The Web3 Security Labs

SMART CONTRACT AUDIT REPORT

Dolphinet PoS Governance Contracts – Comprehensive Audit Report

January 29, 2026 at 01:33:28 PM EST

@0xTheWeb3Labs | theweb3security.com

The Web3 Security Labs focus on Web3 project attack and defense, Web3 project audit and Web3 project security analysis.

Dolphinet PoS Governance Contracts – Comprehensive Audit Report

Report Type: Smart Contract Audit Report
Protocol: Dolphinet PoS Governance Contracts
Date: January 18, 2026
Auditors: Emmy Dev, Deeplake31337, Disha Singh, Nilima Bandurkar, VTZ-Aether

---

Executive Summary

This comprehensive audit consolidates findings from five independent security auditors who reviewed the Dolphinet Proof-of-Stake (PoS) Governance system. The audit identified **4 Critical**, **6 High**, **3 Medium**, and **2 Low severity** vulnerabilities across core contracts including `Governance.sol`, `DelegationManager.sol`, `RewardManager.sol`, `SlashingManager.sol`, and `ChainBase.sol`.

Critical Findings include:

- **Sybil/Flash-Voting Attack:** Attackers can manipulate elections with minimal capital or borrowed funds
- **Broken Slashing:** Jailed operators remain eligible for election and rewards
- **Missing Access Controls:** Critical functions can be exploited by any caller
- **Fund Loss in Rewards:** Reward claims can fail silently, permanently losing user funds

Status: Deployment is strongly discouraged until all Critical and High severity issues are remediated and re-audited.

Table of Contents

- Dolphinet PoS Governance Contracts – Comprehensive Audit Report
  - Executive Summary
  - Table of Contents
- CRITICAL SEVERITY FINDINGS

    - C-01: Sybil Attack in Governance – Vote Manipulation via Fund Transfers (Reusing Funds)
        - Description
        - Attack Vectors
        - Impact Assessment
        - Proof of Concept
        - Recommended Fixes
    - C-02: Broken Slashing Mechanism – Jailed Operators Remain Active
        - Description
        - Impact Assessment
        - Attack Scenario
        - Recommended Fixes
    - C-03: Missing Access Control on unRegisterFromGovernance()
        - Description
        - Impact Assessment
        - Attack Scenario
        - Recommended Fix
    - C-04: Unchecked Return Value in operatorClaimReward() – Permanent Fund Loss
        - Description
        - Threat Model – Checks-Effects-Interactions Violation
        - Attack Vectors
        - Impact Assessment
        - Recommended Fix
- HIGH SEVERITY FINDINGS
    - H-01: DoS Governance Election via Unbounded Candidate List
        - Description
        - Attack Scenario
        - Gas Analysis
        - Impact Assessment
        - Recommended Fixes
    - H-02: Missing Candidate Minimum Stake Enforcement
        - Description
        - Impact Assessment
        - Recommended Fix

    - H-03: Incorrect Loop Logic in Operator Unregistration Strands Stakers
        - Description
        - Attack Scenario
        - Impact Assessment
        - Recommended Fix
    - H-04: Contradictory Logic Disables unRegisterAsOperator
        - Description
        - Impact Assessment
        - Attack Scenario
        - Recommended Fix
    - H-05: Front-Running Slash via Early Undelegate
        - Description
        - Attack Scenario
        - Impact Assessment
        - Recommended Fix
    - H-06: Unbounded Loop DoS in Removal Operations
        - Description
        - Attack Scenario
        - Impact Assessment
        - Recommended Fix
- MEDIUM SEVERITY FINDINGS
    - M-01: Global Staker Cap of 320 Enables Denial-of-Service
        - Description
        - Attack Scenario
        - Impact Assessment
        - Recommended Fix
    - M-02: Missing Validation in updateStakePercent Causes DoS
        - Description
        - Attack Scenario
        - Impact Assessment
        - Recommended Fix
    - M-03: Unbounded Removal Loops – DoS Risk in Operator Management
        - Description

        - Gas Estimation
        - Impact Assessment
        - Recommended Fix
- LOW SEVERITY FINDINGS
    - L-01: ORIGINAL_CHAIN_ID Not Initialized in ChainDepositManager
        - Description
        - Impact
        - Recommended Fix
    - L-02: Missing Storage Gap in SlashingManagerStorage
        - Description
        - Impact
        - Recommended Fix
- SUMMARY TABLE
    - Finding Distribution by Severity
    - Remediation Priority
        - Immediate (Before Deployment)
        - High Priority (Within 1-2 Weeks)
        - Medium Priority (Before Public Launch)
    - Conclusion

CRITICAL SEVERITY FINDINGS

C-01: Sybil Attack in Governance – Vote Manipulation via Fund Transfers (Reusing Funds)

Severity: CRITICAL
Location: `Governance.sol` – Lines 179-196
Auditors: Emmy Dev, Nilima Bandurkar, VTZ-Aether, Deeplake31337

Description

The vote function determines voter eligibility solely based on the user's current native token balance (`msg.sender.balance`) at the moment of the transaction. The contract records that an address has voted using the `hasVoted` mapping to prevent the same address from voting twice. However, the contract fails to lock the funds used for voting or utilize a historical snapshot mechanism.

This allows a single user to execute the following attack loop indefinitely:

1. Vote with Address A (holding MIN_VOTER_BALANCE of 0.001 DOL)
2. Transfer the funds to Address B
3. Vote with Address B (since Address B has not voted yet and now holds the balance)
4. Repeat this process indefinitely

This is a classic **"Sybil Attack"** facilitated by a **"Double Spending"** style vulnerability in the governance weight calculation.

Attack Vectors

Direct Vector (Fund Transfer):

```
Attacker balance: 0.001 DOL
1. Call vote(CandidateA) from Address1 → 1 vote recorded
2. Transfer 0.001 DOL to Address2
3. Call vote(CandidateA) from Address2 → 2 votes recorded
4. Repeat for Address3, Address4, ... Address999
Result: 999 votes with 0.001 DOL capital
```

Flash Loan Vector:

- Attacker does not even need to own the capital
- Borrow funds via Flash Loan
- Generate thousands of votes within a single transaction (using a contract to deploy temporary sub-contracts)
- Repay the loan within the same transaction
- Effectively control governance **with zero cost**

Impact Assessment

Impact Category	Severity	Details

Election Integrity	CRITICAL	Entire election process becomes meaningless; consensus mechanism is completely bypassed
Infinite Voting Power	CRITICAL	Attacker with MIN_VOTER_BALANCE can generate unlimited votes by cycling funds through fresh addresses
Governance Takeover	CRITICAL	Malicious actors can easily elect malicious validators or block honest voters
Network Security	CRITICAL	Elected validators can censor transactions, reorganize blocks, or attack the network
Economic Attack Cost	CRITICAL	Attack cost is effectively **zero** (gas only, no capital lockup)

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract GovernanceExploitTest {
    DolphinetGovernance governance;
    address candidate = address(0x100);
    address attacker = address(0xBAD);
    address sybil1 = address(0xBAD1);
    address sybil2 = address(0xBAD2);

    uint256 MIN_VOTER_BALANCE = 1 ether;

    function test_SybilAttack_VoteManipulation() public {
        // Fund the attacker
        vm.deal(attacker, MIN_VOTER_BALANCE);
        vm.deal(sybil1, 0);
        vm.deal(sybil2, 0);

        // Attack Step 1: Attacker votes
        vm.prank(attacker);
        governance.vote(candidate);
        (,, uint256 votesAfterFirst,,) = governance.candidates(candidate);
        assert(votesAfterFirst == 1);
```

```
        // Attack Step 2: Transfer funds to Sybil 1
        vm.prank(attacker);
        payable(sybil1).transfer(MIN_VOTER_BALANCE);

        // Attack Step 3: Sybil 1 votes using SAME funds
        vm.prank(sybil1);
        governance.vote(candidate);
        (,, uint256 votesAfterSecond,,) = governance.candidates(candidate);
        assert(votesAfterSecond == 2); // Exploit successful!

        // Attack Step 4: Transfer to Sybil 2 and vote again
        vm.prank(sybil1);
        payable(sybil2).transfer(MIN_VOTER_BALANCE);
        vm.prank(sybil2);
        governance.vote(candidate);

        // Result: 1 unit of capital generated 3 votes
    }
}
```

Recommended Fixes

Short-term: Implement a snapshot mechanism

```
// Option 1: Check balance at a specific past block
function vote(address candidateOp) external {
    uint256 snapshotBlock = electionStartBlock;
    uint256 voterBalance = getBalance(msg.sender, snapshotBlock);
    require(voterBalance >= MIN_VOTER_BALANCE, "insufficient balance at snapshot
    // ... rest of logic
}

// Option 2: Use OpenZeppelin ERC20Votes with historical checkpoints
function vote(address candidateOp) external {
    uint256 votingPower = governanceToken.getPastVotes(msg.sender, electionStart
    require(votingPower >= MIN_VOTER_BALANCE, "insufficient voting power");
    // ... rest of logic
}
```

Long-term: Implement voting power locking

```
mapping(address => uint256) public voterLockExpiry;
mapping(address => uint256) public voterStakedAmount;

function vote(address candidateOp) external payable {
    require(msg.value >= MIN_VOTER_BALANCE, "insufficient stake");

    uint256 eid = currentElectionId;
    require(eid > 0, "election not started");
    require(!hasVoted[eid][msg.sender], "already voted");

    // Lock funds until next election or fixed duration
    uint256 lockDuration = 7 days;
    voterStakedAmount[msg.sender] += msg.value;
    voterLockExpiry[msg.sender] = block.timestamp + lockDuration;

    hasVoted[eid][msg.sender] = true;
    candidates[candidateOp].votes += 1;
}

function withdrawStake() external {
    require(block.timestamp >= voterLockExpiry[msg.sender], "stake is still lock
    uint256 amount = voterStakedAmount[msg.sender];
    require(amount > 0, "no stake");
    voterStakedAmount[msg.sender] = 0;
    (bool sent, ) = msg.sender.call{value: amount}("");
    require(sent, "transfer failed");
}
```

C-02: Broken Slashing Mechanism – Jailed Operators Remain Active

Severity: CRITICAL
Location: `SlashingManager.sol` Line 49, `Governance.sol` Line 204, `DelegationManager.sol` Line 183
Auditors: Nilima Bandurkar, Emmy Dev

Description

The `SlashingManager` allows the slasher to `jail` an operator:

```
function jail(address operator) external onlySlasher {
    isOperatorJail[operator] = true;
    emit IsJail(operator, true);
}
```

However, the `isOperatorJail` state is **never checked** in any critical logic flows:

1. **Governance.sol** – `finalizeElection()` : Jailed operators are not filtered out and can still be elected as top validators
2. **DelegationManager.sol** – `delegateTo()` : Users can still delegate to jailed operators
3. **RewardManager.sol**: Reward generation continues for jailed operators

Impact Assessment

Impact	Severity	Details
Jailing is Non-Functional	CRITICAL	The `jail` function has zero security effect; jailed operators continue operating normally
Malicious Validators	CRITICAL	Caught malicious validators remain in the active validator set and can continue attacking
Continued Staking	CRITICAL	Funds delegated to jailed operators continue earning rewards despite misconduct
Security Illusion	CRITICAL	Protocol appears to have slashing safeguards but they don't function

Attack Scenario

1. Validator X misbehaves (double-signs, censors transactions, etc.)
2. Slasher calls `jail(ValidatorX)` – operator is marked as jailed
3. ValidatorX continues:
    - Receiving new delegations
    - Generating and claiming rewards
    - Participating in governance
    - Being re-elected as validator in next election

4. Network security is completely compromised

Recommended Fixes

```
// In Governance.sol - finalizeElection()
function finalizeElection() external {
    require(!electionFinalized, "election already finalized");

    // Filter out jailed operators
    address[] memory filteredCandidates = new address[](candidateList.length);
    uint256 count = 0;

    for (uint256 i = 0; i < candidateList.length; i++) {
        address candidate = candidateList[i];
        require(
            !slashingManager.isOperatorJail(candidate),
            "jailed operator cannot be elected"
        );
        filteredCandidates[count] = candidate;
        count++;
    }

    // ... rest of election logic with filtered candidates
}

// In DelegationManager.sol - delegateTo()
function delegateTo(address op) external {
    require(
        !slashingManager.isOperatorJail(op),
        "cannot delegate to jailed operator"
    );
    // ... rest of delegation logic
}
```

C-03: Missing Access Control on `unRegisterFromGovernance()`

Severity: CRITICAL
Location: `DelegationManager.sol` – Line 126

Auditors: Nilima Bandurkar, Emmy Dev

Description

The `unRegisterFromGovernance()` function can be called by **anyone**, not just the governance contract. It can unregister any operator and forcibly undelegate all their stakers.

```
function unRegisterFromGovernance(address op) external {  // No access control!
    require(isOperator(op), "...");
    delete _operatorDetails[op];
    // ... undelegates all stakers
}
```

The function accepts an arbitrary `op` address parameter and performs destructive operations without verifying that `msg.sender` is the authorized governance contract.

Impact Assessment

Impact	Severity	Details
Griefing Attack	CRITICAL	Any malicious actor can unregister any legitimate operator at will
Fund Disruption	CRITICAL	All stakers delegated to the target operator are force-undelegated
Denial of Service	CRITICAL	Attackers can systematically remove all operators from the network
Network Paralysis	CRITICAL	PoS system becomes inoperable with no active operators

Attack Scenario

```
// Attacker calls this with any operator address
unRegisterFromGovernance(address(legitimateOperator));

// Result:
// 1. Operator's registration is deleted
// 2. All stakers are forcibly undelegated
```

```
// 3. Network loses a validator
// Repeat for all operators → network becomes non-functional
```

Recommended Fix

```
modifier onlyGovernance() {
    require(msg.sender == address(governance), "onlyGovernance");
    _;
}

function unRegisterFromGovernance(address op) external onlyGovernance {
    require(isOperator(op), "not an operator");
    delete _operatorDetails[op];

    // Undelegate all stakers
    address[] storage delegatedStakers = _operatorDelegatedStakers[op];
    while (delegatedStakers.length > 0) {
        _undelegate(delegatedStakers[delegatedStakers.length - 1]);
    }
}
```

C-04: Unchecked Return Value in `operatorClaimReward()` – Permanent Fund Loss

Severity: CRITICAL
Location: `RewardManager.sol` – Lines 100 and 120
Auditors: Emmy Dev, Nilima Bandurkar

Description

The `operatorClaimReward()` function does not revert on failed ETH transfer. It sets `operatorRewards[msg.sender] = 0` **before** the transfer and only returns the success status without requiring it.

```
function operatorClaimReward() external returns (bool) {
    uint256 claimAmount = operatorRewards[msg.sender];
    require(claimAmount > 0, "...");
```

```
    require(address(this).balance >= claimAmount, "...");

    operatorRewards[msg.sender] = 0;  // State updated BEFORE transfer

    emit OperatorClaimReward(msg.sender, claimAmount);

    (bool success, ) = payable(msg.sender).call{value: claimAmount}("");
    return success;  // No revert on failure - funds are lost!
}
```

The same vulnerability exists in `stakeHolderClaimReward()` at Line 120.

Threat Model – Checks-Effects-Interactions Violation

The code violates the **Checks-Effects-Interactions** pattern:

1. **Checks:** Validates balance requirement ✓
2. **Effects:** Zeroes the reward balance ✓
3. **Interactions:** Calls external address ✓ BUT doesn't validate success

When an external call fails, the reward is permanently lost.

Attack Vectors

Vector 1: Contract Receiver Revert

```
contract MaliciousReceiver {
    receive() external payable {
        revert("Cannot receive");
    }
}

// Operator delegates their reward address to MaliciousReceiver
// When they call operatorClaimReward(), the transfer fails silently
// Their reward balance is permanently zeroed
```

Vector 2: Out-of-Gas

- Transfer operation encounters gas issues
- Function returns false
- Rewards are already zeroed – permanently lost

Impact Assessment

Impact	Severity	Details
Permanent Fund Loss	CRITICAL	Operators/stakers lose their rewards entirely if transfer fails
Silent Failure	CRITICAL	Function returns false instead of reverting – caller may not detect failure
Reward Accounting Breakdown	CRITICAL	State is inconsistent: reward zeroed but funds never received
Economic Loss	CRITICAL	Direct financial impact on protocol users

Recommended Fix

```
function operatorClaimReward() external returns (bool) {
    uint256 claimAmount = operatorRewards[msg.sender];
    require(claimAmount > 0, "RewardManager: no rewards to claim");
    require(address(this).balance >= claimAmount, "RewardManager: insufficient b

    // Effects: Clear rewards AFTER validation, but BEFORE transfer
    operatorRewards[msg.sender] = 0;

    // Interactions: Perform transfer and REQUIRE success
    (bool success, ) = payable(msg.sender).call{value: claimAmount}("");
    require(success, "RewardManager: ETH transfer failed");

    emit OperatorClaimReward(msg.sender, claimAmount);
    return true;
}

function stakeHolderClaimReward(address chainBase) external returns (bool) {
    uint256 claimAmount = stakerRewards[chainBase][msg.sender];
    require(claimAmount > 0, "RewardManager: no rewards to claim");

    stakerRewards[chainBase][msg.sender] = 0;

    (bool success, ) = payable(msg.sender).call{value: claimAmount}("");
    require(success, "RewardManager: ETH transfer failed");
```

```
    emit StakeHolderClaimReward(msg.sender, claimAmount);
    return true;
}
```

HIGH SEVERITY FINDINGS

H-01: DoS Governance Election via Unbounded Candidate List

Severity: HIGH
Location: `Governance.sol` – Lines 70-82 (registerCandidate) and Lines 35-54 (reStartElection)
Auditors: Deeplake31337, Nilima Bandurkar, Emmy Dev

Description

Anyone can register as a candidate if their operator shares are greater than zero. The `registerCandidate` function pushes the caller's address to an unbounded `candidateList` array:

```
function registerCandidate() external {
    require(!candidates[msg.sender].exists, "already candidate");

    uint256 shares = delegationManager.getOperatorShares(msg.sender);
    require(
        shares > 0,
        "Governance.registerCandidate: operator has no shares delegated"
    );

    // ... candidate setup ...
    candidateList.push(msg.sender);  // Unbounded array growth!
}
```

The `reStartElection()` function iterates over the entire unbounded `candidateList`:

```
function reStartElection() external onlyOwner {
    currentElectionId += 1;
```

```
        lastElectionTime = block.timestamp;
        electionFinalized = false;
        finalizedElectionId = 0;
        _clearRankedSets();

        for (uint256 i = 0; i < candidateList.length; i++) {  // O(n) unbounded loop
            address op = candidateList[i];
            Candidate storage c = candidates[op];
            c.votes = 0;
            c.lastElectionId = currentElectionId;
            c.isValidator = false;
            c.isBlockVoter = false;
        }
    }
```

Attack Scenario

```
1. Attacker creates multiple addresses
2. Attacker obtains minimal shares for each (by depositing or receiving delegati
3. Attacker calls registerCandidate() from each address
   - Fills candidateList with 1,000+ entries
4. When reStartElection() is called:
   - Each iteration: 3,000–5,000 gas
   - Total: 1,000 × 4,000 = 4,000,000 gas
5. Function exceeds block gas limit
6. reStartElection() reverts – election cannot restart
7. GOVERNANCE IS FROZEN – no new elections possible
```

Gas Analysis

Assuming 5,000 malicious candidates:

- Loop iterations: 5,000
- Gas per iteration: ~4,000-5,000 (storage write + state operations)
- Total gas: 20,000,000 - 25,000,000 gas
- Block limit: ~30,000,000 gas on most EVM chains
- **Result: Function exceeds block limit or barely fits, making it unreliable**

Impact Assessment

Impact	Severity	Details
Governance Freeze	CRITICAL	Election restart becomes permanently impossible
Protocol Paralysis	CRITICAL	No new elections, validators cannot be rotated
Network Stagnation	CRITICAL	Current validator set is locked indefinitely
Low-Cost Attack	HIGH	Attacker only needs 0.001 DOL per address to participate

Recommended Fixes

Option 1: Enforce Maximum Candidates

```
uint256 public constant MAX_CANDIDATES = 100;

function registerCandidate() external {
    require(candidateList.length < MAX_CANDIDATES, "max candidates reached");
    require(!candidates[msg.sender].exists, "already candidate");

    uint256 shares = delegationManager.getOperatorShares(msg.sender);
    require(shares >= MIN_CANDIDATE_SHARES, "insufficient shares");

    Candidate storage c = candidates[msg.sender];
    c.exists = true;
    c.votes = 0;
    c.lastElectionId = currentElectionId;

    candidateList.push(msg.sender);
    emit CandidateRegistered(msg.sender, shares);
}
```

Option 2: Batch Processing

```
function reStartElectionBatch(uint256 startIdx, uint256 count) external onlyOwne
    uint256 endIdx = min(startIdx + count, candidateList.length);

    for (uint256 i = startIdx; i < endIdx; i++) {
        Candidate storage c = candidates[candidateList[i]];
        c.votes = 0;
```

```
            c.lastElectionId = currentElectionId + 1;
            c.isValidator = false;
            c.isBlockVoter = false;
        }

        if (endIdx == candidateList.length) {
            currentElectionId += 1;
            electionFinalized = false;
            emit ElectionStarted(currentElectionId, block.timestamp);
        }
    }
```

H-02: Missing Candidate Minimum Stake Enforcement

Severity: HIGH
Location: `Governance.sol` – Lines 70-82
Auditors: Emmy Dev, Nilima Bandurkar, Deeplake31337

Description

The contract documentation specifies: **"Candidates must stake native DOL: minimum = 320,000 DOL"**

However, the `registerCandidate` function only checks if delegated shares are greater than zero:

```
uint256 shares = delegationManager.getOperatorShares(msg.sender);
require(
    shares > 0,  // Only checks > 0, NOT >= 320,000 equivalent!
    "Governance.registerCandidate: operator has no shares delegated"
);
```

Impact Assessment

Spam Attack Vector:

- Attacker creates 10,000 addresses
- Deposits 1 wei to each address

- Calls `registerCandidate()` from each
- Inflates `candidateList` to 10,000+ entries
- Combined with H-01, causes permanent governance freeze

Economic Security Bypass:

- Validators operate with negligible “skin in the game”
- PoS security relies on economic incentives – removes them entirely
- Malicious validator has minimal loss if slashed (only had 1 wei)

Combined Attack (C-01 + H-02):

- Use C-01 Sybil attack to vote for 10,000 zero-stake candidates
- Elect malicious validators at near-zero cost
- Network is completely compromised

Recommended Fix

```
uint256 public constant MIN_CANDIDATE_SHARES = 320_000 ether; // 320,000 DOL

function registerCandidate() external {
    require(!candidates[msg.sender].exists, "already candidate");

    uint256 shares = delegationManager.getOperatorShares(msg.sender);
    require(
        shares >= MIN_CANDIDATE_SHARES,
        "Governance.registerCandidate: insufficient stake (requires 320,000+ DOL
    );

    Candidate storage c = candidates[msg.sender];
    c.exists = true;
    c.votes = 0;
    c.lastElectionId = currentElectionId;
    c.isValidator = false;
    c.isBlockVoter = false;

    candidateList.push(msg.sender);

    emit CandidateRegistered(msg.sender, shares);
}
```

H-03: Incorrect Loop Logic in Operator Unregistration Strands Stakers

Severity: HIGH
Location: `DelegationManager.sol` – Lines 112-117 (unRegisterAsOperator) and Lines 137-142 (unRegisterFromGovernance)
Auditors: Nilima Bandurkar

Description

When an operator unregisters (voluntarily or forced by governance), the contract attempts to undelegate all their stakers using a forward-iteration loop with swap-and-pop removal:

```
for (uint256 i = 0; i < delegatedStakers.length; i++) {
    address staker = delegatedStakers[i];
    _undelegate(staker);
    // ...
}

// Inside _undelegate():
for (uint256 i = 0; i < delegatedStakers.length; i++) {
    if (delegatedStakers[i] == staker) {
        delegatedStakers[i] = delegatedStakers[delegatedStakers.length - 1];
        delegatedStakers.pop();
        break;
    }
}
```

The Problem: Index Skipping

When `_undelegate()` removes an element via swap-and-pop:

- Element at index 0 is removed
- Element at end (index N-1) moves to index 0
- Main loop increments to index 1
- **Element that moved to index 0 is SKIPPED**

Attack Scenario

```
Initial: [staker1, staker2, staker3, staker4, staker5]

i=0: _undelegate(staker1)
  → Removes index 0, moves staker5 to index 0
  → Array: [staker5, staker2, staker3, staker4]

i=1: _undelegate(staker2)
  → staker5 at index 0 was SKIPPED
  → Removes index 1 (staker2)
  → Array: [staker5, staker4, staker3]

i=2: _undelegate(staker3)
  → Removes index 2
  → Array: [staker5, staker4]

i=3: Loop exits (i >= length)
  → staker4 was NEVER UNDELEGATED!

Result: ~50% of stakers are not properly undelegated
```

Impact Assessment

Impact	Severity	Details
Stranded Delegations	HIGH	~50% of operator's stakers remain mapped to operator
Broken State	HIGH	Stakers delegated to non-existent operator cannot undelegate or withdraw
Fund Lock	HIGH	Affected stakers' funds may be permanently locked or require manual recovery
Operator Shutdown Failure	HIGH	Operator cannot effectively retire due to stranded delegations

Recommended Fix

Option 1: Backwards Iteration (Simplest)

```
function unRegisterAsOperator() external {
    require(isOperator(msg.sender), "not an operator");

    address[] storage delegatedStakers = _operatorDelegatedStakers[msg.sender];

    // Iterate backwards to avoid skipping elements
    while (delegatedStakers.length > 0) {
        address staker = delegatedStakers[delegatedStakers.length - 1];
        _undelegate(staker);
    }

    delete _operatorDetails[msg.sender];
    emit OperatorUnregistered(msg.sender);
}
```

Option 2: Index-Based Mapping (O(1) removal)

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

function unRegisterAsOperator() external {
    require(isOperator(msg.sender), "not an operator");

    address[] storage delegatedStakers = _operatorDelegatedStakers[msg.sender];

    // Remove all stakers using their stored indices
    for (uint256 i = delegatedStakers.length; i > 0; i--) {
        address staker = delegatedStakers[i - 1];
        uint256 idx = stakerIndex[msg.sender][staker];

        // Swap with last element
        address lastStaker = delegatedStakers[delegatedStakers.length - 1];
        delegatedStakers[idx] = lastStaker;
        stakerIndex[msg.sender][lastStaker] = idx;

        delegatedStakers.pop();
        delete stakerIndex[msg.sender][staker];
    }

    delete _operatorDetails[msg.sender];
}
```

H-04: Contradictory Logic Disables `unRegisterAsOperator`

Severity: HIGH
Location: `DelegationManager.sol` – Lines 102-105
Auditors: Nilima Bandurkar

Description

The `unRegisterAsOperator` function enforces a contradictory requirement:

```
function unRegisterAsOperator() external {
    require(
        operatorShares[msg.sender] == 0,  // CONTRADICTION!
        "operator cannot have shares delegated"
    );

    // This code tries to undelegate stakers:
    address[] storage delegatedStakers = _operatorDelegatedStakers[msg.sender];
    for (uint256 i = 0; i < delegatedStakers.length; i++) {
        _undelegate(delegatedStakers[i]);
    }
}
```

The Contradiction:

- Function purpose: Undelegate all stakers
- Precondition check: `operatorShares` must be 0
- Logic: If `operatorShares == 0`, the array is essentially empty
- Result: Function can **only** succeed when there's nothing to undelegate

Impact Assessment

Impact	Severity	Details
Function is Unusable	HIGH	Any active operator with stakers cannot call this function
Forced Manual Undelegate	HIGH	Operator must manually call `undelegate()` for each staker individually

Operational Friction	HIGH	Thousands of stakers = thousands of transactions needed to retire
Gas Cost Explosion	HIGH	Operator must spend millions of gas to retire gracefully

Attack Scenario

```
Operator has 1,000 delegated stakers

Scenario 1: Operator tries to unRegisterAsOperator()
→ operatorShares[operator] = 1000 (from all delegations)
→ Precondition fails: "operator cannot have shares delegated"
→ Transaction reverts - cannot retire!

Scenario 2: Operator manually calls undelegate() 1,000 times
→ Cost: 1,000 transactions × 100k gas = 100 million gas
→ Cost at 50 gwei: ~500 DOL in gas fees alone
→ Alternative: Operator stays active indefinitely
```

Recommended Fix

```
function unRegisterAsOperator() external {
    require(isOperator(msg.sender), "not an operator");

    // Remove the contradictory require statement
    // The function is DESIGNED to undelegate all stakers

    address[] storage delegatedStakers = _operatorDelegatedStakers[msg.sender];

    // Iterate backwards to properly handle removals
    while (delegatedStakers.length > 0) {
        address staker = delegatedStakers[delegatedStakers.length - 1];
        _undelegate(staker);
    }

    delete _operatorDetails[msg.sender];

    emit OperatorUnregistered(msg.sender);
}
```

H-05: Front-Running Slash via Early Undelegate

Severity: HIGH
Location: `SlashingManager.sol`, `DelegationManager.sol` (undelegate)
Auditors: Deeplake31337

Description

When slashing an operator, the protocol iterates through all delegated stakers and distributes the slash penalty:

```
function freezeAndSlashingShares(
    address operator,
    uint256 slashShare
) external onlySlasher returns (uint256) {
    (address[] memory stakers, uint256[] memory shares) =
        delegationManager.getStakerSharesOfOperator(operator);

    for (uint256 i = 0; i < stakers.length; i++) {
        if (shares[i] > 0) {
            uint256 stakerSlashedShare = (slashShare * shares[i] * SCALE) /
                totalShares / SCALE;
            slashingStakerShares[stakers[i]] += stakerSlashedShare;
            delegationManager.slashingStakingShares(operator, stakers[i], staker
        }
    }
}
```

The Problem: There is **no cooldown period** on `undelegate()`. A staker can:

1. Monitor the mempool for a slash transaction
2. Front-run the slash with an `undelegate()` call
3. Remove their shares from the operator before slashing is applied
4. Avoid the penalty entirely

Attack Scenario

```
1. Staker has 1,000 DOL delegated to MaliciousOperator
2. MaliciousOperator is caught misbehaving
```

```
3. Slasher initiates: freezeAndSlashingShares(MaliciousOperator, 500 DOL slash)
   - Slash transaction in mempool
4. Staker monitors mempool, sees slash transaction
5. Staker immediately calls: undelegate() → removes 1,000 DOL from operator
6. Staker's undelegate tx is mined BEFORE slash tx
7. Slash transaction executes, but staker has no shares → NO PENALTY
8. Staker successfully evades 500 DOL loss

Result: Economic slashing mechanism is defeated
```

Impact Assessment

Impact	Severity	Details
Slashing Bypass	HIGH	Well-positioned stakers can completely evade penalties
Economic Security Failure	HIGH	PoS incentive system is undermined
Validator Collusion	HIGH	Validators and stakers can coordinate to avoid punishment

Recommended Fix

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

uint256 public constant UNDELEGATE_COOLDOWN = 7 days;

function undelegate(address operator) external {
    require(
        block.timestamp >= undelegateLockedUntil[msg.sender][operator],
        "undelegate cooldown not expired"
    );

    uint256 amount = delegatedTo[msg.sender][operator];
    require(amount > 0, "no delegation");

    // Lock the undelegation to prevent front-running
    undelegateLockedUntil[msg.sender][operator] = block.timestamp + UNDELEGATE_C
```

```
    // Actual removal happens after cooldown
    // OR immediate removal but funds locked

    _undelegate(msg.sender);
    emit Undelegated(msg.sender, operator, amount);
}
```

H-06: Unbounded Loop DoS in Removal Operations

Severity: HIGH
Location: `DelegationManager.sol` – Multiple functions
Auditors: Emmy Dev, Disha Singh

Description

Multiple functions iterate over potentially unbounded arrays (`stakerList` and `operatorDelegatedStakers`):

```
// In _undelegate()
for (uint256 i = 0; i < stakerList.length; i++) {
    if (stakerList[i] == staker) {
        stakerList[i] = stakerList[stakerList.length - 1];
        stakerList.pop();
        break;
    }
}

// In unRegisterAsOperator() and unRegisterFromGovernance()
for (uint256 i = 0; i < delegatedStakers.length; i++) {
    if (delegatedStakers[i] == staker) {
        // ... removal logic
    }
}
```

As the network grows, these arrays can contain **thousands of entries**. Each iteration consumes ~3,000-5,000 gas.

Attack Scenario

```
Scenario: Network with 10,000 stakers delegated to single operator

1. Operator calls unRegisterAsOperator()
2. Function loops through 10,000 stakers
3. Each iteration: ~4,000 gas
4. Total: 10,000 × 4,000 = 40,000,000 gas
5. Block limit: ~30,000,000 gas
6. Result: TRANSACTION REVERTS – operator cannot unregister

Alternative: Staker tries to undelegate
– Similar issue: loop through 10,000 stakers
– May exceed gas limit
```

Impact Assessment

Impact	Severity	Details
Out-of-Gas DoS	HIGH	Critical functions become permanently unusable as network grows
Operator Stranding	HIGH	Operators cannot unregister when delegation count is high
Withdrawal Blocking	HIGH	Stakers may be unable to undelegate their funds
Network Scaling Failure	HIGH	System breaks as network grows beyond ~2,000 stakers

Recommended Fix

Use Index Mapping for O(1) Removal:

```
mapping(address => uint256) public stakerIndexInList;
mapping(address => mapping(address => uint256)) public delegatedStakerIndex;

function _undelegate(address staker) internal {
    uint256 idx = delegatedStakerIndex[msg.sender][staker];

    address[] storage delegatedStakers = _operatorDelegatedStakers[msg.sender];
    address lastStaker = delegatedStakers[delegatedStakers.length - 1];
```

```
    delegatedStakers[idx] = lastStaker;
    delegatedStakerIndex[msg.sender][lastStaker] = idx;
    delegatedStakers.pop();

    delete delegatedStakerIndex[msg.sender][staker];
}

function removeFromStakerList(address staker) internal {
    uint256 idx = stakerIndexInList[staker];

    address lastStaker = stakerList[stakerList.length - 1];
    stakerList[idx] = lastStaker;
    stakerIndexInList[lastStaker] = idx;
    stakerList.pop();

    delete stakerIndexInList[staker];
}
```

MEDIUM SEVERITY FINDINGS

M-01: Global Staker Cap of 320 Enables Denial-of-Service

Severity: MEDIUM
Location: `ChainBase.sol` – Line 18, Line 42
Auditors: Disha Singh, Nilima Bandurkar

Description

The protocol enforces a hard-coded maximum of 320 unique stakers:

```
uint256 internal constant MAX_STAKER_NUMBERS = 320;

// In deposit():
require(stakerNumbers < MAX_STAKER_NUMBERS, "Stakers too much in this pool");
```

Attack Scenario

```
1. Attacker creates 320 EOAs
2. Funds each with minimum deposit amount
3. Calls deposit() from each address
4. Sets stakerNumbers = 320
5. Network becomes CLOSED
6. Honest users attempting to join are blocked
```

Impact Assessment

- **Network Centralization:** Attacker monopolizes validator set early
- **Economic DoS:** Honest participants cannot stake or participate
- **Scaling Limitation:** Hard cap prevents network growth
- **Future Incompatibility:** Static limit becomes obsolete as ecosystem grows

Recommended Fix

```
uint256 public MAX_STAKER_NUMBERS = 10_000; // Increase from 320

// Add admin function for dynamic adjustment
function updateMaxStakers(uint256 newMax) external onlyAdmin {
    require(newMax > currentStakerCount, "new max must be > current count");
    MAX_STAKER_NUMBERS = newMax;
    emit MaxStakersUpdated(newMax);
}

// Alternative: Dynamic cap based on total stakes
uint256 public MIN_STAKE_PER_STAKER = 100 ether;

function getMaxStakers() public view returns (uint256) {
    return totalStaked / MIN_STAKE_PER_STAKER;
}
```

M-02: Missing Validation in `updateStakePercent` Causes DoS

Severity: MEDIUM
Location: `RewardManager.sol` – Lines 174-182
Auditors: Disha Singh, Nilima Bandurkar

Description

The `updateStakePercent` function allows setting the percentage of fees allocated to stakers. It validates `_stakePercent > 0` but fails to check `_stakePercent <= 100`:

```
function updateStakePercent(uint256 _stakePercent) external onlyRewardManager {
    require(_stakePercent > 0, "percent must be > 0");
    // Missing: require(_stakePercent <= 100, "percent must be <= 100");

    stakePercent = _stakePercent;
}

// Later in payFee():
uint256 stakeFee = (operatorTotalFee * stakePercent) / 100;
uint256 operatorFee = operatorTotalFee - stakeFee;  // UNDERFLOW if stakePercent
```

Attack Scenario

```
1. Malicious admin calls: updateStakePercent(150)
2. Subsequent payFee() calls calculate:
   - stakeFee = (1000 * 150) / 100 = 1500
   - operatorFee = 1000 - 1500 = UNDERFLOW
3. Transaction reverts, rewards cannot be distributed
4. ALL reward payouts blocked until corrected
```

Impact Assessment

- **Complete Reward DoS:** Entire reward system frozen
- **Economic Loss:** Users cannot receive earned fees
- **Governance Risk:** Admin role is critical vulnerability
- **Single Key Failure:** One compromised key = full protocol DoS

Recommended Fix

```
function updateStakePercent(uint256 _stakePercent) external onlyRewardManager {
    require(
        _stakePercent > 0 && _stakePercent <= 100,
        "RewardManager: stakePercent must be between 1 and 100"
    );

    uint256 oldPercent = stakePercent;
    stakePercent = _stakePercent;

    emit StakePercentUpdated(oldPercent, _stakePercent);
}
```

M-03: Unbounded Removal Loops – DoS Risk in Operator Management

Severity: MEDIUM
Location: `DelegationManager.sol` – Multiple functions
Auditors: Disha Singh

Description

Both `stakerList` and `operatorDelegatedStakers` arrays are unbounded and iterated linearly:

```
// Removal from stakerList
for (uint256 i = 0; i < stakerList.length; i++) {
    if (stakerList[i] == staker) {
        stakerList[i] = stakerList[stakerList.length - 1];
        stakerList.pop();
        break;
    }
}

// Removal from operatorDelegatedStakers
for (uint256 i = 0; i < delegatedStakers.length; i++) {
    if (delegatedStakers[i] == staker) {
        delegatedStakers[i] = delegatedStakers[delegatedStakers.length - 1];
        delegatedStakers.pop();
        break;
```

```
    }
}
```

Gas Estimation

```
Assuming 5,000 stakers:
- Iterations: 5,000
- Gas per iteration: 3,000 gas
- Total: 15,000,000 gas (near block limit at 30M)

Assuming 10,000 stakers:
- Iterations: 10,000
- Gas per iteration: 3,000 gas
- Total: 30,000,000 gas (hits block limit)
```

Impact Assessment

- **Gas Scaling Issues:** Operations become unreliable with 5,000+ delegations
- **Function Unusability:** Critical operations fail when delegation count exceeds threshold
- **Network Size Limitation:** Network cannot scale beyond ~5,000 stakers

Recommended Fix

Implement index-based O(1) removal (see H-06 detailed fix above).

LOW SEVERITY FINDINGS

L-01: ORIGINAL_CHAIN_ID Not Initialized in ChainDepositManager

Severity: LOW
Location: `ChainDepositManager.sol` – Line 131
Auditors: Emmy Dev

Description

The `ORIGINAL_CHAIN_ID` variable is never initialized:

```
function domainSeparator() public view returns (bytes32) {
    if (block.chainid == ORIGINAL_CHAIN_ID) {  // ORIGINAL_CHAIN_ID is 0 (never
        return _DOMAIN_SEPARATOR;
    } else {
        return _calculateDomainSeparator();
    }
}
```

Impact

- **Unnecessary Gas:** Domain separator check always fails, forcing recalculation
- **Inefficiency:** Every call to `domainSeparator()` triggers expensive calculation
- **No Security Risk:** Logic still works correctly, just inefficiently

Recommended Fix

```
function initialize() external initializer {
    // ... existing initialization code ...
    ORIGINAL_CHAIN_ID = block.chainid;
}
```

L-02: Missing Storage Gap in SlashingManagerStorage

Severity: LOW
Location: `SlashingManagerStorage.sol`
Auditors: Emmy Dev

Description

Unlike other storage contracts, `SlashingManagerStorage` doesn't include a `__gap` array for future upgradability:

```
contract SlashingManagerStorage {
    // ... storage variables ...

    // Missing: uint256[50] private __gap;
}
```

Impact

- **Future Upgrade Risk:** New variables in storage may overwrite existing data
- **UUPS Pattern:** Required for safe upgradeable contracts
- **No Immediate Risk:** Only affects future upgrades

Recommended Fix

```
contract SlashingManagerStorage {
    // ... existing storage ...

    // Reserve storage for future upgrades
    uint256[50] private __gap;
}
```

SUMMARY TABLE

Finding Distribution by Severity

Severity	Count	Issues
CRITICAL	4	C-01, C-02, C-03, C-04
HIGH	6	H-01, H-02, H-03, H-04, H-05, H-06
MEDIUM	3	M-01, M-02, M-03

LOW	2	L-01, L-02
TOTAL	**15**	Comprehensive Security Review

Remediation Priority

Immediate (Before Deployment)

1. **C-01:** Implement snapshot mechanism for voting
2. **C-02:** Add jail status checks in election and delegation functions
3. **C-03:** Add `onlyGovernance` modifier to `unRegisterFromGovernance()`
4. **C-04:** Add `require(success)` checks in reward claim functions
5. **H-01:** Enforce maximum candidate limit or implement batch processing
6. **H-02:** Enforce minimum stake requirement for candidates
7. **H-03:** Fix loop iteration (backward or index-based removal)
8. **H-04:** Remove contradictory require statement

High Priority (Within 1-2 Weeks)

9. **H-05:** Implement undelegate cooldown
10. **H-06:** Implement index-based O(1) removal
11. **M-02:** Add upper bound validation to `updateStakePercent`

Medium Priority (Before Public Launch)

12. **M-01:** Increase or dynamically adjust staker cap
13. **M-03:** Optimize removal loops
14. **L-01:** Initialize `ORIGINAL_CHAIN_ID`
15. **L-02:** Add `__gap` to storage contracts

Conclusion

The Dolphinet PoS Governance system contains multiple critical vulnerabilities that render it unsuitable for deployment in its current state. The combination of:

- **Sybil voting attacks** (C-01) + **missing stake minimums** (H-02) = Trivial governance takeover
- **Non-functional slashing** (C-02) + **undelegate front-running** (H-05) = Broken economic security
- **Missing access controls** (C-03) = Grief attacks on operators
- **Fund loss in rewards** (C-04) = User funds permanently lost on transfer failures

Immediate remediation and comprehensive re-audit are mandatory before mainnet deployment.

Consolidated Audit Report prepared by The Web3 Security Labs (@0xTheWeb3Labs | theweb3security.com)