer-approved set.

If the product intentionally allows curator-selected collateral without onchain buyer approval, provide clear buyer-facing warnings and documentation. Include the approved collateral tokens or vaults in the offchain terms represented by `termsHash`, and make the binding flow or user interface show those terms before the buyer is considered committed.

Resolution

The accepted collateral token list is now locked at policy request time and enforced during binding.

Resolved in coverage PR 102.

CYS3-08	Slashed Collateral Can Remain Stranded After Payout Is Filled		
Assets	`coverage/src/ClaimManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Collateral from later vaults can be slashed and delivered to `ClaimManager`, then left idle in the contract if an earlier collateral leg fully satisfies the beneficiary payout.

`ClaimManager._executeSlashingAndPayout()` first computes all vault slash instructions and executes slashing through core. Only after the full slashing call returns does it iterate over the returned collateral tokens and amounts.

coverage/src/ClaimManager.sol::_executeSlashingAndPayout()

```
// @audit Slashing for every selected vault happens before any collateral is processed.
(address[] memory collateralTokens, uint256[] memory collateralAmounts) =
    _slash(metadata, claimId, requestedAmount);

uint256 tokensLength = collateralTokens.length;
// slither-disable-next-line incorrect-equality
require(tokensLength > 0 && tokensLength == collateralAmounts.length, NoCollateralSlashed(policyId));

uint256 remainingPayout = requestedAmount;
for (uint256 i = 0; i < tokensLength && remainingPayout > 0; ++i) {
    if (collateralAmounts[i] > 0) {
        uint256 amountOut = _processCollateral(
            policyId,
            claimId,
            collateralTokens[i],
            collateralAmounts[i],
            metadata.payoutToken,
            metadata.beneficiary,
            metadata.curator,
            remainingPayout,
            i
        );
        totalPayout += amountOut;
        remainingPayout -= amountOut;
    }
}
```

The loop condition stops processing once `remainingPayout` reaches zero. If the first processed token produces enough payout token to fill the claim, every later collateral entry is skipped. Those skipped entries have already been slashed and transferred to `ClaimManager`, but they are not sent to the beneficiary, swapped, or routed through the surplus distribution path.

The surplus distribution logic exists only inside `_processCollateral()`. Skipped entries never reach this function, so the surplus handling is bypassed.

coverage/src/ClaimManager.sol::_processCollateral()

```
if (collateralToken == payoutToken) {
    uint256 transferAmount = collateralAmount > remainingPayout ? remainingPayout : collateralAmount;
    if (transferAmount > 0) {
        _transferToken(collateralToken, beneficiary, transferAmount);
    }
    amountOut = transferAmount;

    // @audit Same-token surplus is distributed only when this collateral entry is processed.
    if (collateralAmount > transferAmount && premiumManager != address(0) && collateralToken != _NATIVE_ETH) {
        _distributeSurplus(
            policyId, claimId, collateralToken, curator, collateralAmount - transferAmount, surplusTaskId
        );
    }
} else {
    // ...

    // @audit Cross-token surplus is distributed only when this collateral entry is processed.
    if (swapOutput > amountOut && premiumManager != address(0) && payoutToken != _NATIVE_ETH) {
        _distributeSurplus(policyId, claimId, payoutToken, curator, swapOutput - amountOut, surplusTaskId);
    }
}
```

The only generic recovery path for these stranded assets is admin-controlled token recovery.

This issue has a low impact because the beneficiary still receives the requested payout and the stranded assets remain in `ClaimManager`. However, already-slashed restaker collateral becomes admin-custodied, can be omitted from intended surplus reward routing, and requires manual intervention to resolve.

This issue has a low likelihood because it requires a multi-vault or multi-token slash where an earlier collateral leg fully satisfies the claim before later returned collateral entries are processed. The condition is more likely during favourable swap execution, mixed same-token and cross-token collateral, or over-slashing caused by slippage buffers.

Recommendations

Consider processing every returned collateral entry after slashing, even once the beneficiary payout has been filled. When `remainingPayout == 0`, route the full remaining collateral entry through the existing surplus handling path instead of skipping it.

Resolution

All returned collateral entries are now processed through the surplus distribution path even after the beneficiary payout is filled.

Resolved in coverage PR 100.

CYS3-09	Oracle-Based Slash Sizing Temporarily Reduces Effective Coverage		
Assets	`coverage/src/ClaimManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Cross-token claim settlement systematically over-slashes restaker vaults by a configurable buffer percentage, temporarily reducing the collateral pool available for other claims.

This issue was introduced by the resolution of the CYS3-03 vulnerability. The fix changed `_computeVaultSlashes()` to use the oracle fair value rather than the DEX quote for proportional slash sizing, and applies a compound buffer to account for potential price deviation and swap slippage.

coverage/src/ClaimManager.sol::_computeVaultSlashes()

```
for (uint256 i = 0; i < vaultsLength; ++i) {
    if (tokenStakes[i] == 0) {
        continue;
    }

    if (tokens[i] == payoutToken) {
        payoutEquivalents[i] = tokenStakes[i];
    } else {
        // @audit Oracle fair value used as the valuation anchor
        uint256 fair = _oracleFairValue(tokens[i], payoutToken, tokenStakes[i]);

        uint256 dex = 0;
        // slither-disable-next-line calls-loop
        try ISwapper(swapper).quoteSwap(tokens[i], payoutToken, tokenStakes[i]) returns (uint256 quoted) {
            dex = quoted;
        } catch {
            revert QuoteUnavailable(tokens[i], payoutToken);
        }
        if (dex == 0) revert QuoteUnavailable(tokens[i], payoutToken);

        // @audit DEX quote validated against oracle but NOT used for sizing
        _validateDexAgainstOracle(tokens[i], payoutToken, dex, fair);
        // @audit Oracle fair value used for sizing
        payoutEquivalents[i] = fair;
    }
    totalPayoutEquivalent += payoutEquivalents[i];
}
```

coverage/src/ClaimManager.sol::_computeVaultSlashes()

```
// @audit Over-slash buffer compounds slippage and deviation tolerances
uint256 baseSlash = (tokenStakes[i] * requestedAmount) / totalPayoutEquivalent;
if (tokens[i] != payoutToken) {
    {
        uint256 deviationBps = priceDeviationToleranceBps;
        // slither-disable-next-line divide-before-multiply
        baseSlash = (baseSlash * uint256(_MAX_SLIPPAGE_BPS) * uint256(_MAX_SLIPPAGE_BPS))
            / ((cachedMaxSwapSlippageBps > 0
                    ? uint256(_MAX_SLIPPAGE_BPS) - cachedMaxSwapSlippageBps
                    : uint256(_MAX_SLIPPAGE_BPS))
                * (deviationBps > 0
                    ? uint256(_MAX_SLIPPAGE_BPS) - deviationBps
                    : uint256(_MAX_SLIPPAGE_BPS)));
    }
}
```

With default `maxSwapSlippageBps = 200` (2%) and `priceDeviationToleranceBps` at its maximum of 1000 (10%), the inflation factor is:

$$\frac{10000^2}{(10000 - 200) \times (10000 - 1000)} = \frac{100{,}000{,}000}{88{,}200{,}000} \approx 1.134$$

This results in approximately 13.4% more collateral slashed than the minimum required to fill the beneficiary payout. At the maximum configurable slippage of 3000 bps (30%), the factor reaches approximately 1.587 (58.7% over-slash), though such extreme configurations are unlikely in practice.

The surplus collateral (the difference between what was slashed and what the beneficiary receives) is routed through `_distributeSurplus()` to the `RewardsManager`, which distributes it proportionally back to the vaults backing that committee. Restakers are therefore not permanently deprived of the excess; however, the rewards distribution path operates on a different timescale from the immediate collateral reduction.

The practical effect is that other policies backed by the same vaults experience a temporary reduction in effective coverage. A claim that triggers the 13.4% buffer on a vault also holding collateral for other policies reduces the backing available for those policies by the buffer amount until the rewards cycle returns the surplus. In a realistic scenario with typical parameters, effective coverage could be around 7% less than what was purchased, or as much as 30% less in an extreme high-slippage configuration.

This issue has a low impact because the surplus is not lost but returned to stakers via the rewards path, and the coverage reduction is bounded by admin-configurable parameters and also temporary if the tokens are restaked when returned.

This issue has a low likelihood because it requires a cross-token claim of sufficient size relative to the vault's total stake to produce a meaningful reduction in coverage for co-backed policies.

Recommendations

It would be possible to use the DEX quote for proportional slash sizing while retaining the oracle as a rejection gate for manipulated quotes. This preserves manipulation resistance (quotes outside oracle tolerance are still rejected) while sizing slashes closer to actual execution output, reducing the required buffer to only cover execution slippage between quote and fill. However, it does make the slash sizes vulnerable to DEX manipulation within the configured boundaries.

Another approach would be to use the oracle to determine how much payout token each vault should produce, then retrieve a quote from the DEX for how much collateral is needed to get exactly that amount out. If that passes the oracle sanity check, slash exactly that amount and execute a swap with zero (or minimal) slippage tolerance. This however introduces substantially more complexity especially for edge cases such as partial fills.

Resolution

The development team has closed the issue with the following comment:

> This is a known design trade-off: the slash buffer is intentional, guaranteeing the vault can always cover the swap cost even in adverse market conditions, with any excess flowing to surplus distribution as intended. We considered the exact-output approach Sigma Prime described but found it substantially more complex; partial fills, multi-hop routing, and per-vault collateral accounting introduced edge cases that were difficult to reason about safely. The current design sacrifices some capital efficiency in exchange for simplicity and predictability. In practice, slashes are expected to be infrequent events, which further limits the aggregate impact of the over-slash buffer on restakers.

CYS3-10	Any Account Can Fund Policy Premiums
Assets	coverage/src/PremiumManager.sol coverage/src/interfaces/IPremiumManager.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

Any account can call `PremiumManager.distributePremium()` and fund rewards for an existing bound policy, so offchain systems should not infer that the premium payer is the policy buyer.

`distributePremium()` checks that the policy is bound and that the premium token is approved. It then pulls `amount` from `msg.sender`. There is no requirement that `msg.sender` is the policy buyer, beneficiary, claimer, curator, or pool.

coverage/src/PremiumManager.sol::distributePremium()

```
require(ICoverPool(pool).hasBoundPolicy(policyId), PolicyNotBound(policyId, pool));
// ...
require(_approvedPremiumTokens.contains(premiumToken), TokenNotApproved(premiumToken));

(uint256 platformSplit, uint256 poolSplit, uint256 restakerSplit) =
    getFeeSplits(amount, platformFeeBps, poolFeeBps);

IERC20 token = IERC20(premiumToken);
// @audit Any caller can fund premium distribution for a bound policy.
//        Funds are pulled from msg.sender, not from the policy buyer.
token.safeTransferFrom(msg.sender, address(this), amount);

// ... fee and restaker reward distribution ...

emit PremiumDistributed(pool, policyId, amount);
```

This behaviour appears intentional, as the interface documents `distributePremium()` as permissionless. The main risk is an accounting, monitoring, or analytics mismatch if downstream systems assume each premium payment was made by the buyer. The emitted `PremiumDistributed` event also does not identify the payer.

This issue is rated as informational because third-party premium funding does not directly break policy accounting or allow unauthorised withdrawal of funds. It is a documentation and observability concern.

Recommendations

Ensure documentation, accounting, monitoring, and analytics systems explicitly recognise that any account can fund premiums for a bound policy. Do not infer the payer from the policy buyer unless the payment source is separately verified.

Consider including `msg.sender` in the `PremiumDistributed` event.

Resolution

The payer address is now included in the emitted premium distribution event.

Resolved in coverage PR 103.

CYS3-11	Curator Next-Claim Approval Does Not Bind Claim Details
Assets	coverage/src/ClaimManager.sol coverage/src/interfaces/IClaimManager.sol coverage/src/PolicyManager.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

The curator approval step in `ClaimManager` authorises the next claim attempt for a policy, but it does not approve any specific claim parameters.

This should be documented clearly for integrators, especially where the configured policy claimer is a contract or multisig rather than a single tightly controlled account.

After a payable claim succeeds, `ClaimManager.fileClaim()` sets `_claimApprovalRequired[policyId]` to `true`. Until the flag is cleared, no further claim can be filed for that policy.

coverage/src/ClaimManager.sol::fileClaim()

```
require(!_claimApprovalRequired[policyId], ClaimApprovalRequired(policyId));

// ... main function body

_claimApprovalRequired[policyId] = true;
```

The curator therefore needs to call `approveNextClaim()` first. This function only checks that the caller is the policy's stored curator and then clears the flag. It does not include or bind a requested amount, evidence hash, claimant data, beneficiary, claim identifier, or any other claim-specific fields.

coverage/src/ClaimManager.sol::approveNextClaim()

```
function approveNextClaim(uint96 policyId) external override {
    IPolicyManager.PolicyMetadata memory metadata = IPolicyManager(policyManager).policyMetadata(policyId);
    require(metadata.buyer != address(0), PolicyNotFound(policyId));
    require(msg.sender == metadata.curator, UnauthorizedApprover());
    require(_claimApprovalRequired[policyId], NoApprovalPending(policyId));
    _claimApprovalRequired[policyId] = false;
    emit NextClaimApproved(policyId, msg.sender);
}
```

The next claim details are still supplied by the policy's configured `claimer` when `fileClaim()` is called. `ClaimManager` verifies that `msg.sender` is the stored `metadata.claimer`, but does not know whether that address is an EOA, multisig, automation contract, wrapper, or other contract with its own internal permissions.

coverage/src/ClaimManager.sol::fileClaim()

```
(claimId, metadata) = _fileClaimWithMetadata(policyId, requestedAmount, evidenceHash);
require(msg.sender == metadata.claimer, UnauthorizedClaimer(msg.sender, metadata.claimer));
```

As a result, once the curator has cleared the gate, whichever valid call from the configured claimer reaches `fileClaim()` first determines the claim parameters. If the claimer is a contract with multiple authorised operators, a public trigger path, or a multisig with competing transactions, the curator's approval does not protect against one permitted claimer-side transaction front-running another with different claim details. The submitted claim must still pass the normal policy checks and `Spec` evaluation, so this is not a bypass of claim validation. It is a documentation and integration expectation issue.

This issue is rated informational because the current behaviour appears intentional as a rate-limit or pacing mechanism. The curator is approving that one more claim may be attempted, not attesting to the content of a particular claim. However, unclear documentation could cause pool curators or integrators to overestimate what `approveNextClaim()` protects.

Recommendations

Document that `approveNextClaim()` does not approve a specific claim: it only permits the configured policy claimer to submit the next claim attempt, and any front-running or ordering risk within a contract or multisig claimer remains open to that claimer's own internal controls.

Resolution

Documentation has been updated to clarify that curator approval is a rate-limit gate and does not bind specific claim parameters.

Resolved in coverage PR 104.

CYS3-12	Miscellaneous General Comments
Assets	core/src/contracts/SlashingManager.sol core/src/contracts/StakeManager.sol core/src/contracts/SSPRouter.sol core/src/contracts/Adapters/SymbioticAdapter.sol
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. **`previewSlashing()` function is misnamed and does not preview slashing.**

   The `previewSlashing()` function in `SlashingManager.sol` is misleading and does not fulfil the purpose implied by its name. According to its NatSpec documentation, the function is meant to "preview the per-vault slash distribution without executing slashing". However, the current implementation simply returns vault token stakes (balances) without accepting a slash amount parameter or calculating how a slashing operation would distribute funds across vaults.

   core/src/contracts/SlashingManager.sol::previewSlashing()

   ```
   // @inheritdoc ISlashingManager
   function previewSlashing(uint96 committeeId, address operator)
       external
       view
       returns (address[] memory vaults, address[] memory tokens, uint256[] memory tokenStakes)
   {
       if (committeeId == 0) revert InvalidCommitteeId();
       if (stakeManager == address(0)) revert StakeManagerNotSet();
       if (sspRouter == address(0)) revert InvalidRouter();
       if (operator == address(0)) revert ZeroAddress();
       if (!IStakeManager(stakeManager).isOperatorInCommittee(operator, committeeId)) {
           revert OperatorNotInCommittee();
       }
       // @audit Simply returns vault token stakes, does not preview slashing distribution
       (vaults, tokens, tokenStakes) = ISSPRouter(sspRouter).getCommitteeVaultTokenStakes(committeeId);
   }
   ```

   The function was originally introduced in commit `29f93ea` with a `slashAmountUSD` parameter to calculate vault slash distributions, but was simplified in commit `c6641aa` when USD-denominated accounting was removed. The current implementation essentially provides the same information as `getCommitteeVaultTokenStakes()` from the SSP Router, arguably making the function redundant but certainly rendering its name misleading.

2. **`OperatorNotInCommittee()` error is used for multiple distinct failure states.**

   The `StakeManager`, `SSPRouter`, `SymbioticAdapter`, and `SlashingManager` contracts use `OperatorNotInCommittee()` both when a supplied operator is not the assigned operator for a committee and when a committee has no operator assigned at all. These are related but distinct states: the first indicates a relationship mismatch between a specific operator and a committee, while the second indicates incomplete committee configuration.

   In addition, `StakeManager.isOperatorInCommittee()` reverts with a different error (`OperatorNotRegistered()`) when the operator has never been registered, but returns `false` when a registered operator is simply not assigned to the committee. Callers that translate a `false` result into `OperatorNotInCommittee()` therefore expose different errors depending on whether the operator is globally registered.

   To improve clarity, consider splitting the error into more specific errors such as `CommitteeHasNoOperator()` and `OperatorNotCommitteeOperator()`.

Recommendations

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

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned where appropriate.

Resolved in core PR 539.

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.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

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