

Crown: BRL

Security Review

Cantina Managed review by:
Devtooligan, Lead Security Researcher
0xluk3, Associate Security Researcher

March 20, 2026

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**
- 2.1 Scope . . . 3

3 Findings **4**
- 3.1 Medium Risk . . . 4
  - 3.1.1 Batched cross-chain operations fail due to transaction hash collision in replay protection 4
  - 3.1.2 Inconsistent blocklist implementation . . . 5
- 3.2 Low Risk . . . 6
  - 3.2.1 BRLY allows zero-share mints leading to accounting inconsistency in BRLV . . . 6
  - 3.2.2 Inconsistent pause mechanism allows operations to bypass emergency stop . . . 7
  - 3.2.3 Gateway `bridgedSharesMint()` allows unbacked remote supply inflation by skipping surplus validation . . . 9
  - 3.2.4 BRLY rounding mismatch causes bridged burn transactions to revert . . . 10
  - 3.2.5 New chain configurations are enabled by default . . . 12
- 3.3 Informational . . . 12
  - 3.3.1 Bridge integrity hardening . . . 12
  - 3.3.2 Chain reorganization resilience depends on relayer finality handling . . . 12

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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

From Jan 26th to Jan 30th the Cantina team conducted a review of Crown - BRL on commit hash ab7bdef7. The team identified a total of **9** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	2	0
Low Risk	5	3	2
Gas Optimizations	0	0	0
Informational	2	0	2
Total	**9**	**5**	**4**

2.1 Scope

The security review had the following components in scope for Crown - BRL on commit hash ab7bdef7:

```
contracts
├── bridgedBRLV.sol
├── BRLV.sol
├── BRLVCanonicalGateway.sol
├── BRLVRewards.sol
├── BRLY.sol
├── wBRLY.sol
└── YieldStripping.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Batched cross-chain operations fail due to transaction hash collision in replay protection

Severity: Medium Risk

Context: bridgedBRLV.sol#L119, BRLVCanonicalGateway.sol#L176, BRLVCanonicalGateway.sol#L181

Summary: The bridged BRLV system uses transaction hashes as the sole replay-protection key, which prevents processing multiple bridge events emitted within a single source-chain transaction. When privileged operators batch multiple mint or burn requests into one transaction, only the first event can be processed on the destination chain while subsequent events revert, creating stuck bridge requests and accounting mismatches.

Description: The replay-protection mechanism in `bridgedBRLV.mint()` and `BRLVCanonicalGateway.bridgedBurn()` keys idempotency checks solely on transaction hash. In `bridgedBRLV.sol`, the `mint()` function stores `_processedTxHashes[canonicalTxHash]` and reverts with `BridgedBRLVTxHashAlreadyProcessed` on duplicate calls:

```
function mint(
    bytes32 canonicalTxHash,
    bytes32 canonicalDomain_,
    uint256 amount,
    address to,
    uint256 rewardMultiplier
) external onlyRole(MINTER_ROLE) {
    // Check for replay attack
    if (_processedTxHashes[canonicalTxHash]) {
        revert BridgedBRLVTxHashAlreadyProcessed(canonicalTxHash);
    }
    // Mark transaction as processed
    _processedTxHashes[canonicalTxHash] = true;
    // ...
}
```

Similarly, `BRLVCanonicalGateway.bridgedBurn()` uses `_processedRemoteTxHashes[remoteTxHash]` as the idempotency key.

When an operator with `MINTER_ROLE` batches multiple `bridgedMint()` calls in a single canonical-chain transaction, the gateway emits multiple `BridgedMintRequested` events that all share the same transaction hash. The relayer then attempts to complete each mint on the remote chain by calling `bridgedBRLV.mint()` with the same `canonicalTxHash` parameter for each event. The first call succeeds and marks the hash as processed; subsequent calls for the remaining events revert because the contract cannot distinguish between them using only the transaction hash.

A symmetric scenario occurs for burns: if an account with `BURNER_ROLE` on the remote chain burns tokens multiple times in one transaction, the `BridgedBurnRequested` events share the same remote transaction hash. When the relayer attempts to finalize all burns on the canonical chain via `BRLVCanonicalGateway.bridgedBurn()`, only the first completion succeeds.

Recommendation: Consider implementing per-event unique identifiers for replay protection by adding a `uint256 logIndex` parameter to both functions and modifying the replay-protection mappings to use a composite key:

```
  // In bridgedBRLV.sol
- mapping(bytes32 => bool) private _processedTxHashes;
+ mapping(bytes32 => mapping(uint256 => bool)) private _processedTxHashes;

  function mint(
      bytes32 canonicalTxHash,
+     uint256 logIndex,
      bytes32 canonicalDomain_,
      uint256 amount,
      address to,
      uint256 rewardMultiplier
```

```
  ) external onlyRole(MINTER_ROLE) {
      if (canonicalDomain_ != canonicalDomain) {
          revert BridgedBRLVInvalidCanonicalDomain(canonicalDomain_, canonicalDomain);
      }

-     if (_processedTxHashes[canonicalTxHash]) {
+     if (_processedTxHashes[canonicalTxHash][logIndex]) {
          revert BridgedBRLVTxHashAlreadyProcessed(canonicalTxHash);
      }

-     _processedTxHashes[canonicalTxHash] = true;
+     _processedTxHashes[canonicalTxHash][logIndex] = true;

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

Apply the same pattern to `BRLVCanonicalGateway.bridgedBurn()`. The off-chain relayer should extract the `logIndex` from transaction receipts when processing bridge events. Alternatively, consider introducing an explicit incrementing `requestId` emitted from the source chain that serves as a globally unique identifier, though this approach requires additional state management on the source chain.

Crown: Fixed in PR 47.

Cantina Managed: Fix verified.

3.1.2 Inconsistent blocklist implementation

Severity: Medium Risk

Context: wBRLY.sol#L118

Summary: The blocklist implementation is inconsistent and contain few flaws that may allow blocked users to still interact with the protocol.

Description: There are two misconfigurations of the blocklist:

- Missing spender check in `transferFrom` in (`wBRLY`, `BRLV`, `bridgedBRLV`):

All three contracts inherit the standard `ERC20Upgradeable.transferFrom` without checking if `msg.sender` is blocked. In contrast, `BRLY.sol` explicitly overrides `transferFrom` to validate the spender:

```
function transferFrom(address from, address to, uint256 amount) external returns (bool) {
    address spender = _msgSender();

    if (isBlocked(spender)) {
        revert BRLYBlockedSender(spender);
    }
    [...]
```

A blocked address with pre-existing approvals can still execute `transferFrom` calls as the spender. This may happen for instance, if an phishing is executed against users, and the phisher will still be able to move funds around since only target or origin addresses can be blocked.

- Missing receiver check in `wBRLY`:

Unlike BRLV and bridgedBRLV which check both from and to against the blocklist, `wBRLY` only checks the sender.

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual
↪  override {
   // Each blocklist check is an SLOAD, which is gas intensive.
   // We only block sender not receiver, so we don't tax every user
   if (BRLY.isBlocked(from)) {
       revert wBRLYBlockedSender(from);
   }
```

This allows tokens to be transferred to blocked addresses in `wBRLY`.

Impact Explanation: Blocked addresses can bypass restrictions by using existing approvals as spenders, and can receive wBRLY tokens directly.

Recommendation: Override `transferFrom` in all three contracts to check `isBlocked(msg.sender)`. Add receiver blocklist check to `wBRLY._beforeTokenTransfer`.

Crown: Fixed in PR 42.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 BRLY allows zero-share mints leading to accounting inconsistency in BRLV

Severity: Low Risk

Context: BRLY.sol#L247

Summary: The BRLY contract's share-based accounting allows minting operations that result in zero shares when small token amounts round down during conversion. This can cause accounting inconsistencies in the BRLV wrapper contract where BRLV shares are issued without corresponding BRLY backing, breaking vault invariants.

Description: When BRLY's `rewardMultiplier` exceeds 1e18 (after positive rebases), the `convertToShares()` function can round small token amounts down to zero shares. This occurs when `amount < rewardMultiplier / 1e18`.

In BRLV's `mint()` function, which is restricted to MINTER_ROLE holders, calling `BRLY.mint(address(this), amount)` with such small amounts results in zero BRLY shares being minted to the vault, while BRLV still proceeds to mint the requested amount of BRLV shares to the recipient.

This edge case breaks the vault's fundamental invariant that `totalAssets >= totalSupply`. When this invariant is violated, surplus-dependent operations like `claimRewardsV2()` and `claimTax()` will fail due to arithmetic underflow when calculating `totalAssets() - totalSupply()`.

While this requires MINTER_ROLE access (which is only granted to trusted protocol components), the contract should prevent such edge cases to maintain consistent accounting and avoid potential integration issues.

Recommendation: Consider adding validation in the BRLY contract to prevent mints and burns that would result in zero shares:

```
  function _mint(address to, uint256 amount) private {
      if (to == address(0)) {
          revert BRLYInvalidMintReceiver(to);
      }

      _beforeTokenTransfer(address(0), to, amount);

      uint256 shares = convertToShares(amount);
+     if (shares == 0) {
+         revert BRLYZeroShareMint(amount);
+     }
      _totalShares += shares;

      unchecked {
          _shares[to] += shares;
      }

      _afterTokenTransfer(address(0), to, amount);
  }

  function _burn(address account, uint256 amount) private {
      if (account == address(0)) {
          revert BRLYInvalidBurnSender(account);
      }
```

```
      _beforeTokenTransfer(account, address(0), amount);

      uint256 shares = convertToSharesRoundUp(amount);
+     if (shares == 0) {
+         revert BRLYZeroShareBurn(amount);
+     }

      uint256 accountShares = sharesOf(account);
      if (accountShares < shares) {
          revert BRLYInsufficientBurnBalance(account, accountShares, shares);
      }

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

Crown: Fixed in PR 46.

Cantina Managed: Fix verified.

3.2.2 Inconsistent pause mechanism allows operations to bypass emergency stop

Severity: Low Risk

Context: BRLVCanonicalGateway.sol#L135, BRLV.sol#L199-L202, BRLV.sol#L241

Summary: The BRLV protocol implements a pause mechanism intended as an emergency stop for all token operations. However, several critical functions across BRLV, BRLVCanonicalGateway, and BRLVRewards contracts do not respect the pause state, allowing minting, burning, and cross-chain operations to continue during emergencies. This undermines the effectiveness of the pause functionality as a security control.

Description: The BRLV contract inherits from `PausableUpgradeable` and implements pause checks in its `_beforeTokenTransfer()` hook, which affects `transfer()`, `transferFrom()`, `mint()`, `burn()`, `mintShares()`, and `burnShares()` functions. However, two permissioned functions bypass this protection:

```
// BRLV.sol - These functions lack pause checks
function mintAssets(uint256 amount) external onlyRole(MINTER_ROLE) {
    _mint(address(this), amount);  // This will be paused
    _totalAssets += amount;
    emit AssetsMinted(amount);
}

function burnAssets(uint256 amount) external onlyRole(BURNER_ROLE) {
    _burn(address(this), amount);  // This will be paused
    _totalAssets -= amount;
    emit AssetsBurned(amount);
}
```

Note: Upon closer inspection, `mintAssets()` and `burnAssets()` call `_mint()` and `_burn()` respectively, which do trigger `_beforeTokenTransfer()` and will be paused. However, the `BRLVCanonicalGateway.bridgedSharesMint()` function presents a genuine gap:

```
// BRLVCanonicalGateway.sol
function bridgedSharesMint(
    uint256 amount,
    bytes32 domain,
    address to
) external onlyRole(MINTER_ROLE) whenNotPaused {
    _validateBridgedMintParams(amount, domain, to);

    // No collateral minting - uses existing surplus
    // Track bridged supply
    totalBridgedSupply += amount;

    // Get current reward multiplier
    uint256 rewardMultiplier = brly.rewardMultiplier();
```

```
    // Emit event for relayer - NO BRLV INTERACTION, NO PAUSE CHECK
    emit BridgedMintRequested(amount, domain, to, rewardMultiplier);
}
```

This function only checks its own pause state via `whenNotPaused`, but does not verify whether BRLV itself is paused. When BRLV is paused during an emergency but BRLVCanonicalGateway is not, reward claims to remote chains can still proceed through `BRLVRewards.claimRewards()`:

```
// BRLVRewards.sol
function claimRewards(...) external onlyRole(MINTER_ROLE) whenNotPaused {
    // ...
    if (domain == canonicalDomain) {
        brlv.mintShares(to, amount);  // Will revert if BRLV paused
    } else {
        gateway.bridgedSharesMint(amount, domain, to);  // Succeeds even if BRLV paused
    }
    // ...
}
```

The inconsistency creates a situation where:

- Canonical chain operations are properly blocked when BRLV is paused.
- Remote chain reward claims can still proceed, creating an asymmetric emergency response.

Contract	Function	Respects BRLV Pause?
BRLV	transfer()/transferFrom()	Yes
BRLV	mint()/burn()	Yes
BRLV	mintShares()/burnShares()	Yes
BRLVCanonicalGateway	bridgedMint()	Yes (uses mintAssets)
BRLVCanonicalGateway	bridgedBurn()	Yes (uses burnAssets)
BRLVCanonicalGateway	bridgedSharesMint()	**No**
BRLVRewards	claimRewards() (canonical)	Yes
BRLVRewards	claimRewards() (remote)	**No**

Impact Explanation: During an emergency that requires pausing the BRLV system, cross-chain reward minting operations can continue to function. This could allow the protocol to continue emitting `BridgedMintRequested` events that the off-chain relayer might process, potentially minting tokens on remote chains when the canonical chain is in an emergency state. The partial effectiveness of the pause mechanism reduces confidence in the protocol's ability to halt operations during critical situations.

Likelihood Explanation: This issue requires an emergency scenario where BRLV needs to be paused but the operator either forgets to pause all related contracts or intentionally wants asymmetric behavior. While the likelihood of exploitation is low given the permissioned nature of the affected functions, the inconsistency represents a design flaw that could lead to unexpected behavior during high-stress emergency situations.

Recommendation: Consider adding a check for BRLV's pause state in `BRLVCanonicalGateway.bridgedSharesMint()`:

```
  function bridgedSharesMint(
      uint256 amount,
      bytes32 domain,
      address to
  ) external onlyRole(MINTER_ROLE) whenNotPaused {
+     if (brlv.paused()) {
+         revert GatewayBRLVPaused();
+     }
      _validateBridgedMintParams(amount, domain, to);
      // ... rest of function
  }
```

Additionally, consider implementing a coordinated emergency pause mechanism that can halt operations across all related contracts simultaneously. This could be achieved through:

1. A shared pause registry contract that all protocol contracts check.
2. A multi-contract pause function callable by emergency administrators.
3. An off-chain script that can pause multiple contracts.

Crown: Fixed in PR 45.

Cantina Managed: Fix verified.

3.2.3 Gateway `bridgedSharesMint()` allows unbacked remote supply inflation by skipping surplus validation

Severity: Low Risk

Context: BRLVCanonicalGateway.sol#L139, BRLVRewards.sol#L117

Summary: The `bridgedSharesMint()` function in `BRLVCanonicalGateway` increments `totalBridgedSupply` without verifying that sufficient surplus collateral exists. While `BRLVRewards.claimRewards()` performs this validation before calling the gateway, any account with `MINTER_ROLE` on the gateway can bypass this check by calling `bridgedSharesMint()` directly, potentially creating unbacked remote supply that breaks the protocol's collateralization invariant.

Description: The `bridgedSharesMint()` function is designed to mint shares on remote chains against existing surplus collateral without creating new assets. However, the function does not enforce that this surplus actually exists:

```
function bridgedSharesMint(
    uint256 amount,
    bytes32 domain,
    address to
) external onlyRole(MINTER_ROLE) whenNotPaused {
    _validateBridgedMintParams(amount, domain, to);

    // No collateral minting - uses existing surplus
    // Track bridged supply
    totalBridgedSupply += amount;  // No surplus validation

    uint256 rewardMultiplier = brly.rewardMultiplier();

    emit BridgedMintRequested(amount, domain, to, rewardMultiplier);
}
```

While `BRLVRewards.claimRewards()` correctly validates available surplus before calling `bridgedSharesMint()`, this check can be bypassed by any gateway minter calling the function directly.

When `totalBridgedSupply` is inflated without corresponding collateral, `availableSurplus()` returns a reduced or zero value since it computes `totalAssets - (canonicalSupply + totalBridgedSupply)`. Because `totalBridgedSupply` is a single global scalar, this inflation affects all domains simultaneously. Remote holders who later attempt to burn their tokens trigger `bridgedBurn()` calls that invoke `brlv.burnAssets()`, which may revert if the BRLV contract lacks sufficient BRLY balance.

Impact Explanation: Unbacked remote supply breaks the core collateralization invariant, leaving remote BRLV partially or fully unbacked. Remote burn and redemption flows may become uncompletable because `brlv.burnAssets()` will revert when insufficient BRLY exists. The reduced global `availableSurplus()` can also block legitimate reward claims across all domains, not just the one targeted.

Likelihood Explanation: Exploitation requires `MINTER_ROLE` on the gateway, which is privileged but may be granted to operational components or integrations. The attack is a single function call with no external dependencies, though it relies on privileged access or misconfiguration.

Recommendation: Consider adding surplus validation directly to `bridgedSharesMint()` to ensure enforcement regardless of the call path:

```
  function bridgedSharesMint(
      uint256 amount,
      bytes32 domain,
      address to
  ) external onlyRole(MINTER_ROLE) whenNotPaused {
      _validateBridgedMintParams(amount, domain, to);

+     // Ensure sufficient surplus exists
+     uint256 currentSurplus = availableSurplus();
+     if (amount > currentSurplus) {
+         revert GatewayInsufficientSurplus(amount, currentSurplus);
+     }

      // No collateral minting - uses existing surplus
      totalBridgedSupply += amount;

      uint256 rewardMultiplier = brly.rewardMultiplier();

      emit BridgedMintRequested(amount, domain, to, rewardMultiplier);
  }
```

With this check in place, the redundant validation in `BRLVRewards.claimRewards()` could be removed. Additionally, consider introducing a dedicated role (e.g., `SHARES_MINTER_ROLE`) granted only to the Rewards contract, providing defense-in-depth by preventing arbitrary gateway minters from bypassing the intended surplus checks.

Crown: Acknowledged.

Cantina Managed: Acknowledged.

3.2.4 BRLY rounding mismatch causes bridged burn transactions to revert

Severity: Low Risk

Context: BRLVCanonicalGateway.sol#L114

Summary: The BRLY token uses asymmetric rounding in its share-to-token conversions, where `mint()` rounds down and `burn()` rounds up. After any yield accrual shifts the `rewardMultiplier` above 1e18, bridged burn operations through the gateway will revert because BRLV holds slightly fewer shares than required to burn the originally minted token amount. This is a low severity issue as transactions can be retried with a 1 wei adjustment and no funds are at risk.

Description: BRLY implements a share-based rebasing model where token balances are calculated by multiplying shares by a `rewardMultiplier`. The `_mint()` function converts token amounts to shares using `convertToShares()`, which rounds down:

```
function convertToShares(uint256 amount) public view returns (uint256) {
    return (amount * _BASE) / rewardMultiplier;
}
```

The `_burn()` function uses `convertToSharesRoundUp()`, which rounds up:

```
function convertToSharesRoundUp(uint256 amount) public view returns (uint256) {
    return (amount * _BASE + rewardMultiplier - 1) / rewardMultiplier;
}
```

When `BRLVCanonicalGateway.bridgedMint()` mints collateral via `brlv.mintAssets(amount)`, the underlying BRLY mint rounds shares down. When `bridgedBurn()` later attempts to burn that same `amount`, BRLY's burn logic rounds shares up, requiring one more share than BRLV possesses.

For example, minting 100 tokens when `rewardMultiplier = 1.001e18` credits `floor(100 * 1e18 / 1.001e18) = 99900099900099900099` shares to BRLV. A subsequent burn of 100 tokens requires `ceil(100 * 1e18 / 1.001e18) = 99900099900099900100` shares-one more than available-causing the transaction to revert with `BRLYInsufficientBurnBalance`.

This affects nearly every transaction once `rewardMultiplier` deviates from 1e18. Only amounts that are exact multiples of the `rewardMultiplier` would work, which is effectively random and not something operators would control.

Impact Explanation: The impact is low. No funds are at risk as transactions simply revert and can be retried. The economic impact is negligible-only 1 wei of dust per operation. Since these are permissioned operations requiring `MINTER_ROLE` or `BURNER_ROLE`, operators can adapt their tooling to account for the adjustment.

Likelihood Explanation: The likelihood is high as this occurs on every bridge operation after the first yield accrual shifts `rewardMultiplier` above 1e18. No attacker action is required; the mismatch is deterministic.

Proof of Concept:

```
it('blocks bridged burn because minted collateral is short due to rounding', async () =>
↪  {
    // Step 1: accrue yield so rewardMultiplier > 1e18
    const increment = parseUnits('0.001'); // +0.1%
    await brly.connect(oracle).addRewardMultiplier(increment);

    // Step 2: Request bridged mint of 100 BRLV
    const bridgedAmount = parseUnits('100');
    await gateway.connect(minter).bridgedMint(bridgedAmount, remoteDomain,
    ↪  remoteUser.address);

    // Collateral actually minted is rounded down in share space
    const brlvShares = await brly.sharesOf(await brlv.getAddress());
    const collateralTokens = await brly.convertToTokens(brlvShares);
    expect(collateralTokens).to.be.lt(bridgedAmount); // collateral < liabilities

    // Step 3: Remote burn completion tries to burn the full 100 tokens - reverts
    const requiredShares = await brly.convertToSharesRoundUp(bridgedAmount);
    await expect(
        gateway.connect(relayer).bridgedBurn(remoteTxHash, remoteDomain,
        ↪  remoteUser.address, bridgedAmount)
    ).to.be.revertedWithCustomError(brly, 'BRLYInsufficientBurnBalance')
     .withArgs(await brlv.getAddress(), brlvShares, requiredShares);
});
```

Recommendation: The simplest fix is to document this behavior and have operators burn `amount - 1` instead of `amount` when completing bridged burns. However, this creates a minor accounting mismatch where 1 wei of collateral per operation remains locked in BRLV.

For a cleaner solution, consider converting gateway operations to share-based accounting:

```
  function bridgedMint(uint256 amount, bytes32 domain, address to) external
  ↪  onlyRole(MINTER_ROLE) whenNotPaused {
      _validateBridgedMintParams(amount, domain, to);

+     // Compute actual collateral based on share rounding
+     uint256 shares = brly.convertToShares(amount);
+     uint256 actualCollateral = brly.convertToTokens(shares);

-     brlv.mintAssets(amount);
-     totalBridgedSupply += amount;
+     brlv.mintAssets(actualCollateral);
+     totalBridgedSupply += actualCollateral;

      uint256 rewardMultiplier = brly.rewardMultiplier();
-     emit BridgedMintRequested(amount, domain, to, rewardMultiplier);
+     emit BridgedMintRequested(actualCollateral, domain, to, rewardMultiplier);
  }
```

This ensures the recorded liability matches the actual collateral credited, eliminating the mismatch at the source.

Crown: Acknowledged.

Cantina Managed: Acknowledged.

3.2.5 New chain configurations are enabled by default

Severity: Low Risk

Context: BRLVCanonicalGateway.sol#L207

Description: The `setChainConfig` function automatically enables newly configured chains, which could lead to premature activation of bridge routes before full operational readiness is confirmed.

When adding a new remote chain configuration via `setChainConfig`, the enabled flag is automatically set to true. Administrators have to explicitly disable the chain if not ready for operation, rather than following a safer default-disabled approach.

If configuration is performed before the remote chain deployment is fully operational, there is a window where the bridge route may be active but potentially causing failed bridging operations.

Recommendation: Initialize new chain configurations with `enabled: false` by default and require explicit enablement via `enableChain()`.

Crown: Fixed in PR 44.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Bridge integrity hardening

Severity: Informational

Context: bridgedBRLV.sol#L109

Description: The mint function accepts arbitrary parameters from the `MINTER_ROLE` without on-chain verification against the canonical chain transaction, relying entirely on relayer integrity.

While the `MINTER_ROLE` is trusted, the logic could be strengthened. A compromised relayer or misconfigured backend could mint arbitrary amounts without on-chain constraints. The `canonicalTxHash` is used only for replay protection. The `amount`, `to`, and `rewardMultiplier` parameters are not verified to match the original transaction.

Recommendation: Consider implementing additional verification such as `keccak256(abi.encode(amount, to, rewardMultiplier, nonce)` which could allow to binding specific amounts and parameter values to original request.

Crown: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 Chain reorganization resilience depends on relayer finality handling

Severity: Informational

Context: BRLVCanonicalGateway.sol#L181

Description: Once a transaction is marked as processed, there is no mechanism to revert this state. If the relayer prematurely or incorrectly marks a transaction as processed (e.g., a blockchain reorg happens), it cannot be reprocessed.

Recommendation: Since the relayer is out of scope, ensure it waits for sufficient block confirmations before submitting transactions.

Crown: Acknowledged.

Cantina Managed: Acknowledged.