POWERS PROTOCOL SECURITY AUDIT REPORT

Disclaimer

This security audit report is provided for informational purposes only. The findings and recommendations contained herein are based on the audit of the Powers Protocol smart contracts as of the audit date. This report does not constitute financial, legal, or investment advice.

The audit was conducted on the provided source code and may not reflect the final deployed version. Users should conduct their own due diligence before interacting with the protocol.

Report prepared by:
Okiki Omisande
Blockchain Security Researcher
Twitter: @okiki_omisande
Telegram: @Invcbull

Table of Contents

Executive Summary

Project: Powers Protocol - Role Restricted Governance Protocol

Audit Date: August 2025

Auditor: INVCBULL AUDIT GROUP

Protocol: Powers Protocol

Contracts Audited: Powers.sol, Law.sol, LawUtilities.sol

Total Findings: 10 vulnerabilities (2 Critical, 2 High, 4 Medium, 2 Low)

Overview:

The Powers Protocol is a role-based governance system for on-chain organizations that allows for the separation and distribution of decision-making power by codifying relationships between stakeholders. The protocol combines a governance engine (Powers) with role-restricted and modular contracts (Laws) to govern actions.

Critical Risk Summary:

The Powers Protocol governance system contains critical vulnerabilities that could lead to permanent governance lockup, privilege escalation, and system manipulation. **IMMEDIATE REMEDIATION REQUIRED** before any mainnet deployment.

Introduction

The Powers Protocol is an innovative governance system designed for on-chain organizations that implements a role-based approach to decision-making. Unlike traditional governance models, Powers separates and distributes decision-making power through codified relationships between stakeholders, creating a modular and flexible governance framework.

The protocol consists of three main components:

- **Powers**: The core governance engine that manages roles and governance flows
- **Laws**: Role-restricted and modular contracts that define what actions can be taken by which roles under specific conditions
- **Actions**: Consist of calldata and unique nonces sent to target laws

This audit was conducted to identify security vulnerabilities, design flaws, and implementation issues that could compromise the protocol's security, functionality, or user funds.

Audit Overview

Audit Methodology

The security audit was conducted using a combination of:

- **Manual Code Review**: Line-by-line analysis of smart contract code
- **Static Analysis**: Automated vulnerability detection tools
- **Cross-Contract Analysis**: Examination of interactions between contracts
- **Attack Vector Analysis**: Identification of potential exploit scenarios
- **Gas Optimization Review**: Analysis of gas consumption patterns

Audit Timeline

- **Start Date**: August 2025

- **Duration**: Comprehensive security assessment
- **Focus Areas**: Access control, input validation, state management, cross-contract interactions

---

Scope

Audit Scope

- **Powers.sol**: Core governance engine contract
- **Law.sol**: Base law implementation contract
- **LawUtilities.sol**: Utility library for law contract

Risk Classification

Severity Levels

Severity	Description	Impact
Critical	Immediate threat to protocol security or user funds	Complete system compromise, permanent fund loss
High	Significant security risk requiring urgent attention	Major functionality disruption, potential fund loss
Medium	Moderate security concern with limited impact	Minor functionality issues, gas inefficiencies
Low	Minor issues with minimal security impact	Code quality, documentation, or maintenance concerns

Risk Assessment Criteria

- **Exploitability**: How easily can the vulnerability be exploited
- **Impact**: Potential damage to users, protocol, or funds
- **Scope**: Number of users or contracts affected
- **Permanence**: Whether the impact is reversible or permanent

---

FINDINGS SUMMARY

Severity	Count
Critical	2
High	2
Medium	4
Low	2

CRITICAL SEVERITY FINDINGS

C-1: Public Role Overflow in Quorum Calculations

Location: `Powers.sol:470, 484` (quorum calculation functions)
Severity: CRITICAL
Impact: Permanent disabling of public governance laws

Vulnerability Description:

The Powers Protocol uses `PUBLIC_ROLE = type(uint256).max` to represent number of member with public access, but this causes arithmetic overflow in quorum calculations. When calculating quorum requirements, the system multiplies `amountMembers * conditions.quorum`, where `amountMembers` for `PUBLIC_ROLE` is set to `type(uint256).max`. This multiplication overflows and causes all proposals on public governance laws to revert, effectively making these laws permanently unusable.

The vulnerability exists in both `_quorumReached()` and `_voteSucceeded()` functions, which are critical for determining proposal outcomes. When a law is configured with `allowedRole = PUBLIC_ROLE`, any attempt to create proposals or check voting status will fail due to arithmetic overflow, completely disabling public governance functionality.

Root Cause:

```
uint256 public constant PUBLIC_ROLE = type(uint256).max; // == a lot
roles[PUBLIC_ROLE].amountMembers = type(uint256).max; // the number for
holders of the PUBLIC_ROLE is type(uint256).max

// In _quorumReached() and _voteSucceeded():
uint256 amountMembers = _countMembersRole(conditions.allowedRole); // Returns
type(uint256).max for PUBLIC_ROLE
return amountMembers * conditions.quorum <= (proposedAction.forVotes +
proposedAction.abstainVotes) * DENOMINATOR;
// ❌ Overflow: type(uint256).max * quorum causes arithmetic overflow
```

Attack Path:

```
// Step 1: Admin creates a law with public access
// Law configured with allowedRole = PUBLIC_ROLE

// Step 2: User attempts to create proposal
propose(lawId, calldata, nonce, uri);

// Step 3: _propose() calls _quorumReached() for validation
_quorumReached(actionId);

// Step 4: Quorum calculation overflows
uint256 amountMembers = _countMembersRole(PUBLIC_ROLE); // Returns
type(uint256).max
uint256 calculation = amountMembers * conditions.quorum; // OVERFLOW!

// Step 5: Transaction reverts due to arithmetic overflow
// Result: Law becomes permanently unusable
```

Impact Analysis:

- **Public Governance Disabled:** Any law configured with `allowedRole = PUBLIC_ROLE` becomes permanently unusable
- **DAO Functionality Loss:** Public participation in governance is completely blocked
- **Protocol Inoperability:** Laws intended for public access cannot process proposals or votes
- **No Workaround:** The overflow occurs at the protocol level, making it impossible to fix without code changes

Fix:

```
// Option 1: Use special handling for PUBLIC_ROLE
function _countMembersRole(uint256 roleId) internal view virtual returns
(uint256 amountMembers) {
    if (roleId == PUBLIC_ROLE) {
        return 1; // Treat as single member for calculations
    }
    return roles[roleId].amountMembers;
}

// Option 2: Use different constant for PUBLIC_ROLE
uint256 public constant PUBLIC_ROLE =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE;
```

C-2: Protocol DoS via Gas Exhaustion in `fulfill()` Function

Location: `Powers.sol:182-188` (fulfill function execution loop)
Severity: CRITICAL
Impact: Permanent governance lockup through gas exhaustion

Vulnerability Description:

The `fulfill()` function contains a critical vulnerability where Law contracts have unlimited control over execution parameters without gas limit validation. When a Law contract calls `fulfill()` with massive arrays of targets, values, and calldatas, the execution loop can run out of gas and revert. Since the action remains in "requested" state but can never be completed, this creates a permanent denial of service vector.

The vulnerability is particularly dangerous because Law contracts have full control over the execution parameters they pass to `fulfill()`. A malicious or compromised Law contract can intentionally create massive execution arrays to trigger gas exhaustion, effectively creating a permanent DoS attack on the governance system with no recovery mechanism.

Root Cause:

```
function fulfill(
    uint16 lawId,
    uint256 actionId,
    address[] calldata targets,
    uint256[] calldata values,
    bytes[] calldata calldatas
) external payable virtual onlyActiveLaw(lawId) {
    // ... validation checks ...
```

```
    // VULNERABILITY: No gas limit validation on execution arrays
    _actions[actionId].fulfilled = true;

    // Execution loop that can run out of gas
    for (uint256 i = 0; i < targets.length; ++i) {
        (bool success, bytes memory returndata) = targets[i].call{ value:
values[i] }(calldatas[i]);
        Address.verifyCallResult(success, returndata);
    }
}
```

Attack Path:

```
// Step 1: Malicious Law contract implements handleRequest() to return massive
arrays
function handleRequest(...) public view returns (...) {
    // Create 10,000+ target addresses
    address[] memory targets = new address[](10000);
    uint256[] memory values = new uint256[](10000);
    bytes[] memory calldatas = new bytes[](10000);

    // Fill with valid but gas-intensive operations
    for (uint256 i = 0; i < 10000; i++) {
        targets[i] = someContract;
        values[i] = 0;
        calldatas[i] = abi.encodeWithSelector(someFunction.selector,
complexData);
    }

    return (actionId, targets, values, calldatas, stateChange);
}

// Step 2: Law calls _replyPowers() which calls Powers.fulfill()
_replyPowers(lawId, actionId, targets, values, calldatas);

// Step 3: fulfill() sets fulfilled = true immediately
_actions[actionId].fulfilled = true;

// Step 4: Execution loop runs out of gas and reverts
// Gas limit exceeded after ~5000 iterations

// Step 5: State reverts but action remains "requested" but never "fulfilled"
// Result: DAO governance permanently blocked
```

Impact Analysis:

- **Permanent Governance Lockup:** No mechanism to recover from failed fulfillments
- **Treasury Actions Blocked:** Cannot execute approved proposals
- **Protocol Upgrades Blocked:** Governance cannot function
- **No Recovery Mechanism:** Once blocked, system cannot self-recover

Fix:

```
function fulfill(
    uint16 lawId,
    uint256 actionId,
    address[] calldata targets,
```

```
    uint256[] calldata values,
    bytes[] calldata calldatas
) external payable virtual onlyActiveLaw(lawId) {
    // ... validation checks ...

    // Add gas limit validation
    require(targets.length <= MAX_EXECUTION_TARGETS, "Too many execution
targets");

    // Execute first, then set fulfilled
    for (uint256 i = 0; i < targets.length; ++i) {
        (bool success, bytes memory returndata) = targets[i].call{ value:
values[i] }(calldatas[i]);
        Address.verifyCallResult(success, returndata);
    }

    // Only set fulfilled after successful execution
    _actions[actionId].fulfilled = true;

    emit ActionExecuted(lawId, actionId, targets, values, calldatas);
}
```

HIGH SEVERITY FINDINGS

H-1: Unbounded Calldata Storage DoS

Location: `Powers.sol:146` (lawCalldata storage)
Severity: HIGH
Impact: Storage bloat leading to expensive operations and potential DoS

Vulnerability Description:

The Powers Protocol stores `lawCalldata` in the `_actions` mapping without any size validation or limits. This allows attackers to submit massive calldata payloads that bloat storage and make all state operations expensive. The vulnerability affects both `request()` and `propose()` functions, which directly store user-provided calldata without validation.

The impact is particularly severe because the calldata is stored permanently in contract storage and is accessed during various operations including state checks, action retrieval, and execution. Large calldata entries can cause gas limit issues during state reads and writes, potentially making the protocol unusable for all users.

Root Cause:

```
function request(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // ... validation checks ...

    // VULNERABILITY: No size validation before storage
    _actions[actionId].lawCalldata = lawCalldata; // ❌ Can be massive

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

function propose(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
```

```
string memory uriAction) external {
    // ... validation checks ...

    // VULNERABILITY: Same issue in propose
    proposedAction.lawCalldata = lawCalldata; // ❌ No size limits
}
```

Attack Path:

```
// Step 1: Attacker crafts massive calldata
bytes memory hugeCalldata = new bytes(1000000); // 1MB of data
// Fill with arbitrary data to maximize storage bloat

// Step 2: Attacker submits multiple large requests
for (uint256 i = 0; i < 100; i++) {
    powers.request(lawId, hugeCalldata, i, "dos_attack");
}

// Step 3: Storage becomes bloated
// Each action now contains 1MB of calldata
// Total storage bloat: 100MB

// Step 4: State operations become expensive
// Reading action data requires loading massive calldata
// Gas costs increase dramatically for all users

// Step 5: Protocol becomes unusable
// Users cannot afford gas costs for basic operations
```

Impact Analysis:

- **Storage Bloat:** Large calldata entries consume excessive contract storage space
- **Gas Cost Inflation:** Reading action data becomes expensive for all users
- **Protocol Degradation:** State operations become slower and more costly
- **Permanent Impact:** Stored calldata cannot be easily removed once written

Fix:

```
uint256 public constant MAX_CALLDATA_SIZE = 10000; // 10KB limit

function request(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // Add size validation
    require(lawCalldata.length <= MAX_CALLDATA_SIZE, "Calldata too large");

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

H-2: Nonce Manipulation and Replay Attacks

Location: `Powers.sol:148` (nonce handling)
Severity: HIGH
Impact: Duplicate action execution and bypass of intended nonce ordering

Vulnerability Description:

The Powers Protocol allows users to specify arbitrary nonce values without validation or ordering constraints. This enables replay attacks where the same malicious action can be executed multiple times with different nonces, and allows users to bypass intended nonce ordering by submitting actions with lower nonces after higher ones have been processed.

The vulnerability is particularly dangerous because it undermines the security model of action uniqueness and ordering. Attackers can replay successful attacks multiple times, and users can manipulate the order of actions to their advantage and bypassing security measures that rely on nonce ordering.

Root Cause:

```
function request(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // ... validation checks ...

    // VULNERABILITY: No nonce validation or ordering
    _actions[actionId].nonce = nonce; // ❌ User-controlled without
constraints

    // actionId = hash(lawId, lawCalldata, nonce)
    // Different nonces create different actionIds for same action
}

function propose(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // Same vulnerability exists in propose function
    proposedAction.nonce = nonce; // ❌ No validation
}
```

Attack Scenarios:

Scenario 1: Replay Attack

```
// Step 1: Attacker creates malicious action
bytes memory maliciousCalldata = abi.encodeWithSelector(
    transfer.selector,
    attacker,
    1000000 * 10**18
);

// Step 2: Submit same action with different nonces
powers.request(lawId, maliciousCalldata, 1, "attack1");
powers.request(lawId, maliciousCalldata, 2, "attack2"); // Replay!
powers.request(lawId, maliciousCalldata, 3, "attack3"); // Another replay!

// Step 3: All three actions are valid and can be executed
// Result: Attacker gets 3x the intended amount
```

Scenario 2: Nonce Ordering Bypass

```
// Step 1: User submits action with high nonce
powers.request(lawId, calldata, 100, "high_nonce");

// Step 2: Later, user submits action with lower nonce
```

```
powers.request(lawId, differentCalldata, 50, "low_nonce");

// Step 3: Both actions are valid
// Lower nonce action can be executed after higher nonce
// Bypasses intended ordering constraints
```

Impact Analysis:

- **Replay Attacks:** Same malicious action can be executed multiple times with different nonces
- **Ordering Bypass:** Users can manipulate action execution order by using lower nonces after higher ones
- **Security Model Violation:** this bypasses the intended action uniqueness and ordering guarantees
- **Economic Exploitation:** Can lead to multiple unauthorized executions of the same action

Fix:

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

function request(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // Validate nonce ordering
    require(nonce > userNonces[msg.sender], "Nonce too low");
    userNonces[msg.sender] = nonce;

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

MEDIUM SEVERITY FINDINGS

M-1: Law Count Inflation

Location: `Powers.sol:379` (law adoption logic)
Severity: MEDIUM
Impact: Breaks invariants and wastes gas through duplicate law adoption

Vulnerability Description:

The `_adoptLaw()` function does not check whether a law address has already been adopted, allowing the same law contract to be adopted multiple times. This inflates the `lawCount` variable without adding new functionality, breaking the invariant that `lawCount` represents the number of unique laws, and wastes gas through unnecessary increments.

The vulnerability is particularly problematic because it can be exploited by administrators (either malicious or compromised) to artificially inflate the law count, potentially affecting governance logic that relies on this count. Additionally, it creates confusion about the actual number of active laws in the system.

Root Cause:

```
function _adoptLaw(LawInitData memory lawInitData) internal virtual {
    // check if added address is indeed a law
```

```
    if (!ERC165Checker.supportsInterface(lawInitData.targetLaw,
type(ILaw).interfaceId)) {
        revert Powers__IncorrectInterface();
    }

    // VULNERABILITY: No check that the law is not already adopted
    // Same law can be adopted multiple times

    laws[lawCount].active = true;
    laws[lawCount].targetLaw = lawInitData.targetLaw;
    lawCount++; // ❌ Always increments, even for duplicate laws
}
```

Attack Path:

```
// Step 1: Admin adopts Law A for the first time
LawInitData memory lawData = LawInitData({
    targetLaw: address(lawA),
    nameDescription: "Test Law",
    conditions: conditions,
    config: config
});
powers.adoptLaw(lawData); // lawCount = 1

// Step 2: Admin adopts the same Law A again
powers.adoptLaw(lawData); // lawCount = 2 (same law!)

// Step 3: Repeat multiple times
powers.adoptLaw(lawData); // lawCount = 3
powers.adoptLaw(lawData); // lawCount = 4
powers.adoptLaw(lawData); // lawCount = 5

// Result: lawCount = 5 but only 1 unique law exists
// Breaks invariant: "active laws = lawCount - 1"
```

Impact Analysis:

- **Invariant Violation:** `lawCount` no longer represents the actual number of unique laws
- **Gas Waste:** Unnecessary increments and storage operations for duplicate law adoptions
- **Governance Confusion:** Incorrect law count may affect governance logic and decision-making
- **Potential Logic Errors:** Any code relying on law count assumptions may malfunction

Fix:

```
mapping(address => bool) public adoptedLaws;

function _adoptLaw(LawInitData memory lawInitData) internal virtual {
    // Check if law is already adopted
    require(!adoptedLaws[lawInitData.targetLaw], "Law already adopted");
    adoptedLaws[lawInitData.targetLaw] = true;

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

M-2: Invariant Violation After Law Revocation

Location: `Powers.sol:359` (law revocation logic)
Severity: MEDIUM
Impact: Breaks system invariants and affects governance logic

Vulnerability Description:

The Powers Protocol maintains a `lawCount` variable that only increases and never decreases, even when laws are revoked. This breaks the invariant that "active laws = lawCount - 1" after any law revocation, potentially affecting governance logic that relies on this relationship.

The vulnerability is particularly problematic because it creates a permanent discrepancy between the actual number of active laws and the recorded count. This can lead to incorrect assumptions in governance logic, potential gas inefficiencies, and confusion about the system state.

Root Cause:

```
uint16 public lawCount = 1; // Only increases, never decreases

function revokeLaw(uint16 lawId) public onlyPowers {
    if (laws[lawId].active == false) revert Powers__LawNotActive();

    laws[lawId].active = false; // ❌ Only sets active = false
    // lawCount is never decremented

    emit LawRevoked(lawId);
}
```

Attack Path:

```
// Step 1: Admin adopts 5 laws
for (uint256 i = 0; i < 5; i++) {
    powers.adoptLaw(lawData[i]); // lawCount = 6
}

// Step 2: Admin revokes 3 laws
powers.revokeLaw(1); // lawCount still = 6
powers.revokeLaw(2); // lawCount still = 6
powers.revokeLaw(3); // lawCount still = 6

// Step 3: Invariant is broken
// Active laws: 3 (laws 4, 5, 6)
// lawCount: 6
// Invariant "active laws = lawCount - 1" fails: 3 ≠ 5

// Step 4: Governance logic may malfunction
// Any code relying on this invariant will fail
```

Impact Analysis:

- **Invariant Violation:** System state becomes inconsistent after law revocations
- **Governance Logic Errors:** Code relying on law count assumptions may fail
- **Permanent Discrepancy:** Cannot be corrected without manual intervention or code changes

- **Potential Gas Inefficiencies:** Incorrect assumptions about system state may lead to inefficient operations

Fix:

```
uint16 public activeLawCount = 0; // Separate counter for active laws

function _adoptLaw(LawInitData memory lawInitData) internal virtual {
    // ... existing code ...
    activeLawCount++;
    lawCount++;
}

function revokeLaw(uint16 lawId) public onlyPowers {
    if (laws[lawId].active == false) revert Powers__LawNotActive();

    laws[lawId].active = false;
    activeLawCount--; // Decrement active count

    emit LawRevoked(lawId);
}
```

M-3: Missing Action Existence Check in `cancel()` Function

Location: `Powers.sol:274` (cancel function validation)
Severity: MEDIUM
Impact: Allows cancelling non-existent actions, causing confusion

Vulnerability Description:

The `cancel()` function only validates that the caller matches the action's caller but does not verify that the action actually exists. This allows users to "cancel" non-existent actions, which can cause confusion and potentially affect system state in unexpected ways.

The vulnerability is particularly problematic because it creates a false sense of security - users may believe they have successfully cancelled an action when in reality no such action existed. This can lead to incorrect assumptions about system state and potential confusion in governance processes.

Root Cause:

```
function cancel(uint16 lawId, bytes calldata lawCalldata, uint256 nonce)
public virtual {
    uint256 actionId = _hashAction(lawId, lawCalldata, nonce);

    // VULNERABILITY: Only checks caller, not if action exists
    if (msg.sender != _actions[actionId].caller) revert
Powers__AccessDenied();
    // ❌ If action doesn't exist, _actions[actionId].caller == address(0)
    // ❌ Check msg.sender != address(0) will pass for any non-zero address

    return _cancel(lawId, lawCalldata, nonce);
}
```

Attack Path:

```
// Step 1: Attacker calls cancel() for non-existent action
uint256 fakeActionId = _hashAction(1, "fake_calldata", 999);
// _actions[fakeActionId] doesn't exist, so caller = address(0)

// Step 2: Validation passes incorrectly
// msg.sender != address(0) ✅ (any non-zero address passes)
// Action appears to be "cancelled"

// Step 3: System state becomes confusing
// Event is emitted: ProposedActionCancelled(fakeActionId)
// But no actual action was cancelled

// Step 4: Potential confusion in governance
// Users may think an action was cancelled when it never existed
```

Impact Analysis:

- **False Cancellation:** Users can cancel non-existent actions, creating confusion
- **State Confusion:** Creates misleading system state and event logs
- **Event Pollution:** Emits events for non-existent actions, cluttering logs
- **Governance Confusion:** May affect decision-making processes and user understanding

Fix:

```
function cancel(uint16 lawId, bytes calldata lawCalldata, uint256 nonce)
public virtual {
    uint256 actionId = _hashAction(lawId, lawCalldata, nonce);

    // Check if action exists
    require(_actions[actionId].voteStart != 0, "Action does not exist");

    // Check caller
    if (msg.sender != _actions[actionId].caller) revert
Powers__AccessDenied();

    return _cancel(lawId, lawCalldata, nonce);
}
```

M-4: Stuck ETH in `receive()` Function

Location: `Powers.sol:107` (receive function)
Severity: MEDIUM
Impact: ETH becomes permanently stuck in contract with no withdrawal mechanism

Vulnerability Description:

The Powers Protocol includes a `receive()` function that accepts ETH deposits but provides no mechanism for withdrawing these funds. Any ETH sent to the contract becomes permanently stuck, creating a potential loss of funds for users who accidentally send ETH to the contract.

The vulnerability is particularly problematic because it's a common mistake for users to send ETH to contract addresses, especially when interacting with governance systems. The lack of a withdrawal mechanism means these funds are permanently lost, which can lead to user frustration and potential legal issues.

Root Cause:

```
/// @notice receive function enabling ETH deposits.
/// @dev This is a virtual function, and can be overridden in the DAO
implementation.
/// @dev No access control on this function: anyone can send funds in native
currency into the contract.
receive() external payable virtual {
    emit FundsReceived(msg.value, msg.sender);
    // ❌ No withdrawal mechanism provided
    // ❌ ETH becomes permanently stuck
}
```

Attack Path:

```
// Step 1: User accidentally sends ETH to Powers contract
// Common mistake when interacting with governance

// Step 2: ETH is accepted by receive() function
powers.receive{value: 1 ether}(); // 1 ETH sent

// Step 3: User realizes mistake and tries to withdraw
// No withdrawal function exists
// ETH is permanently stuck

// Step 4: Funds are lost forever
// No recovery mechanism available
```

Impact Analysis:

- **Permanent Fund Loss:** ETH sent to the contract cannot be recovered

Fix:

```
function withdrawStuckETH() external onlyRole(ADMIN_ROLE) {
    uint256 balance = address(this).balance;
    require(balance > 0, "No ETH to withdraw");

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

    emit ETHWithdrawn(msg.sender, balance);
}
```

LOW SEVERITY FINDINGS

L-1: Calldata Injection Vulnerabilities

Location: `Powers.sol:151` (law execution call)
Severity: LOW
Impact: Potential unintended code execution through malicious calldata

Vulnerability Description:

The Powers Protocol passes raw calldata directly to Law contracts without validation, enabling a calldata injection attacks. While Law contracts are responsible for validating calldata, there's no guarantee that all Law implementations will properly validate the received data, which can lead to unintended code execution.

The vulnerability is particularly concerning because it relies on Law contract implementations to provide proper validation, creating a security dependency that may not always be met. Malicious or poorly implemented Law contracts could process malicious calldata in unexpected ways.

Root Cause:

```
function request(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // ... validation checks ...

    // VULNERABILITY: Raw calldata passed without validation
    (bool success) = ILaw(law.targetLaw).executeLaw(msg.sender, lawId,
lawCalldata, nonce);
    // ❌ No validation of calldata content
    // ❌ Relies on Law contract to validate properly
}
```

Attack Path:

```
// Step 1: Attacker crafts malicious calldata
bytes memory maliciousCalldata = abi.encodeWithSelector(
    bytes4(0x12345678), // Fake function selector
    uint256(0xdeadbeef),
    address(0x1234567890123456789012345678901234567890),

bytes32(0xabcdef1234567890abcdef1234567890abcdef1234567890abcdef1234567890)
);

// Step 2: Attacker submits malicious calldata
powers.request(lawId, maliciousCalldata, nonce, "injection_attack");

// Step 3: Law contract receives malicious calldata
// If Law contract doesn't validate properly:
// - Fake selectors could trigger unintended functions
// - Malicious parameters could cause unexpected behavior
// - Selector smuggling attacks become possible

// Step 4: Unintended code execution
// Law contract processes malicious calldata incorrectly
// Results in unintended contract manipulation
```

Impact Analysis:

- **Unintended Code Execution:** Malicious calldata can trigger unintended functions in Law contracts
- **Contract Manipulation:** Parameters can be crafted to exploit poorly implemented Law contracts
- **Selector Smuggling:** Fake selectors can bypass intended function calls in Law implementations
- **Security Dependency:** Relies entirely on Law contract implementations to validate calldata properly

Fix:

```
// Option 1: Add basic calldata validation
function request(uint16 lawId, bytes calldata lawCalldata, uint256 nonce,
string memory uriAction) external {
    // Basic validation
    require(lawCalldata.length >= 4, "Invalid calldata");
    require(lawCalldata.length <= MAX_CALLDATA_SIZE, "Calldata too large");

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

// Option 2: Document security requirements for Law contracts
// Ensure all Law implementations validate calldata properly
```

L-2: Documentation Error in LawUtilities

Location: `LawUtilities.sol:275` (hashLaw function documentation)
Severity: LOW
Impact: Misleading documentation can lead to incorrect implementations

Vulnerability Description:

The `hashLaw()` function in LawUtilities has incorrect documentation that states it "hashes the combination of law address and index" when it actually hashes "powers address and index". This misleading documentation can cause developers to misunderstand the function's behavior and implement incorrect logic.

This oversight is particularly problematic because it affects the understanding of how law hashing works, which is neccessary for the security and functionality of the Powers Protocol. Incorrect implementations based on this documentation could lead to security vulnerabilities or functional bugs.

Root Cause:

```
/// @notice Creates a unique identifier for a law, used for sandboxing
executions of laws.
/// @dev Hashes the combination of law address and index //!doc - should be
corrected to hashes the combination of power address and index
/// @param powers Address of the Powers contract
/// @param index Index of the law
/// @return lawHash Unique identifier for the law
function hashLaw(address powers, uint16 index) public pure returns (bytes32
lawHash) {
```

```
    lawHash = keccak256(abi.encode(powers, index)); // ✅ Actually hashes
powers + index
    // ❌ Documentation says "law address and index" but actually uses "powers
address and index"
}
```

Impact Analysis:

- **Developer Confusion:** Misleading documentation causes incorrect understanding of law hashing
- **Implementation Errors:** Developers may implement wrong logic based on incorrect documentation
- **Maintenance Issues:** Wrong documentation makes code harder to maintain and understand

Fix:

```
/// @notice Creates a unique identifier for a law, used for sandboxing
executions of laws.
/// @dev Hashes the combination of powers address and index
/// @param powers Address of the Powers contract
/// @param index Index of the law
/// @return lawHash Unique identifier for the law
function hashLaw(address powers, uint16 index) public pure returns (bytes32
lawHash) {
    lawHash = keccak256(abi.encode(powers, index));
}
```

IMPACT ANALYSIS

Critical Findings:

- **C-1:** Public governance laws become permanently unusable
- **C-2:** Permanent governance lockup through gas exhaustion
- **Total Risk:** Complete governance system failure

High Findings:

- **H-1:** Storage bloat will make protocol unusable due to high gas costs
- **H-2:** Replay attacks will lead to multiple unauthorized executions

---

REMEDIATION SUGGESTED

Phase 1: Critical Fixes (IMMEDIATE)

1. **Fix C-1:** Implement special handling for PUBLIC_ROLE in quorum calculations
2. **Fix C-2:** Set a maximum limit to the array in fulfill()
3. **Fix H-1:** Add calldata size limits to prevent storage bloat
4. **Fix H-2:** Implement nonce validation and ordering

Phase 2: Medium Priority Fixes

5. **Fix M-1:** Add duplicate law adoption prevention
6. **Fix M-2:** Implement separate active law counter
7. **Fix M-3:** Add action existence check in cancel()
8. **Fix M-4:** Add ETH withdrawal mechanism

Phase 3: Low Priority Fixes

9. **Fix L-1:** Add basic calldata validation
10. **Fix L-2:** Correct documentation

TESTING RECOMMENDATIONS

Critical Path Testing:

- Test PUBLIC_ROLE overflow scenarios with various quorum values
- Test gas exhaustion attacks with massive execution arrays
- Test storage bloat scenarios with large calldata
- Test nonce manipulation and replay attack vectors
- Test law adoption and revocation edge cases

Integration Testing:

- Full governance flow testing (propose → vote → execute)
- Cross-contract interaction testing
- Gas optimization verification
- Edge case handling for all functions

Security Testing:

- Fuzz testing for all input parameters
- Stress testing with maximum array sizes
- Access control bypass attempts

CONCLUSION

The Powers Protocol contains **critical vulnerabilities** that must be addressed before any production deployment. The public role overflow and gas exhaustion vulnerabilities alone could lead to complete governance system failure, while the storage bloat and replay attack issues would make the protocol unusable or exploitable.

Key Recommendations:

1. **DO NOT DEPLOY** until all Critical and High severity issues are resolved
2. **IMPLEMENT** comprehensive security testing framework
3. **CONDUCT** additional economic modeling for attack scenarios
4. **CONSIDER** formal verification for critical governance functions

Protocol Assessment:

The Powers Protocol demonstrates innovative governance architecture with its role-based system and modular Law contracts. However, the identified vulnerabilities pose significant risks to the protocol's security and functionality. The critical issues, particularly the PUBLIC_ROLE overflow and gas exhaustion vulnerabilities, require immediate attention before any mainnet deployment.

This report contains confidential security information. Distribution should be limited to authorized personnel only.