# Grove Gov & Xchain Helpers

## Security Review

Cantina Managed review by:

**Optimum**, Lead Security Researcher
**Devtooligan**, Security Researcher

September 8, 2025

# Contents

**1 Introduction** **2**

1.1 About Cantina . . . 2

1.2 Disclaimer . . . 2

1.3 Risk assessment . . . 2

1.3.1 Severity Classification . . . 2

**2 Security Review Summary** **3**

**3 Findings** **4**

3.1 Informational . . . 4

3.1.1 Test Improvements . . . 4

3.1.2 Deployment scripts lack pre-deployment safety validations . . . 6

3.1.3 Missing post-deployment verification checks . . . 7

3.1.4 Missing `msg.value` sanity check . . . 9

3.1.5 Excess gas fees are burned instead of refunded . . . 10

3.1.6 Redundant call to `calculateRetryableSubmissionFee()` . . . 10

3.1.7 `sendMessageL1toL2()`: Redundant local variables . . . 11

# 1 Introduction

## 1.1 About Cantina

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

## 1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

## 1.3 Risk assessment

| Severity | Description |
|---|---|
| **Critical** | *Must* fix as soon as possible (if already deployed). |
| **High** | Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users. |
| **Medium** | Global losses <10% or losses to only a subset of users, but still unacceptable. |
| **Low** | Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies. |
| **Gas Optimization** | Suggestions around gas saving practices. |
| **Informational** | Suggestions around best practices or readability. |

### 1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

## 2 Security Review Summary

Steakhouse builds transparent, efficient, and accessible financial primitives to power the next generation of capital markets on public blockchains.

From Aug 20th to Aug 22nd the Cantina team conducted a review of grove-gov-relay and xchain-helpers on commit hashes 0dbffef8 and 670964a3. The team identified a total of **7** issues:

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 0 | 0 | 0 |
| Medium Risk | 0 | 0 | 0 |
| Low Risk | 0 | 0 | 0 |
| Gas Optimizations | 0 | 0 | 0 |
| Informational | 7 | 0 | 7 |
| **Total** | **7** | **0** | **7** |

# 3 Findings

## 3.1 Informational

### 3.1.1 Test Improvements

**Severity:** Informational

**Context:** ArbitrumERC20Integration.t.sol#L1-L99

**Summary:** The test suite for `ArbitrumERC20Forwarder` focuses primarily on happy path scenarios and lacks coverage for error conditions and boundary value testing. While the existing tests adequately cover the main cross-chain functionality, missing edge case coverage may allow boundary condition bugs to reach production.

**Finding Description:** The current test suite in `ArbitrumERC20Integration.t.sol` provides solid coverage for the primary cross-chain messaging functionality but concentrates on successful execution paths ("happy path"). The tests validate critical security boundaries including authority validation and cross-domain address verification, which is appropriate for the core security model. However, several edge cases and error conditions remain untested. Missing test coverage includes:

- Arithmetic edge cases.
- Property-based fuzzing of parameters and message sizes.
- Boundary value testing with zero or extreme parameters (gasLimit=0, maxFeePerGas=0, very large messages).
- Multiple sequential calls.
- Insufficient token balance scenarios.

**Recommendation:** Consider expanding the test suite to include comprehensive negative test cases, boundary value testing, and property-based fuzzing. The following test categories would provide more thorough coverage:

1. Boundary Value Fuzzing

```
function testFuzz_boundaryValues(uint256 gasLimit, uint256 maxFeePerGas, uint256 baseFee) public {
    // Test extreme boundary values
    gasLimit = bound(gasLimit, 0, 50_000_000); // Include zero and very high values
    maxFeePerGas = bound(maxFeePerGas, 0, 10000 gwei); // Include zero and extreme fees
    baseFee = bound(baseFee, 0, 1000 gwei); // Include zero and high base fees

    vm.assume(gasLimit * maxFeePerGas < type(uint256).max / 2); // Prevent overflow

    initBaseContracts(getChain("plume").createFork());

    // Should handle boundary values gracefully
    if (gasLimit == 0 || maxFeePerGas == 0) {
        // Might revert or succeed depending on implementation
        try ArbitrumERC20Forwarder.sendMessageL1toL2(
            bridge.sourceCrossChainMessenger,
            destinationReceiver
            abi.encodeCall(MessageOrdering.push, (1)),
            gasLimit,
            maxFeePerGas,
            baseFee
        ) {
            // Success is acceptable
        } catch {
            // Revert is also acceptable for edge cases
        }
    } else {
        ArbitrumERC20Forwarder.sendMessageL1toL2(
            bridge.sourceCrossChainMessenger,
            destinationReceiver,
            abi.encodeCall(MessageOrdering.push, (1)),
            gasLimit,
            maxFeePerGas,
            baseFee
        );
    }
}
```

```
function testFuzz_messageSizes(bytes calldata message) public {
    vm.assume(message.length <= 100000); // Test various sizes including empty

    initBaseContracts(getChain("plume").createFork());

    ArbitrumERC20Forwarder.sendMessageL1toL2(
        bridge.sourceCrossChainMessenger,
        destinationReceiver,
        message,
        200000, // Higher gas for larger messages
        1 gwei,
        block.basefee
    );
}
```

2. Multiple Sequential Calls Testing:

```
function test_multipleSequentialCalls() public {
    initBaseContracts(getChain("plume").createFork());

    // Test multiple calls in sequence to verify state consistency
    for (uint256 i = 1; i <= 3; i++) {
        ArbitrumERC20Forwarder.sendMessageL1toL2(
            bridge.sourceCrossChainMessenger,
            destinationReceiver,
            abi.encodeCall(MessageOrdering.push, (i)),
            100000,
            1 gwei,
            block.basefee
        );
    }

    relaySourceToDestination();

    // Verify all messages were processed in order
    assertEq(moDestination.length(), 3);
    assertEq(moDestination.messages(0), 1);
    assertEq(moDestination.messages(1), 2);
    assertEq(moDestination.messages(2), 3);
}
```

3. Property-Based Gas Parameter Fuzzing:

```
function testFuzz_gasParametersWithinBounds(uint256 gasLimit, uint256 maxFeePerGas, uint256 baseFee)
↪   public {
    // Bound inputs to reasonable ranges
    gasLimit = bound(gasLimit, 21000, 10_000_000); // Min gas to reasonable max
    maxFeePerGas = bound(maxFeePerGas, 1 gwei, 1000 gwei); // Reasonable fee range
    baseFee = bound(baseFee, 1 gwei, 100 gwei); // Reasonable base fee range

    vm.assume(gasLimit * maxFeePerGas < type(uint256).max / 2); // Prevent overflow

    initBaseContracts(getChain("plume").createFork());

    // Should succeed with valid bounded parameters
    ArbitrumERC20Forwarder.sendMessageL1toL2(
        bridge.sourceCrossChainMessenger,
        destinationReceiver,
        abi.encodeCall(MessageOrdering.push, (1)),
        gasLimit,
        maxFeePerGas,
        baseFee
    );
}
```

4. Insufficient Token Balance Testing:

```
function test_insufficientTokenBalance() public {
    // Set up environment with insufficient PLUME tokens
    source.selectFork();
    deal(ArbitrumERC20Forwarder.PLUME_GAS_TOKEN, sourceAuthority, 0.001 ether);

    initBaseContracts(getChain("plume").createFork());
```

```
        vm.expectRevert(); // Should revert due to insufficient balance
        ArbitrumERC20Forwarder.sendMessageL1toL2(
            bridge.sourceCrossChainMessenger,
            destinationReceiver,
            abi.encodeCall(MessageOrdering.push, (1)),
            100000,  // High gas requiring more tokens than available
            1 gwei,
            block.basefee + 10 gwei
        );
    }
```

**Steakhouse:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.1.2 Deployment scripts lack pre-deployment safety validations

**Severity:** Informational

**Context:** Deploy.s.sol#L1-L137

**Summary:** The deployment scripts in `Deploy.s.sol` and `Deploy.sol` do not perform comprehensive pre-deployment safety validations, potentially leading to costly deployment failures or incorrect configurations. The scripts could deploy with invalid parameters, fail mid-deployment due to insufficient resources, or deploy to wrong blockchains.

**Finding Description:** The deployment scripts perform minimal validation before executing deployment operations. The current process only validates basic environment setup but lacks comprehensive safety checks that could prevent common deployment failures. Current deployment flow:

```
function run() public {
    vm.createSelectFork(vm.envString("PLUME_RPC_URL"));
    vm.startBroadcast();

    address executor = Deploy.deployExecutor(0, 7 days);
    address receiver = Deploy.deployArbitrumReceiver(Ethereum.GROVE_PROXY, executor);

    Deploy.setUpExecutorPermissions(executor, receiver, msg.sender);

    vm.stopBroadcast();
}
```

Missing pre-deployment validations:

- Environment Validations:
  - Chain ID verification (prevent wrong blockchain deployment).
  - Gas price spikes.
- Resource Validations:
  - Deployer balance sufficiency for deployment costs.
- Parameter Validations:
  - Constructor parameter bounds validation.
  - Address validation for required dependencies.
  - Configuration constant verification.
- Dependency Validations:
  - Required external contracts exist and have code.

Missing validations can cause costly deployment failures: wrong blockchain deployment, mid-deployment failures leaving inconsistent state, or invalid configurations causing governance operation failures after apparent successful deployment. Deployment errors are common in multi-chain environments. Manual deployment processes with minimal validation increase the probability of human error, particularly given varying network conditions and environmental differences between deployment targets.

**Recommendation:** Consider implementing pre-deployment validation to prevent common deployment failures. Here's how to integrate validation into the deployment flow:

- Updated Deployment Flow with Validation:

```
function run() public {
    vm.createSelectFork(vm.envString("PLUME_RPC_URL"));

    // Pre-broadcast validations (test environment)
    _validateResources();
    _validateDependencies();
    _validateParameters(0, 7 days);

    vm.startBroadcast();

    // Post-broadcast validations (real network)
    _validateEnvironment();

    address executor = Deploy.deployExecutor(0, 7 days);
    address receiver = Deploy.deployArbitrumReceiver(Ethereum.GROVE_PROXY, executor);
    Deploy.setUpExecutorPermissions(executor, receiver, msg.sender);

    vm.stopBroadcast();

    console.log("  Deployment completed successfully");
    console.log("Executor:", executor);
    console.log("Receiver:", receiver);
}

// Validation functions...
```

} ```

Representative Validation Examples:

- Environment Validation (after startBroadcast):

```
function _validateEnvironment() private view {
    require(block.chainid == EXPECTED_CHAIN_ID, "Wrong chain - expected Plume mainnet");
    require((tx.gasprice < HIGH_GAS_PRICE_THRESHOLD, "High gas price detected");
    }
}
```

- Resource Validation:

```
function _validateResources() private view {
    require(msg.sender.balance >= MIN_DEPLOYER_BALANCE, "Insufficient balance for deployment");
    console.log("Deployer balance:", msg.sender.balance);
}
```

- Dependency Validation:

```
function _validateDependencies() private view {
    require(EXPECTED_GROVE_PROXY.code.length > 0, "Grove Proxy not deployed on target chain");
    require(EXPECTED_L1_CROSS_DOMAIN.code.length > 0, "L1 cross-domain contract not found");
}
```

- Parameter Validation:

```
function _validateParameters(uint256 delay, uint256 gracePeriod) private pure {
    require(gracePeriod >= MIN_GRACE_PERIOD, "Grace period below minimum");
    require(delay <= MAX_DELAY, "Delay period too long");
}
```

**Steakhouse:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.1.3 Missing post-deployment verification checks

**Severity:** Informational

**Context:** Deploy.s.sol#L1-L139

**Summary:** The deployment scripts do not verify that deployed contracts were configured correctly after deployment completes. While contracts may deploy successfully, misconfigurations in constructor parameters or role assignments could go unnoticed until governance operations fail. This finding is rated as Low severity.

**Finding Description:** The current deployment process completes without verifying that the deployed contracts match expected configurations. The deployment script logs addresses but does not validate the internal state of deployed contracts. Current post-deployment flow:

```
function run() public {
    // ... deployment logic ...

    console.log("executor deployed at:", executor);
    console.log("receiver deployed at:", receiver);

    vm.stopBroadcast();
    // No verification of deployment success or configuration
}
```

- Missing post-deployment verifications:
- Constructor Parameter Verification:
  - Executor delay and grace period settings match expected values.
  - ArbitrumReceiver authority and target addresses are correct.
- Role and Permission Verification:
  - Receiver has SUBMISSION_ROLE on Executor.
  - Deployer no longer has DEFAULT_ADMIN_ROLE on Executor.
  - Executor retains self-admin role for governance operations.

Missing post-deployment verification can result in silent configuration failures where contracts deploy successfully but are misconfigured for operation. These issues may only surface during actual governance operations, potentially causing critical governance actions to fail when time-sensitive decisions are needed.

Configuration errors during deployment are common, particularly in complex multi-contract systems with interdependencies. Without explicit verification, subtle misconfigurations can easily go undetected until operational testing or production usage reveals the issues.

**Recommendation:** Consider implementing comprehensive post-deployment verification to validate that deployed contracts match expected configurations. The verification should occur after deployment but before completing the script to catch configuration issues immediately. Updated Deployment Flow with Post-Deployment Verification:

```
function run() public {
    // ... existing deployment logic ...

    vm.startBroadcast();

    address executor = Deploy.deployExecutor(0, 7 days);
    address receiver = Deploy.deployArbitrumReceiver(Ethereum.GROVE_PROXY, executor);
    Deploy.setUpExecutorPermissions(executor, receiver, msg.sender);

    vm.stopBroadcast();

    // Verify deployment configuration
    _verifyDeployment(executor, receiver, 0, 7 days);

    console.log("  Deployment and verification completed successfully");
}
```

Representative Verification Examples:

- Constructor Parameter Verification:

  ```
  function _verifyConstructorParams(
      address executor,
      address receiver,
      uint256 expectedDelay,
      uint256 expectedGracePeriod
  ```

```
) private view {
    Executor executorContract = Executor(executor);
    ArbitrumReceiver receiverContract = ArbitrumReceiver(receiver);

    // Verify Executor configuration
    require(executorContract.delay() == expectedDelay, "Executor delay mismatch");
    require(executorContract.gracePeriod() == expectedGracePeriod, "Executor grace period mismatch");

    // Verify Receiver configuration
    require(receiverContract.l1Authority() == Ethereum.GROVE_PROXY, "Receiver authority mismatch");
    require(receiverContract.target() == executor, "Receiver target mismatch");

    console.log("  Constructor parameters verified");
}
```

- Role and Permission Verification:

```
function _verifyPermissions(address executor, address receiver, address deployer) private view {
    Executor executorContract = Executor(executor);

    // Verify role assignments
    bool receiverHasSubmission = executorContract.hasRole(executorContract.SUBMISSION_ROLE(), receiver);
    bool deployerHasAdmin = executorContract.hasRole(executorContract.DEFAULT_ADMIN_ROLE(), deployer);
    bool executorHasSelfAdmin = executorContract.hasRole(executorContract.DEFAULT_ADMIN_ROLE(),
    ↪  executor);

    require(receiverHasSubmission, "Receiver missing SUBMISSION_ROLE");
    require(!deployerHasAdmin, "Deployer still has DEFAULT_ADMIN_ROLE");
    require(executorHasSelfAdmin, "Executor missing self-admin role");

    console.log("  Permissions configured correctly");
    console.log("   Receiver has submission role:", receiverHasSubmission);
    console.log("   Deployer admin revoked:", !deployerHasAdmin);
    console.log("   Executor self-admin:", executorHasSelfAdmin);
}
```

**Steakhouse:** Acknowledged. There is a post-deployment verification suite in the spell repository (which was out of scope).

**Cantina Managed:** Acknowledged.

### 3.1.4 Missing `msg.value` sanity check

**Severity:** Informational

**Context:** ArbitrumERC20Forwarder.sol#L52

**Description:** The `sendMessageL1toL2()` function in the ArbitrumERC20Forwarder library hardcodes the `l2CallValue` parameter to 0 when calling `createRetryableTicket()`, but does not validate that `msg.value == 0`. While the current implementation is safe because the only calling function (`ArbitrumERC20ForwarderExecutor.sendMessageL1toL2()`) is not payable, this could become problematic if the library is reused in the future by a payable function. If a payable function were to call this library function with `msg.value > 0`, the sent ether may become locked in the contract since the function explicitly sets `l2CallValue = 0` in the retryable ticket creation.

**Recommendation:** Consider adding a sanity check to ensure `msg.value == 0` at the beginning of the `sendMessageL1toL2()` function to prevent potential misuse:

```
  function sendMessageL1toL2(
      address l1CrossDomain,
      address target,
      bytes memory message,
      uint256 gasLimit,
      uint256 maxFeePerGas,
      uint256 baseFee
  ) internal {
+     require(msg.value == 0, "ArbitrumERC20Forwarder/unexpected-msg-value");

      ]]]
```

**Steakhouse:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.1.5 Excess gas fees are burned instead of refunded

**Severity:** Informational

**Context:** ArbitrumERC20Forwarder.sol#L54-L55

**Summary:** The `ArbitrumERC20Forwarder` burns excess gas fees instead of refunding them to the caller. When gas estimation is conservative or network conditions change, unused fees are permanently lost rather than returned to the governance system.

**Finding Description:** The `sendMessageL1toL2()` function in ArbitrumERC20Forwarder sets both refund addresses to `address(0)`, causing any excess gas fees to be burned rather than refunded:

```
ICrossDomainArbitrum(l1CrossDomain).createRetryableTicket(
    target,
    0, // we always assume that l2CallValue = 0
    maxSubmission,
    address(0), // burn the excess gas
    address(0), // burn the excess gas
    gasLimit,
    maxFeePerGas,
    maxSubmission + gasLimit * maxFeePerGas, // max redemption fee
    message
);
```

**Recommendation:** Consider modifying the gas refund addresses to return excess fees to the message sender. This approach would recover unused gas fees while maintaining the same external interface:

```
ICrossDomainArbitrum(l1CrossDomain).createRetryableTicket(
    target,
    0, // we always assume that l2CallValue = 0
    maxSubmission,
    msg.sender, // refund excess submission fee to sender
    msg.sender, // refund excess call value to sender
    gasLimit,
    maxFeePerGas,
    maxSubmission + gasLimit * maxFeePerGas,
    message
);
```

Alternatively, consider adding a refund recipient parameter to provide flexibility in refund destination.

Note: If this library is updated, consider also updating the ArbitrumForwarder implementation to maintain consistency across the forwarder library and ensure both ETH and ERC20 variants handle gas refunds in the same manner.

**Steakhouse:** Acknowledged. Decided to keep it as is to be sure to avoid any unexpected behaviour. The amount of tokens stored in the contract performing this operation will be small enough to be expendable.

**Cantina Managed:** Acknowledged.

### 3.1.6 Redundant call to `calculateRetryableSubmissionFee()`

**Severity:** Informational

**Context:** ArbitrumERC20Forwarder.sol#L46

**Finding Description:** `sendMessageL1toL2()` is calling `calculateRetryableSubmissionFee()` to calculate the max submission while as we can see in `ERC20Inbox` will always return 0 and therefore is redundant.

**Recommendation:** Consider removing the call to `calculateRetryableSubmissionFee()` as well as `maxSub-mission` and `baseFee` to simplify the function and save gas.

**Steakhouse:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.1.7 `sendMessageL1toL2()`: Redundant local variables

**Severity:** Informational

**Context:** ArbitrumERC20Forwarder.sol#L43

**Description:** `l1CrossDomain` is redundant since the code reverts in case it is not `L1_CROSS_DOMAIN_PLUME` as we can see:

```
if (l1CrossDomain == L1_CROSS_DOMAIN_PLUME) gasToken = IERC20(PLUME_GAS_TOKEN);
else revert("ArbitrumERC20Forwarder/invalid-l1-cross-domain");
```

**Recommendation:** Consider removing `l1CrossDomain` and use `L1_CROSS_DOMAIN_PLUME` instead. `gasToken` can also be removed and `PLUME_GAS_TOKEN` can be used instead.

**Steakhouse:** Acknowledged.

**Cantina Managed:** Acknowledged.