CATALYSIS

Coverage Contracts

Security Assessment Report

Version: 2.0

April, 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 Coverage is the coverage layer built on top of the Catalysis protocol. It serves both as a frontend-facing protocol layer and as the integration hub that connects Catalysis products to the rest of the external onchain system.

It manages the full policy lifecycle, from pool creation and policy binding to premium distribution and claim payouts. It also handles external integrations and interactions with the broader DeFi ecosystem.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the 0xcatalysis/coverage repository.

The scope of this time-boxed review was strictly limited to files at commit f31c5f2. The specific files in scope were:

- `/src/ClaimManager.sol`
- `/src/CoverPool.sol`
- `/src/CoverPoolFactory.sol`
- `/src/PolicyManager.sol`
- `/src/PremiumCollector.sol`
- `/src/PremiumManager.sol`
- `/src/SpecRegistry.sol`
- `/src/Swapper.sol`
- `/src/swappers/UniswapV3Adapter.sol`
- `/src/interfaces/*`

The fixes of the identified issues were assessed at commit d8d492a.

Additionally, The changes made in PR 499 in the 0xcatalysis/core repository were assessed at commit bb46e72. The fixes of the identified issues from this PR were assessed at commit 29f93ea

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.

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 16 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 3 issues.
- Medium: 1 issue.
- Low: 2 issues.
- Informational: 9 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
COV-01	Duplicate Claim Prevention Logic Is Invalid	Critical	Resolved
COV-02	`ClaimManager` Cannot Receive Native ETH Payouts	High	Resolved
COV-03	Incorrect ERC20 Handling For Native ETH Collateral In `ClaimManager`	High	Resolved
COV-04	User Can Call `fileClaim()` Without Paying Policy Premiums	High	Resolved
COV-05	Zero-Valued Vault Rewards Can Revert Committee Reward Distribution	Medium	Resolved
COV-06	Malicious Spec Registration Could Cause Cover Pool Drainage	Low	Resolved
COV-07	Unchecked Return Value In Cover Pool Registration	Low	Resolved
COV-08	Ambiguous Semantics Of `committeeStakeRequirement`	Informational	Resolved
COV-09	ERC20 Validation Uses Optional `decimals()`	Informational	Resolved
COV-10	Evidence Hash Lacks Structure And Validation	Informational	Resolved
COV-11	Hardcoded Zero Slippage Protection In `UniswapV3Adapter`	Informational	Resolved
COV-12	Missing Access Control In `UniswapV3Adapter`	Informational	Resolved
COV-13	Revert When `wadToSlash` Rounds To Zero	Informational	Resolved
COV-14	Rounding Dust Accumulates In `EigenAdapter` During Reward Distribution	Informational	Resolved
COV-15	Stake Validation Against Incorrect Coverage Value	Informational	Resolved
COV-16	Miscellaneous General Comments	Informational	Resolved

COV-01	Duplicate Claim Prevention Logic Is Invalid		
Assets	`ClaimManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The duplicate claim prevention mechanism fails to prevent actual duplicate claims, allowing users to submit identical loss events multiple times and drain coverage limits through repeated payouts for the same underlying loss.

The `_fileClaimWithMetadata()` function attempts to prevent duplicate claims using a claim ID-based check:

ClaimManager.sol::_fileClaimWithMetadata()

```
280 claimId = _policyClaims[policyId].claimCount;
    ClaimRecord storage claim = _claims[policyId][claimId];
282 require(claim.status == ClaimStatus.None, ClaimAlreadyFiled(policyId, claimId));
```

However, further down in the function, after passing the duplicate check, the function increments the claim count:

ClaimManager.sol::_fileClaimWithMetadata()

```
290 _policyClaims[policyId].claimCount++;
```

This increment ensures that subsequent claim submissions will receive new, unused claim IDs, causing the duplicate check to pass every time since `claim.status == ClaimStatus.None` for all new claim IDs.

Consider Bob with a $30 million coverage policy who suffers a legitimate $1 million loss:

1. First Claim: Bob submits claim for $1M loss → assigned claimId = 0 → duplicate check passes (new claim ID) → claim approved and paid
2. Duplicate Claim: Bob resubmits identical $1M claim → assigned claimId = 1 → duplicate check passes (new claim ID) → claim approved and paid
3. Repeated Exploitation: Bob continues submitting the same $1M loss → receives new claim IDs each time → all duplicate checks pass

Bob can repeat this process 30 times, claiming the full $30 million coverage limit despite only suffering a $1 million actual loss.

This vulnerability allows malicious users to completely drain policy coverage limits by filing the same legitimate claim multiple times. Since each submission receives a unique claim ID, the current duplicate prevention mechanism provides no protection against repeated exploitation of single loss events.

This issue has a high impact as it enables complete drainage of coverage limits through repeated payouts for a single loss event. This issue has a high likelihood as any policyholder with a single legitimate claim can exploit it with no special conditions or setup required.

Recommendations

The testing team recognises that the intended autonomous nature of claiming results in there being no way to identify what constitutes the “same loss event” because the evidence is unstructured and users have control over the content of their self-reported claims. Nevertheless, a reliable safeguard against duplicate claims is essential.

One possible fix is to implement a curator approval mechanism for users who claim more than once. This would reduce the autonomous nature of claiming but ensure an attacker cannot process duplicate claims and drain the cover pool funds.

Resolution

The development team has fixed this issue in PR 58. The function `fileClaim()` now checks the status of the new mapping `_claimApprovalRequired` and updates it to `true` after a successful claim. Only the pool curator can change to status the `_claimApprovalRequired` via the new function `approveNextClaim()`.

COV-02	`ClaimManager` Cannot Receive Native ETH Payouts		
Assets	`ClaimManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

`ClaimManager` is intended to handle native ETH, but it lacks either a `receive()` function or a `payable` fallback. This creates an issue in its interaction with `Swapper`.

`Swapper` explicitly supports native ETH output and sends it to the configured recipient via a low-level call when `tokenOut` is `NATIVE_ETH` in `_handleTokenOutput()`:

Swapper.sol::_handleTokenOutput()

```
function _handleTokenOutput(address tokenOut, uint256 amountOut, address recipient) private {
    if (tokenOut == NATIVE_ETH) {
        IWETH(nativeWrapper).withdraw(amountOut);
        (bool sent,) = recipient.call{value: amountOut}("");
        require(sent, NativeTransferFailed());
    } else {
        // ...
    }
}
```

In `_processCollateral()`, `ClaimManager` sets itself as the recipient of these swaps.

Because `ClaimManager` cannot accept plain ETH transfers, any swap that resolves to native ETH with a `recipient` of `address(this)` will revert when `Swapper` attempts to deliver the proceeds. This makes the native ETH payout path unusable and can block settlement of otherwise valid approved claims.

This issue has a high impact as it causes approved claim payouts to revert, blocking beneficiaries from receiving funds through the native ETH path. This issue has a medium likelihood as it requires a claim flow that uses native ETH as the payout asset, causing `Swapper` to execute with `tokenOut == NATIVE_ETH` and attempt to deliver ETH to `ClaimManager`. Native ETH output is explicitly supported by `Swapper`, but only affects deployments that choose to use that payout path.

Recommendations

Add a `receive()` function to `ClaimManager`.

Resolution

The `receive()` function has been added to the contract `ClaimManager` in PR 59.

COV-03	Incorrect ERC20 Handling For Native ETH Collateral In `ClaimManager`		
Assets	`ClaimManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When processing slashed collateral, `ClaimManager._processCollateral()` unconditionally executes ERC20 `approve()` and `transfer()` operations, even when the collateral is `NATIVE_ETH`. This is invalid as the `NATIVE_ETH` address is not an `ERC20` token and cannot be approved or transferred with `IERC20` methods.

ClaimManager.sol::_processCollateral()

```
if (collateralToken == payoutToken) {
    uint256 transferAmount = collateralAmount > remainingPayout ? remainingPayout : collateralAmount;
    IERC20(collateralToken).safeTransfer(beneficiary, transferAmount); // @audit doesn't handle NATIVE_ETH case
    amountOut = transferAmount;
} else {
    IERC20(collateralToken).forceApprove(swapper, collateralAmount);
```

As a result, the claim settlement logic is incompatible with native ETH collateral. Any slashing result that uses `NATIVE_ETH` will cause the payout path to revert, preventing settlement of otherwise valid claims.

This issue has a high impact as it blocks claim settlement for native ETH collateral, preventing beneficiaries from receiving approved payouts. This issue has a medium likelihood as it requires the slashing flow to return native ETH as collateral. That is a supported asset type in the broader settlement design, but it is a narrower configuration than the default ERC20 collateral path. When it does occur, the bug is triggered immediately because `ClaimManager` treats `NATIVE_ETH` as an `IERC20` in both the direct-transfer and swap branches.

Recommendations

Add explicit native ETH handling in `ClaimManager` so `NATIVE_ETH` collateral bypasses `ERC20` approval logic and is passed through a dedicated native-value path instead of being cast to `IERC20`.

Resolution

This issue has been fixed in PR 59. The development team has introduced two new internal functions `_transferToken()` and `_approveIfERC20()` to handle both ERC20 tokens and native ETH. These functions are correctly used within `_processCollateral()`.

COV-04	User Can Call `fileClaim()` Without Paying Policy Premiums		
Assets	`ClaimManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Users with bound policies can file valid claims even after defaulting on premium payments, allowing unpaid policyholders to receive payouts.

The `fileClaim()` function in `ClaimManager.sol` processes claims autonomously without validating premium payment status. When users file claims, the system evaluates claim validity exclusively through the cover's respective spec contract:

ClaimManager.sol::fileClaim()

```
190 ISpec.EvaluationResult memory result = _evaluateSpec(policyId, metadata, evidenceHash, additionalData);
```

Claims are approved or rejected based solely on the spec evaluation result. If the evaluation returns false, the claim is rejected:

ClaimManager.sol::fileClaim()

```
195 if (!result.isPayable) {
        claim.status = ClaimStatus.Rejected;
197     emit ClaimRejected(policyId, claimId, "Spec evaluation failed", block.timestamp);
        return claimId;
199 }
```

If the evaluation returns true, the system automatically processes the payout:

ClaimManager.sol::fileClaim()

```
201 claim.status = ClaimStatus.Approved;

203 uint256 totalPayout = _capAndExecutePayout(policyId, claimId, metadata, requestedAmount);
    require(totalPayout > 0, ZeroPayoutAmount(policyId));
```

Since premium payments occur offchain while claims processing is autonomous onchain, the contract cannot verify whether users are current on their premium obligations. This allows defaulted users to successfully submit claims and receive full payouts despite having stopped premium payments, effectively granting free coverage at the expense of cover pools.

This issue has a high impact as it allows unpaid policyholders to drain cover pool funds through payouts they are not entitled to. This issue has a medium likelihood as exploitation requires a user to hold a bound policy with a valid spec evaluation, but no premium payment enforcement exists to prevent it once those conditions are met.

Recommendations

Implement a premium status validation mechanism within the autonomous claim process by adding a `premiumDefaulted` boolean flag to the `PolicyMetadata` struct. This flag should be controlled by the policy curator and checked during claim validation to prevent payouts to users in premium default while preserving the autonomous nature of legitimate claim processing.

Resolution

The development team has fixed this issue in PR 62. A new bool field `premiumDefaulted` has been added to the `PolicyMetadata` struct. This field is set to `false` in the function `bindPolicy()`. Only the function `Policy.setPremiumDefaulted()` can update the value of the field and this can only be called by the function `CoverPool.setPremiumDefaulted()`.

COV-05	Zero-Valued Vault Rewards Can Revert Committee Reward Distribution		
Assets	`EigenAdapter.sol` `SSPRouter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Small per-vault reward allocations can cause `EigenAdapter._distributeRewards()` to construct invalid EigenLayer reward submissions, causing reward distribution to revert for the entire committee.

`EigenAdapter._distributeRewards()` first counts every vault with positive stake and allocates both `strategiesAndMultipliers` and `operatorRewards` to that `vaultCount`. However, inside the main distribution loop, the function reserves an array slot for every positive-stake vault before it knows whether that vault's reward remains non-zero after rounding.

EigenAdapter.sol::_distributeRewards()

```
if (vaultStakesUSD[i] > 0) {
    strategiesAndMultipliers[index] = IRewardsCoordinatorTypes.StrategyAndMultiplier({
        strategy: IStrategy(eigenVaults[i]),
        multiplier: uint96(1e18)
    }); // @audit slot is reserved before the reward is known to be non-zero

    uint256 vaultReward = (amount * vaultStakesUSD[i]) / totalStakeUSD;
    if (vaultReward > 0) {
        uint256 vaultRewardAmount = _convertUSDToTokenAmount(address(rewardToken), vaultReward);
        totalRewardTokenAmount += vaultRewardAmount;
        operatorRewards[index] =
            IRewardsCoordinatorTypes.OperatorReward({operator: eigenVaults[i], amount: vaultRewardAmount});
    }
    index++; // @audit advances even when the operator reward entry is invalid
}
```

This creates two failure modes. If `vaultReward == 0` due to the USD pro-rata split rounding down, the corresponding `operatorRewards[index]` entry remains default-initialised as `{operator: address(0), amount: 0}`. If `vaultReward > 0` but the subsequent USD-to-token conversion rounds down to zero, the entry becomes `{operator: eigenVaults[i], amount: 0}`.

The second case is realistic because `ChainlinkPriceFeed.getTokenAmount()` floors the conversion:

ChainlinkPriceFeed.sol::getTokenAmount()

```
112 return (usdValue * (10 ** tokenDecimals)) / uint256(price);
```

For example, with a 6-decimal reward token priced at 1 USD, a `vaultReward` of `1` (0.00000001 USD in 8-decimal precision) converts to zero token units.

EigenLayer's `RewardsCoordinator` rejects both forms of invalid entry during submission validation:

RewardsCoordinator.sol::_validateOperatorDirectedRewardsSubmission()

```
659 // Check that each operator is a non-zero address.
    require(submission.operatorRewards[i].operator != address(0), InvalidAddressZero());
661 // Check that each operator is in ascending order.
    require(lastOperator < submission.operatorRewards[i].operator, OperatorsNotInAscendingOrder());
663 // Check that each operator reward amount is non-zero.
    require(submission.operatorRewards[i].amount > 0, AmountIsZero());
```

As a result, `createOperatorDirectedOperatorSetRewardsSubmission()` reverts, `EigenAdapter` rethrows `RewardDistributionFailed()`, and the surrounding `SSPRouter.distributeRewards()` transaction reverts in full. This blocks reward distribution for the committee rather than merely skipping the affected vault.

This issue has a medium impact as it can prevent reward settlement for an entire committee and revert the router-level distribution transaction. This issue has a medium likelihood as it only requires a committee with at least one low-share vault and a reward size or token decimal configuration that causes a per-vault reward to round to zero, which is plausible for small reward distributions.

Recommendations

Build the EigenLayer submission arrays from only the vaults whose converted token reward is strictly positive. Do not reserve an array slot or advance the final submission index until `vaultRewardAmount > 0` is known. If all per-vault rewards round down to zero after conversion, short-circuit the EigenLayer reward path instead of submitting an invalid `operatorRewards` array. Add regression tests covering both `vaultReward == 0` and `vaultRewardAmount == 0` cases.

Resolution

This issue has been addressed in PR 504. `SSPRouter` now converts the committee's USD reward into token units once, then transfers tokens to each adapter. Each adapter splits that token amount across vaults while skipping zero allocations, and `EigenAdapter` skips the EigenLayer submission entirely when every per-vault amount rounds to zero.

COV-06	Malicious Spec Registration Could Cause Cover Pool Drainage		
Assets	`SpecRegistry.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Cover pool curators can register spec contracts through the `SpecRegistry` without external validation or oversight. The spec registration process allows curators to deploy and register any contract implementing the `ISpec` interface. Once registered, these specs have complete control over claim approval decisions through the `evaluate()` function. The `ClaimManager.sol` trusts spec evaluation results unconditionally:

ClaimManager.sol::fileClaim()

```
ClaimRecord storage claim = _claims[policyId][claimId];
ISpec.EvaluationResult memory result = _evaluateSpec(policyId, metadata, evidenceHash, additionalData);

// ... snip ...

if (!result.isPayable) {
    claim.status = ClaimStatus.Rejected;
    emit ClaimRejected(policyId, claimId, "Spec evaluation failed", block.timestamp);
    return claimId;
}
```

This vulnerability allows cover pool curators to completely drain operator collateral through fabricated claims. Since spec evaluation is the sole determinant of claim validity and curators control spec implementation, operators have no protection against malicious curators who can approve any claim they submit.

Recommendations

Implement spec validation and governance mechanisms such as:

- Multi-signature approval requirements for spec registration
- Time-delayed spec activation with challenge periods

Resolution

The development team has fixed this issue in PR 63 by introducing a new approval requirement in the contract `SpecRegistry`. A spec now cannot be registered in the function `registerSpec()` unless is it approved by the admin. The approval can be done by the function `approveSpec()`.

COV-07	Unchecked Return Value In Cover Pool Registration		
Assets	`CoverPoolFactory.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `createCoverPool()` function does not validate the return value when adding newly created pools to the registry, potentially resulting in unusable pools that cannot interact with the protocol.

When creating cover pools, the factory adds the new pool address to an internal registry without checking if the operation succeeded:

CoverPoolFactory.sol::createCoverPool()

```
108 _pools.add(coverPool);
```

The `EnumerableSet.add()` function returns `false` if the element already exists in the set. If this occurs, the pool creation appears successful (it emits the `CoverPoolCreated` event) but the pool is not properly registered.

This results in pools that exist onchain but cannot accept policies, register specs, or collect premiums.

Recommendations

Validate the return value and revert if pool registration fails:

```
bool added = _pools.add(coverPool);
require(added, "Pool registration failed");
```

Resolution

The recommendation has been implemented in PR 64.

COV-08	Ambiguous Semantics Of `committeeStakeRequirement`
Assets	`SSPRouter.sol` `Adapters/EigenAdapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `committeeStakeRequirement` state variable is documented as both a minimum delegation threshold and a maximum stake limit, but it is applied per vault rather than as an aggregate committee cap.

Because `createCommittee()` deploys a separate `DurationVaultStrategy` for each acceptable token, each with the full `stakeRequirement` as its `stakeCap`, the effective maximum deposit capacity for a committee is `stakeRequirement * N`, where `N` is the number of acceptable tokens. There is no mechanism to enforce an aggregate maximum across all vaults belonging to a committee.

The NatSpec on `committeeStakeRequirement` describes it as serving dual purposes:

SSPRouter.sol

```
60  /// @notice Stake requirement for a committee in USD.
    /// @dev This value serves two purposes:
62  /// 1. Minimum stake an operator must delegate to participate in the committee.
    /// 2. Maximum stake limit enforced in external staking modules (e.g., Symbiotic network limit).
64  mapping(uint96 => uint256) public committeeStakeRequirement;
```

In `createCommittee()`, a vault is deployed for every acceptable token, each receiving the full `stakeRequirement` as its cap:

SSPRouter.sol::createCommittee()

```
304 for (uint256 i = 0; i < acceptableTokens.length(); i++) {
        address token = acceptableTokens.at(i);
306     address durationVault = address(
            IEigenAdapter(eigenAdapter).deployDurationVaultStrategy(
308             committeeId, token, stakeRequirement, duration
            )
310     );

312     // Register the deployed vault with EIGENLAYER module type
        _registerVaultModule(durationVault, ISSPRouter.SSPModuleType.EIGENLAYER);
314 }
```

Inside `EigenAdapter.deployDurationVaultStrategy()`, the USD-denominated `stakeRequirement` is converted to a token amount and used as both `maxPerDeposit` and `stakeCap`:

EigenAdapter.sol::deployDurationVaultStrategy()

```
uint256 tokenAmount = _convertUSDToTokenAmount(underlyingToken, maxStakeForCommittee);
// ... snip ...
maxPerDeposit: tokenAmount,
stakeCap: tokenAmount,
```

The same pattern applies on the Symbiotic side, where `_configureSymbioticVaults()` sets each vault's network limit to the full `stakeRequirement`:

SSPRouter.sol::_configureSymbioticVaults()

```
760 for (uint256 i = 0; i < symbioticCount; i++) {
        ISymbioticAdapter(symbioticAdapter).setMaxNetworkLimit(
762         symbioticVaults[i], committeeId, stakeRequirement
        );
764 }
```

However, the delegation check in `_checkDelegation()` aggregates stake across all vaults and only requires the total to meet the single `committeeStakeRequirement`:

SSPRouter.sol::_checkDelegation()

```
830 uint256 totalDelegationAmountUSD = eigenStakeUSD + symbioticStakeUSD;
    if (totalDelegationAmountUSD < committeeStakeRequirement[committeeId]) {
832     return false;
    }
```

With N acceptable tokens, a committee can accept up to N times its stated `committeeStakeRequirement` in total deposits, while only requiring `committeeStakeRequirement` to pass the delegation check. Anyone reasoning about committee-level economic exposure based on `committeeStakeRequirement` would underestimate the actual maximum by a factor of N. Committees could end up heavily over-collateralised relative to the stated requirement, with capital locked in duration vaults beyond what is necessary.

This issue is rated as informational as the over-collateralisation does not cause direct loss of funds and requires multiple acceptable tokens to be registered, which is an admin-controlled configuration.

Recommendations

Consider clarifying the intended semantics of `committeeStakeRequirement`. If it is meant as a per-vault cap, update the NatSpec to reflect that the aggregate committee capacity scales with the number of tokens. If it is meant as a committee-level cap, consider dividing it across vaults or enforcing an aggregate limit.

Resolution

This issue has been addressed in PR 509. The NatSpec for `committeeStakeRequirement` has been clarified to describe it as a per-vault USD deposit cap and minimum aggregate delegation threshold, with expanded documentation on how the cap is applied across EigenLayer and Symbiotic paths.

COV-09	ERC20 Validation Uses Optional `decimals()`
Assets	`SSPRouter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `registerAcceptableTokens()` function in `SSPRouter` uses a `decimals()` static call as a proxy for validating that an address implements ERC20, but this check is neither sufficient nor necessary for that purpose.

The function performs the following validation on each token address:

SSPRouter.sol::registerAcceptableTokens()

```
213 // Verify it implements ERC20 by checking decimals() function
    (bool success, bytes memory data) = token.staticcall(abi.encodeWithSignature("decimals()"));
215 if (!success || data.length == 0) revert InvalidTokenAddress();
```

The `decimals()` function is strictly optional in the ERC20 specification (EIP-20). A compliant ERC20 token is not required to implement it, meaning a legitimate token could be rejected by this check. Conversely, `decimals()` is present on non-ERC20 contracts: notably Chainlink's `AggregatorV3Interface` exposes a `decimals()` function, as do other token standards. Passing this check therefore does not guarantee the address is a valid ERC20 token.

This issue is rated as informational as the function is admin-gated via `onlyRole(DEFAULT_ADMIN_ROLE)`, and in practice the admin is expected to register only known, vetted tokens. The `decimals()` check serves as a reasonable sanity check for the common case, but it cannot substitute for manual verification of the token contract.

Recommendations

Consider documenting the limitation that the `decimals()` check is a heuristic rather than a definitive ERC20 validation. Ultimately, the correctness of registered tokens depends on offchain due diligence by the admin. If stronger onchain validation is desired, consider checking for the presence of core ERC20 functions such as `balanceOf()` or `totalSupply()`, though no single static call can fully guarantee ERC20 compliance.

Resolution

This issue has been addressed in PR 505. Token registration now validates contract code existence and calls `totalSupply()` instead of relying solely on the optional `decimals()` function for ERC20 validation. As mentioned above, there is no perfect solution to this problem, but this check adheres more closely to the token specification.

COV-10	Evidence Hash Lacks Structure And Validation
Assets	`ClaimManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When filing claims through `fileClaim()`, users provide an `evidenceHash` parameter intended to reference supporting evidence. The system performs only a basic non-zero validation:

ClaimManager.sol::_fileClaimWithMetadata()

```
262 require(evidenceHash != bytes32(0), InvalidEvidenceHash());
```

The interface documentation suggests using IPFS CIDs, but this is not enforced:

interfaces/IClaimManager.sol::fileClaim()

```
301 // @param evidenceHash Hash of evidence supporting the claim (e.g., IPFS CID).
```

In practice, users can submit any bytes32 value, as demonstrated in test cases using simple string hashes such as `keccak256("evidence-hash")`.

While this does not directly affect claim processing or security, it undermines the integrity of the evidence tracking system. Users could submit meaningless hashes, duplicate evidence across claims, or provide no actual evidence reference, making post-claim auditing and dispute resolution more difficult.

Recommendations

Consider implementing evidence hash format validation or requiring specific hash structures (e.g., IPFS CID format) to ensure evidence hashes represent actual, retrievable evidence documents.

Resolution

The development team has fixed this issue in PR 65 by checking the uniqueness of an evidence hash per policyId in the function `fileClaim()` using the new mapping `_usedEvidenceHashes`.

COV-11	Hardcoded Zero Slippage Protection In `UniswapV3Adapter`
Assets	`UniswapV3Adapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `UniswapV3Adapter.swap()` function hardcodes `amountOutMinimum` and `sqrtPriceLimitX96` to zero when calling the Uniswap V3 router, disabling all router-level slippage protection.

swappers/UniswapV3Adapter.sol::swap()

```
IV3SwapRouter.ExactInputSingleParams memory params = IV3SwapRouter.ExactInputSingleParams({
    tokenIn: tokenIn,
    tokenOut: tokenOut,
    fee: fee,
    recipient: msg.sender,
    amountIn: amountIn,
    amountOutMinimum: 0, // @audit hardcoded to zero
    sqrtPriceLimitX96: 0 // @audit hardcoded to zero
});
```

In the current integration path, slippage is enforced at the `Swapper` layer: `Swapper.executeSwap()` computes a Chainlink-derived `amountOutMin` and reverts if the final output falls below it, so the hardcoded zeros in the adapter do not create an immediate exploit vector.

However, hardcoding these safety parameters to their most dangerous values is a fragile pattern. If the adapter is reused in a different context, integrated by a future contract that does not enforce its own slippage check, or if the `Swapper` layer logic changes, the zero values silently remove all sandwich-attack protection at the router level.

This issue is rated as informational as the current call path enforces slippage externally, but the adapter violates defence-in-depth best practices by unconditionally disabling the router's built-in protections.

Recommendations

Consider accepting an `amountOutMinimum` parameter in `swap()` and forwarding it to the Uniswap router call, so that slippage protection is enforced at both the adapter and caller levels.

Resolution

This issue has been fixed in PR 66. `UniswapV3Adapter.swap()` now takes an `amountOutMinimum` argument and forwards it into the Uniswap V3 router call `exactInputSingle()` by setting `ExactInputSingleParams.amountOutMinimum` to that same value.

COV-12	Missing Access Control In `UniswapV3Adapter`
Assets	`UniswapV3Adapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `UniswapV3Adapter.swap()` function lacks access control, allowing any external account to execute swaps through the adapter.

The adapter is designed to be called exclusively by the `Swapper` contract as part of the claim settlement flow. However, nothing restricts who may call the `swap()` function:

swappers/UniswapV3Adapter.sol::swap()

```
function swap(address tokenIn, address tokenOut, uint24 fee) external returns (uint256 amountOut) {
    uint256 amountIn = IERC20(tokenIn).allowance(msg.sender, address(this));
    require(amountIn != 0, ZeroAllowance());

    IERC20(tokenIn).safeTransferFrom(msg.sender, address(this), amountIn);
    // ...
}
```

Because the adapter resolves `amountIn` from `allowance(msg.sender, address(this))` and pulls tokens via `safeTransferFrom(msg.sender, ...)`, a third-party caller can only spend tokens they have themselves approved to the adapter. This limits the practical attack surface; however, unrestricted access still permits unintended usage of the adapter outside its designed integration path with `Swapper`.

This issue is rated as informational as there is no direct exploit path given the allowance-based design, but the lack of access control deviates from the principle of least privilege.

Recommendations

Consider restricting `swap()` to only the `Swapper` contract by storing the authorised caller at construction time and enforcing it with a `require` check.

Resolution

The access control has been added in PR 67.

The function `UniswapV3Adapter()` can only be called by the `Swapper` contract now.

COV-13	Revert When `wadToSlash` Rounds To Zero
Assets	`EigenAdapter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In `EigenAdapter._executeSlashing()`, the `wadToSlash` calculation can round down to zero when a non-zero slash amount is extremely small relative to the vault's total USD stake, causing the downstream call to `AllocationManager.slashOperator()` to revert.

The proportion to slash is computed as:

EigenAdapter.sol::_executeSlashing()

```
528 uint256 totalStake = _getStrategiesStakeUSD(singleVaultArr);
530 // Skip if vault has no stake - can't slash with wadToSlash = 0
    if (totalStake == 0) {
532     continue;
    }
534
    uint256 wadToSlash = (slashAmount * 1e18) / totalStake;
```

When `slashAmount * 1e18 < totalStake`, integer division truncates `wadToSlash` to zero. Both `slashAmount` and `totalStake` are USD values with 8 decimals, so this threshold is much more extreme than it would be for 18 decimal values. For example, a vault with `totalStake = 2e18` (20 billion USD in 8-decimal precision) and `slashAmount = 1` (0.00000001 USD) would produce `wadToSlash = 0`. By contrast, a 1 million USD vault corresponds to `totalStake = 1e14`, so even the minimum non-zero slash unit would still produce a positive `wadToSlash`.

While there is an existing guard for `totalStake == 0`, there is no equivalent check for the computed `wadToSlash`. EigenLayer's `AllocationManager` validates that each wad is strictly positive:

AllocationManager.sol::_slashOperator()

```
428 require(0 < params.wadsToSlash[i] && params.wadsToSlash[i] <= WAD, InvalidWadToSlash());
```

A zero `wadToSlash` would therefore cause the entire slashing call to revert with `InvalidWadToSlash()`. Because `EigenAdapter` reverts on any failed slashing call, this would revert the EigenLayer slashing path for that transaction.

However, reaching this condition requires an extremely small slash relative to a very large USD-denominated stake, making it a narrow edge case rather than a realistic concern for ordinary slash sizes.

This issue is rated as informational as the conditions required are extreme, the slashing parameters are supplied by a privileged role, and the impact is limited to an unlikely edge case.

Recommendations

Consider adding an explicit `wadToSlash == 0` check after the calculation and either skipping the vault or reverting with a dedicated custom error. This would make the edge case explicit.

Resolution

This issue has been addressed in PR 506. `EigenAdapter` now explicitly reverts with a `ZeroWadToSlash` error when the computed `wadToSlash` rounds to zero, avoiding a zero-value submission to EigenLayer's `AllocationManager`.

COV-14	Rounding Dust Accumulates In `EigenAdapter` During Reward Distribution
Assets	`Adapters/EigenAdapter.sol` `SSPRouter.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Reward distribution involves multiple layers of integer division, each rounding down independently. On the EigenLayer path, this can leave a small balance of reward tokens behind in the `EigenAdapter` after a distribution.

When `SSPRouter.distributeRewards()` processes the EigenLayer portion, it converts the USD reward amount to tokens in a single bulk conversion and transfers the result to the `EigenAdapter`:

SSPRouter.sol::distributeRewards()

```
uint256 eigenReward = (rewardAmount * stakes.eigenStake) / stakes.totalStake;
if (eigenReward > 0) {
    uint256 eigenRewardTokenAmount = _convertUSDToTokenAmount(address(rewardToken), eigenReward);
    rewardToken.safeTransferFrom(tokenSource, eigenAdapter, eigenRewardTokenAmount);
    // ... snip ...
}
```

Inside `EigenAdapter._distributeRewards()`, the USD amount is then split across individual vaults proportionally, and each vault's share is independently converted from USD to tokens:

EigenAdapter.sol::_distributeRewards()

```
uint256 vaultReward = (amount * vaultStakesUSD[i]) / totalStakeUSD;
if (vaultReward > 0) {
    uint256 vaultRewardAmount = _convertUSDToTokenAmount(address(rewardToken), vaultReward);
    totalRewardTokenAmount += vaultRewardAmount;
    // ... snip ...
}
```

Only `totalRewardTokenAmount` is approved and forwarded to the `RewardsCoordinator`. Because each per-vault USD-to-token conversion rounds down independently, the sum of the per-vault token amounts is less than or equal to the bulk token amount that was transferred in. The difference remains in the `EigenAdapter` rather than being forwarded as part of that reward distribution. The contract already exposes `recoverStuckTokens()`, so this dust is not permanently stuck, but it is not automatically reused by the reward flow.

A similar rounding loss occurs one level up in `SSPRouter.distributeRewards()`, where the split between Symbiotic and EigenLayer rewards rounds down, meaning `symbioticReward + eigenReward` may be less than `rewardAmount`. That higher-level remainder stays with `tokenSource` because only the rounded module amounts are transferred to the adapters.

While the dust per transaction is negligible, it can accumulate over many reward distributions as an operational nuisance.

This issue is rated as informational as the per-transaction amount is extremely small, does not affect reward correctness, and can already be recovered by an admin.

Recommendations

Consider assigning the remainder to the last vault in the EigenLayer distribution loop so the full transferred amount is forwarded to the `RewardsCoordinator`. If the current behaviour is acceptable, consider documenting that a small residual token balance can remain in `EigenAdapter` and may need to be recovered periodically via the existing `recoverStuckTokens()` function.

Resolution

This issue has been addressed in PR 507. `SSPRouter` now converts each module's USD share to a single token amount and transfers it to the adapter, which splits the exact token amount pro-rata by USD stake, eliminating the per-vault USD-to-token conversion step that caused rounding dust accumulation.

COV-15	Stake Validation Against Incorrect Coverage Value
Assets	`PolicyManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `bindPolicy()` function validates committee stake against the user's requested coverage amount rather than the actual coverage limit that will be stored and used for claim payouts, creating potential for undercollateralised policies.

During policy binding, the system validates that sufficient stake exists to back the policy:

PolicyManager.sol::bindPolicy()

```
251 uint256 totalStake = IStakeManager(stakeManager).getCommitteeTotalStakeUSD(policyId);
    require(totalStake >= request.coverageAmount, InsufficientStake(policyId, request.coverageAmount, totalStake));
```

However, the actual coverage limit stored in the policy metadata comes from the pool's `boundPolicy` parameter:

PolicyManager.sol::bindPolicy()

```
IPolicyManager.PolicyMetadata memory metadata = IPolicyManager.PolicyMetadata({
    // ... other fields ...
    coverageLimit: boundPolicy.coverageLimit,  // This value is used for claims
    // ... other fields ...
});
```

This creates a validation gap where stake is verified against the user's request (`coverageAmount`) but claims are paid against the pool's limit (`coverageLimit`). While pools cannot currently modify coverage limits after binding, future system upgrades that enable coverage limit modifications could create undercollateralised policies where actual coverage exceeds the validated collateral backing.

Recommendations

Validate stake against the actual coverage limit that will be used for claim processing:

```
require(totalStake >= boundPolicy.coverageLimit, InsufficientStake(policyId, boundPolicy.coverageLimit, totalStake));
```

This ensures collateral always matches the policy's actual exposure and provides defence-in-depth against potential future vulnerabilities related to coverage limit modifications.

Resolution

This issue has been fixed in PR 68 by checking the `totalStake` against the equivalent USD value of `boundPolicy.coverageLimit` .

COV-16	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. **Misleading Error Name**

   Related Asset(s): PremiumCollector.sol

   The `registerVault()` function reverts with a `ZeroAmount()` error when `policyId == 0`. This error is misleading as the value being checked is a policy identifier, not an amount.

   Consider using a more descriptive error such as `InvalidPolicyId()`.

2. **Token Registration Events And Errors Lack Address Context**

   Related Asset(s): SSPRouter.sol, ISSPRouter.sol

   The events emitted by `registerAcceptableTokens()` and `removeAcceptableTokens()` (`TokensRegistered()`, `TokensRemoved()`) carry no parameters, and the associated errors (`TokenAlreadyRegistered()`, `TokenNotRegistered()`, `InvalidTokenAddress()`) do not identify which token caused the failure. When a multi-token registration reverts, the caller receives no indication of the offending token, and offchain tooling must re-derive registered tokens from calldata.

   Consider adding the token address to each error (e.g. `error TokenAlreadyRegistered(address token)`) and emitting a per-token event within the loop.

3. **Redundant `IStrategy` Cast In `_getStrategiesStakeUSD()`**

   Related Asset(s): EigenAdapter.sol

   In `EigenAdapter._getStrategiesStakeUSD()`, the `vault` address is cast to `IStrategy` to call `totalShares()`, but a second `IStrategy` variable is created inside the subsequent `if` block rather than reusing the first cast.

   Consider declaring the `IStrategy strategy` variable once at the start of the loop body and reusing it for all calls.

4. **USD-Denominated Variables Lack Unit Indicators**

   Related Asset(s): SSPRouter.sol, EigenAdapter.sol, SymbioticAdapter.sol, SlashingManager.sol, RewardsManager.sol, StakeManager.sol, IBaseAdapter.sol, ISymbioticAdapter.sol

   Many variables, parameters, and struct fields hold USD-denominated values (8-decimal Chainlink standard) but are not named to indicate this, creating ambiguity with token-native units. Some internal functions already use the `USD` suffix consistently (e.g. `totalStakeUSD`, `slashAmountUSD`), which makes the unmarked cases more confusing by contrast.

   The most problematic instance is the `collateralAmounts` return value from `executeSlashing()` on `IBaseAdapter` and `ISSPRouter`: both adapters convert slashed token amounts back to USD before returning, but the name and NatSpec ("amounts slashed for each collateral token") imply token-native units.

   Affected variables span the full call chain across multiple contracts.

   - `SSPRouter`: `committeeStakeRequirement`, `stakeRequirement`, `rewardAmount`, `symbioticReward`, `eigenReward`, `totalNetAmount`, and `totalSlashAmount`.

- `EigenAdapter`: `rewardAmount`, the `amount` parameter in `_distributeRewards()`, `vaultReward`, `slashAmount`, and `totalStake` (the result of `_getStrategiesStakeUSD()`).
- `SymbioticAdapter`: `rewardAmount`, the `amount` parameter in `setMaxNetworkLimit()`, `totalRewardAmount`, `vaultRewardAmount`, and `actualSlashedAmounts`.
- `SlashingManager`: the `slashAmount` parameter in `executeSlashing()` and `vaultSlashAmount`.
- `RewardsManager`: `rewardAmount` and `netAmount`.
- `StakeManager`: the `stakeRequirement` parameter in `createCommittee()`.
- The struct fields `VaultSlashing.slashAmount` and `VaultReward.amount` propagate the ambiguity across contract boundaries.

Consider adopting a consistent `USD` suffix for all USD-denominated values, and in particular renaming `collateralAmounts` or updating its NatSpec to reflect the actual unit.

Recommendations

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

Resolution

The development team's responses to the raised issues above are as follows.

1. The error `InvalidPolicyId` is used now as recommended in PR 69.
2. Token registration and removal events are now per-token and errors include the offending token address, as recommended in PR 508.
3. The redundant `IStrategy` cast has been cleaned up in PR 508.
4. USD-denominated variables have been renamed with a `USD` suffix throughout the codebase in PR 508.

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.

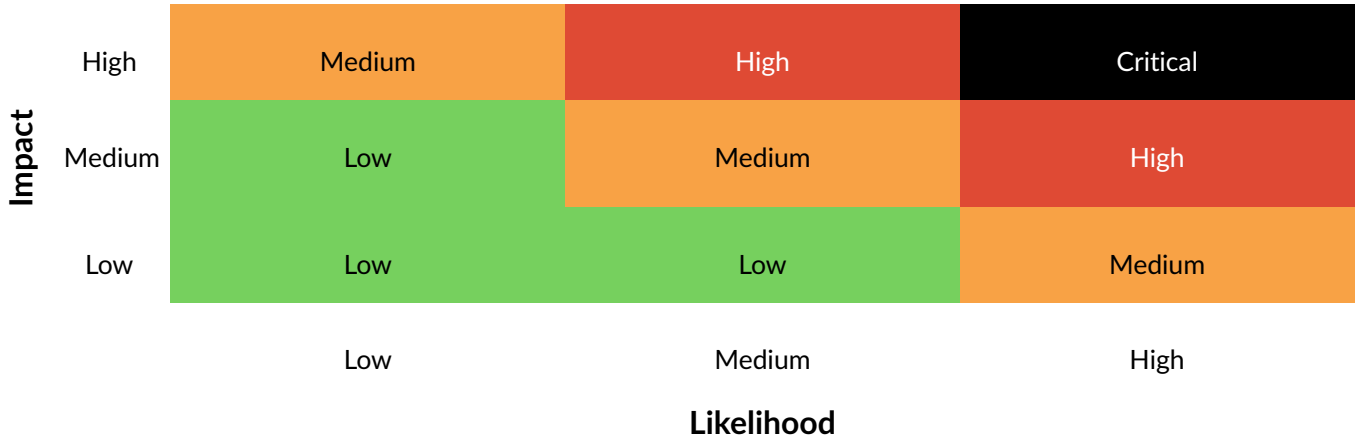

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.