Linea - Burn Mechanism

Date	October 2025
Auditors	Arturo Roura, Martin Ortner

1 Executive Summary

This report presents the results of our engagement with **Linea** to review the new **Linea Revenue Profit Burn Mechanism**.

The review was conducted from **October 13, 2025** to **October 20, 2025**, with a total effort of **2 x 5** person-days.

The Linea Burn Mechanism consists of a three-contract system designed to facilitate the systematic burning of `LINEA` tokens as part of Linea's rollup revenue management strategy. This operational infrastructure enables the collection of rollup revenues, conversion to `LINEA` tokens, and their subsequent destruction to reduce token supply.

The burn mechanism operates through a coordinated cross-chain workflow:

- Revenue Collection through `RollupRevenueVault` which receives `ETH` from Linea rollup operations and processes invoices for operational expenses
- A Token Conversion where during burn operations, 20% of available `ETH` is permanently destroyed while the remaining 80% is swapped for `LINEA` tokens via the `V3DexSwap` contract
- Acquired `LINEA` tokens are bridged from L2 to L1 using Linea's canonical token bridge to the `L1LineaTokenBurner` contract
- The L1 burner contract processes cross-chain messages and permanently burns all received `LINEA` tokens, synchronizing the reduced supply back to L2

Remarks on Validated Design Decissions:

In `RollupRevenueVault` there is only one active receiver at a time. The system does not support multiple simultaneous receivers, as the debt system enforces a single counterparty. Operationally, this receiver is separate from the invoice submitter, who remains a trusted role.

`RollupRevenueVault` can receive `ETH` through `fallback()` and `receive()` without requiring invoice submission. This `ETH` becomes immediately available for burning if there are no outstanding arrears, which could bypass standard revenue accounting and invoice workflows. These functions are primarily intended to accept `ETH` from contributors wishing to fund the burn. Off-chain procedures and timing routines are carefully managed to ensure that invoices are settled first, allowing any excess `ETH` to be safely burned.

Neither `RollupRevenueVault` nor `V3DexSwap` enforce slippage control by themselve, placing critical responsibility on the caller to correctly parameterize `burnAndBridge(bytes calldata _swapData)` with appropriate slippage settings. The contracts trust the configured DEX router to return at least the caller provided `amountOutMinimum`, there is no explicit verification. This may be valid, however, the caller should consider implementing additional verification mechanisms to ensure swap outputs meet expected thresholds, as incorrect parameterization could result in excessive slippage or MEV exploitation.

The `RollupRevenueVault` admin, overseen by a distributed security council including non-Linea and non-CSI members, can adjust arrears via `updateInvoiceArrears` to correct reporting inaccuracies; this capability is tightly controlled and expected to be used only in rare cases of major reporting errors.

Here's a more concise and polished version suitable for an **executive summary** section of an audit report. It keeps all the technical detail but presents it with clearer structure and formal tone:

1.1 Update – October 24 + 30, 2025

All issues identified in this report have been resolved. The final audited codebase is represented by commit efe83ff992b38eda5fd5a58220acb6952c519f75 (tag: `contract-audit-2025-10-30`), which contains the verified deployment artifacts.

1.2 Update – November 2, 2025

At the request of the Linea team, the deployed contracts have been added and verified as follows.

Deployment Commit: linea-monorepo@ `contract-audit-2025-10-30` (`efe83ff992b38eda5fd5a58220acb6952c519f75`)

The listed source files were recompiled from scratch, and the resulting bytecode was compared against the deployed contracts. In cases where an exact match was not observed, the differences were confirmed to correspond to the expected constructor parameters. The findings are as follows:

- RollupRevenueVault@ `0x84a5ba2c12a15071660b0682b59e665dc2faaedb` : Verified on `lineascan.build` (Nov 2, 2025, 15:45 CET). The deployed bytecode is an exact match to the re-compiled code of `contract-audit-2025-10-30@efe83ff` and the bytecode artifacts recorded under `linea-monorepo:contracts/deployments/bytecode/2025-10-27` .
- V3DexSwapAdapter@ `0x30a20a3a9991c939290f4329cb52daac8e97f353` : Verified on `lineascan.build` (Nov 2, 2025, 15:45 CET). The deployed bytecode matches, with expected constructor argument substitutions. The re-compiled bytecode matches the bytecode artifacts recorded under `linea-monorepo:contracts/deployments/bytecode/2025-10-27` .
- L1LineaTokenBurner@ `0x5Ad9369254F29b724d98F6ce98Cb7bAD729969F3` : Verified on `lineascan.build` (Nov 2, 2025, 15:45 CET). The deployed bytecode matches, with expected constructor argument substitutions. The re-compiled bytecode matches the

bytecode artifacts recorded under `linea-monorepo:contracts/deployments/bytecode/2025-10-27`.

The re-compiled bytecode for `V3DexSwapWethDepositAdapter` matches the bytecode artifacts recorded under `linea-monorepo:contracts/deployments/bytecode/2025-10-27`.

2 Scope

This review focused on the following repository and code revision:

- Consensys/linea-monorepo @ contract-freeze-2025-10-12

The following files are in scope:

```
contracts/src/operational/
├── RollupRevenueVault.sol
├── V3DexSwap.sol
├── L1LineaTokenBurner.sol
└── interfaces/
    ├── IV3DexSwap.sol
    ├── IRollupRevenueVault.sol
    ├── ISwapRouterV3.sol
    ├── IL1LineaToken.sol
    ├── IL1LineaTokenBurner.sol
    └── IWETH9.sol
```

The detailed list of files in scope can be found in the Appendix.

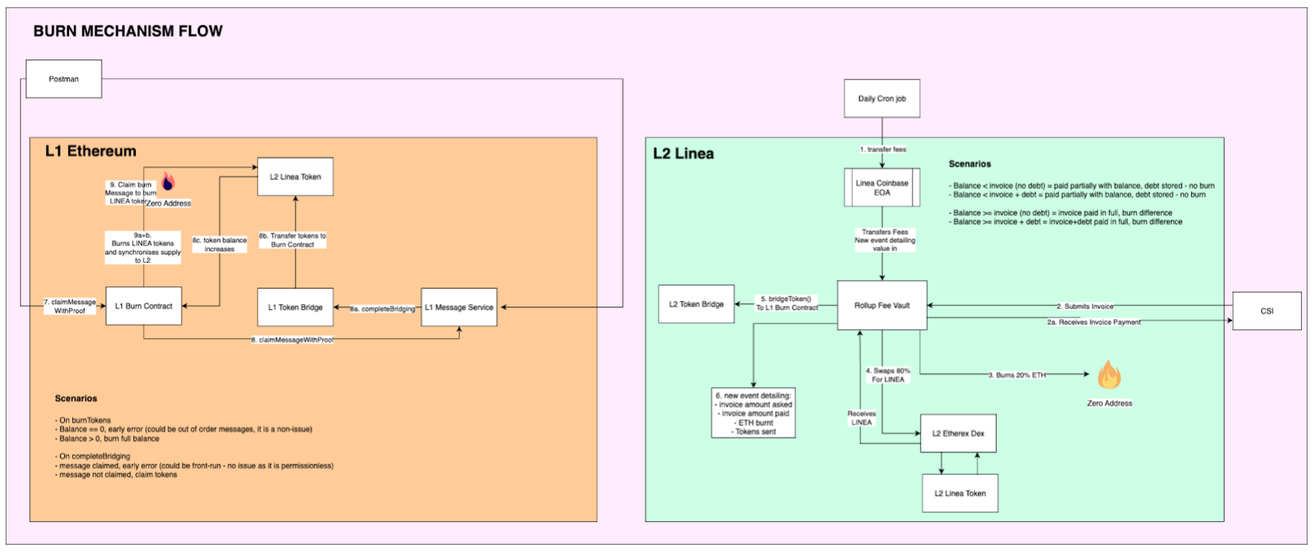
2.1 Objectives

Together with the Linea Team, we identified the following priorities for this review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Proper Handling of Funds, fee collection, profit burning, and token swaps.
4. Swaps comply with industry standards via the defined Dex interface.
5. Documentation and Transparency, operations are clearly documented and understandable by stakeholders.

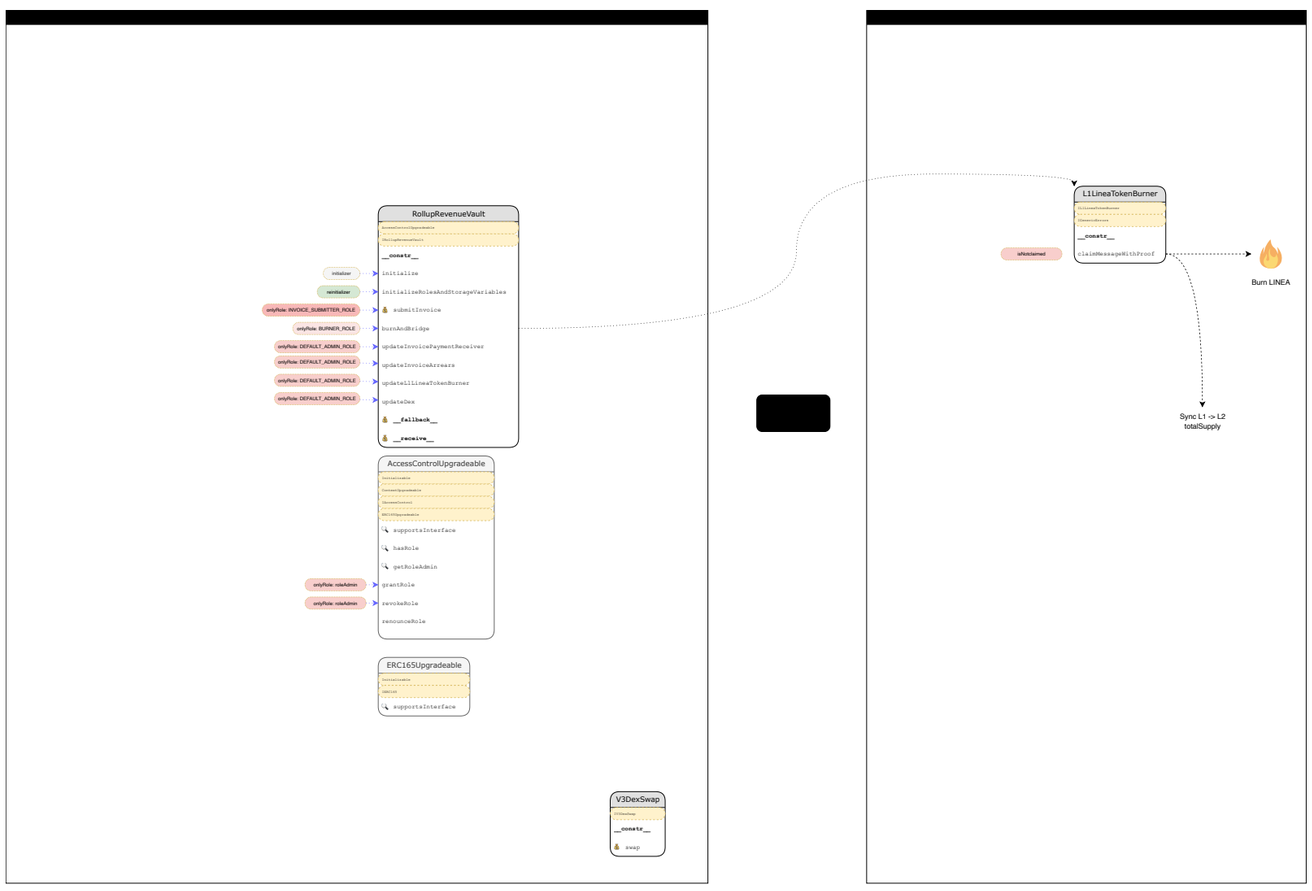
3 System Overview

Burn Mechanism



(This image was provided by the Linea Team)

Contract Overview



3.1 Rollup Revenue Vault (Upgradable)

- Handles Sequencer and L2 DDoS fee collection, invoice management, and profit burning.
- Burns 20% as ETH and swaps 80% to Linea tokens, which are bridged to the L1 Token Burner.
- Supports configuration changes (Dex swap address, invoice recipient, L1 Token Burner).

Technical Details:

- **Upgradeable Contract**: Uses OpenZeppelin's `AccessControlUpgradeable` with reinitializer support (version 2)
- **Role-Based Access Control**:
  - `DEFAULT_ADMIN_ROLE`: Can update system parameters (dex, invoice receiver, L1 burner, invoice arrears)
  - `INVOICE_SUBMITTER_ROLE`: Can submit and process invoices for operational expenses
  - `BURNER_ROLE`: Can execute the burn-and-bridge operations

Revenue Collection & Invoice Management:

- **Revenue Reception**: Accepts ETH through `receive()` and `fallback()` functions, emitting `EthReceived` events
- **Invoice Processing**: Strict sequential invoice system requiring `_startTimestamp == lastInvoiceDate + 1`
- **Arrears Handling**: Automatically calculates and tracks unpaid amounts when contract balance is insufficient
- **Payment Priority**: Invoice payments are processed before any burn operations (`invoiceArrears == 0` requirement)

Burn-and-Bridge Mechanism:

- **Fixed Burn Ratio**: Exactly 20% of available ETH (after message fees) is permanently destroyed via `address(0).call{value: ethToBurn}("")`
- **DEX Integration**: Remaining 80% is sent to the DEX contract via low-level call with encoded swap data
- **Token Bridging**: Acquired LINEA tokens are approved and bridged to L1 using the canonical `TokenBridge.bridgeToken()` function
- **Balance Management**: Reserves L2MessageService minimum fees before calculating burn amounts

Configuration Management:

- **Updateable Components**: DEX address, invoice payment receiver, L1 token burner address, and invoice arrears
- **Validation**: All updates include zero-address checks and existing-address-same prevention

3.2 L1 Linea Token Burner (Non-Upgradable)

- Accepts token bridging messages and burns tokens in a single permissionless transaction.
- Includes safeguards against front-running, duplicate claims, and zero-balance burns.

Technical Details:

- **Immutable Design**: All addresses (`LINEA_TOKEN`, `MESSAGE_SERVICE`) are immutable, preventing configuration changes post-deployment
- **Permissionless Operation**: Anyone can call `claimMessageWithProof()` to trigger token burning

Message Claiming & Burning Process:

- **Idempotent Message Claiming**: Checks `IL1MessageManager.isMessageClaimed()` before attempting to claim, preventing revert on duplicate claims
- **Automatic Token Destruction**: Burns entire contract balance in a single transaction using `IL1LineaToken.burn(balance)`
- **Supply Synchronization**: Calls `IL1LineaToken.syncTotalSupplyToL2()` to update L2 total supply after burning
- **Zero-Balance Protection**: Requires `balance > 0` before executing burn operations

Front-Running & Duplicate Claim Safeguards:

- **Message Status Check**: Prevents claiming already-processed messages
- **Balance Validation**: Ensures tokens exist before burning
- **Atomic Operation**: Combines message claiming and token burning in single transaction

3.3 V3 Dex Swap (L2, Non-Upgradable)

- Executes exact-input token swaps, ensuring no residual ETH remains.
- Token receiver is always the caller funding the swap.

Technical Details:

- **Immutable Configuration**: Router, WETH, LINEA token addresses, and pool tick spacing are set at deployment
- **Uniswap V3 Integration**: Uses `ISwapRouterV3.exactInputSingle()` for precise ETH-to-LINEA swaps

Swap Execution Details:

- **ETH Handling**: Converts incoming ETH to WETH via `IWETH9.deposit{value: msg.value}()`
- **Exact Input Swaps**: Guarantees all input ETH is consumed, leaving zero residual balance
- **Direct Token Delivery**: LINEA tokens are sent directly to `msg.sender` (the caller), never held by the contract
- **Slippage Protection**: Enforces minimum output amounts via `amountOutMinimum` parameter
- **Deadline Protection**: Prevents transaction execution after specified timestamp

Parameter Validation:

- **Input Validation**: Requires `msg.value > 0`, `_deadline > block.timestamp`, and `_minLineaOut > 0`
- **Price Limit Control**: Accepts `_sqrtPriceLimitX96` for additional price protection
- **Return Value**: Returns actual LINEA tokens received from the swap

4 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

4.1 18_deploy_RollupRevenueVault.ts – Deployment Script Leaves Contract Uninitialized; `fallback` Does Not Enforce `msg.value > 0` Major ✓ Fixed

Resolution

Fixed in commit 831c529479faebb380df1478e17f03bf0a3b9b80. The Linea team modified the deployment script to call the initialize function with the proper signature and function arguments, and added `require(msg.value > 0, NoEthSent())` in the fallback.

Description

The deployment script for `RollupRevenueVault` incorrectly attempts to call a non-existent `initialize()` function with no parameters, leaving the contract uninitialized and non-functional.

The script specifies `EMPTY_INITIALIZE_SIGNATURE = "initialize()"`, but the `RollupRevenueVault` contract only defines an `initialize(addr, ...)` function that requires 10 parameters. When `deployProxy` (from OpenZeppelin) executes this empty initializer, the call is routed to the contract's fallback function instead of performing initialization.

As a result, the contract stays uninitialized allowing anyone to claim it.

Examples

- Deployment Script calls empty `initialize()`

contracts/deploy/18_deploy_RollupRevenueVault.ts:L15-L17

```
const contractName = "RollupRevenueVault";
const existingContractAddress = await getDeployedContractAddress(contractName, deployments);
```

contracts/deploy/18_deploy_RollupRevenueVault.ts:L25-L28

```
const contract = await deployUpgradableFromFactory(contractName, [], {
  initializer: EMPTY_INITIALIZE_SIGNATURE,
  unsafeAllow: ["constructor"],
});
```

- Contract only defines `initialize(uint256,address,address,address,address,address,address,address,address,address)`

Since no zero-argument `initialize()` exists, calls to `initialize()` without parameters trigger the fallback, contributing to hiding this issue. Deployment returns success.

contracts/src/operational/RollupRevenueVault.sol:L58-L74

```
function initialize(
  uint256 _lastInvoiceDate,
  address _defaultAdmin,
  address _invoiceSubmitter,
  address _burner,
  address _invoicePaymentReceiver,
  address _tokenBridge,
  address _messageService,
  address _l1LineaTokenBurner,
  address _lineaToken,
  address _dex
) external initializer {
  __AccessControl_init();
  __RollupRevenueVault_init(
    _lastInvoiceDate,
    _defaultAdmin,
    _invoiceSubmitter,
```

contracts/src/operational/RollupRevenueVault.sol:L282-L288

```
/**
 * @notice Fallback function - Receives Funds.
 */
fallback() external payable {
  emit EthReceived(msg.value);
}
```

Recommendation

- Remove the fallback function or require `msg.value > 0`.

- Update the deployment script to call the correct `initialize(addr, ...)` function and supply all required parameters.
- Bundle with a call to the `reinitialize` function to avoid another front-running window.

4.2 RollupRevenueVault - Deployment and Initialization Flow Medium ✓ Fixed

Resolution

Fixed in commit 831c529479faebb380df1478e17f03bf0a3b9b80. A new execution flow has been created for upgrading the proxy where `upgradeAndCall` calls directly the `initializeRolesAndStorageVariables` function. This function has the `reinitializer(2)` modifier, which after reinitialization would prevent any call to the initialize function.

Description

The contract implements **two** initialization functions:

- `initialize(addr, ...) initializer` – for initial setup after proxy deployment.
- `initializeRolesAndStorageVariables(addr, ...) reinitializer(2)` – available immediately after `initialize` has been called (Version 2 of the contract).

Observations and Concerns

1) Deployment and Initialization Flow

When `initialize` functions are not permissioned, they must be called within the same transaction that deploys the proxy. Otherwise, this creates a window of opportunity for an attacker to front-run initialization and gain control of the contract. With certain proxy patterns (e.g., UUPS), this can even allow an attacker to upgrade the proxy immediately after initialization, enabling arbitrary future initialization or upgrades. See a live example of such an attack.

In this case, the proxy pattern appears to be a minimal transparent proxy (not UUPS) with `upgradeTo() onlyAdmin` and `delegateCall(impl)` branching. It is assumed proxies are deployed and initialized in the same transaction.

The `RollupRevenueVault` contract defines a `reinitializer(2)` function that becomes available immediately after the first initialization. For a typical upgrade path of `[deploy][initialize]-[reinitialize]` this means, that after deployment or an upgrade, anyone could potentially reinitialize the contract unless the upgrade transaction includes an atomic `upgradeToAndCall(reinitializer_function)` call. Each upgrade therefore **must** include the `reinitializer` call in the same transaction to avoid leaving a reinitialization window open.

Originally, our concern was that the assumed deployment process (`deploy → initialize → reinitialize`) introduced gaps between calls. However, the client has clarified that their deployment **directly invokes** `reinitialize` rather than performing a chained initialization. This mitigates the initial frontrunning concern but does not eliminate it entirely - especially if future upgrades expose the `reinitializer(2)` function in isolation or if initialization logic changes.

In the test suite, `reinitialize` is still observed being called in a separate transaction:

contracts/test/hardhat/operational/RollupRevenueVault.ts:L317-L330

```
it("Should revert when reinitializing twice", async () => {
  const txPromise = rollupRevenueVault.initializeRolesAndStorageVariables(
    await time.latest(),
    admin.address,
    invoiceSubmitter.address,
    burner.address,
    invoicePaymentReceiver.address,
    await tokenBridge.getAddress(),
    await messageService.getAddress(),
    l1LineaTokenBurner.address,
    await l2LineaToken.getAddress(),
    await dex.getAddress(),
  );
```

If this pattern is replicated during upgrade execution, it reintroduces a short window of opportunity for attackers to front-run the reinitialization and potentially claim ownership or modify key state.

Scenarios to consider:

- **(a)** `[deploy proxy] ----window of opportunity--- [initialize/reinitializer]` → mitigated by `deployAndInit`
- **(b)** `[initialized contract] ----window of opportunity--- [reinitializer(2)]` → must be mitigated by `upgradeAndReInit` (or `deployAndInit + upgradeAndReInit` atomically for new deployments)

---

2) Cumulative Initialization for upgrades

Several issues arise from having both `initializer` and `reinitializer(2)` functions that perform overlapping logic when upgrading the contract (instead of deploying from scratch and calling `reinitializer`). The following discussion points address the upgrade path:

- **(a)** Both functions initialize the same storage variables. For the upgrade scenario, this might lead to inconsistencies or unnecessary state resets as crucial storage variables that should not be tampered with, could be reset.

contracts/src/operational/RollupRevenueVault.sol:L69-L71

```
) external initializer {
  __AccessControl_init();
  __RollupRevenueVault_init(
```

contracts/src/operational/RollupRevenueVault.sol:L109-L111

```
) external reinitializer(2) {
  __AccessControl_init();
  __RollupRevenueVault_init(
```

For example, calling `__RollupRevenueVault_init` again allows resetting critical state variables such as `lastInvoiceDate` or the `LINEA` token address. These values are intended to remain managed by the contract only. Allowing an upgrade to change them can create inconsistencies or break accounting logic, especially when an upgrade accidentally wraps invoicer calls due to unfortunate timing. A reinitialization routine should only adjust non-critical system parameters or perform narrowly scoped updates. Any regular configuration changes should occur through explicit setter functions with 2-step governance (e.g., time delays, multisig approvals, or controlled modules).

- **(b)** Calling `__AccessControl_init` again is redundant. While this is effectively a no-op, it still creates confusion and weakens clarity of upgrade semantics.
- **(c)** Reinvoking `__RollupRevenueVault_init` with new role assignments **adds** new administrators via `grant_role` but does **not revoke** existing ones. This can lead to privilege accumulation. If the intention is to rotate or replace admins, the logic should explicitly revoke outdated roles before granting new ones.

contracts/src/operational/RollupRevenueVault.sol:L148-L150

```
_grantRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);
_grantRole(INVOICE_SUBMITTER_ROLE, _invoiceSubmitter);
_grantRole(BURNER_ROLE, _burner);
```

Using the same initialization logic for fresh deployment and upgrades causes state reset, increasing potential for inconsistencies or misparameterization of a running system, potential for overlooking that permission grants are additive for upgrades and generally gives weaker guarantees. Ideally, the two scenarios should not reuse the same internal function - they serve different purposes and should preserve vs. reset state accordingly.

Examples

- Deployment as seen in the test cases performing an OZ `deployAndInitialize` correctly calling `initialize()` bundled to the deployment call:

contracts/test/hardhat/operational/RollupRevenueVault.ts:L68-L87

```
describe("Initialization", () => {
  it("should revert if lastInvoiceDate is zero", async () => {
    const deployCall = deployUpgradableFromFactory(
      "RollupRevenueVault",
      [
        0n,
        admin.address,
        invoiceSubmitter.address,
        burner.address,
        invoicePaymentReceiver.address,
        await tokenBridge.getAddress(),
        await messageService.getAddress(),
        l1LineaTokenBurner.address,
        await l2LineaToken.getAddress(),
        await dex.getAddress(),
      ],
      {
        initializer: ROLLUP_REVENUE_VAULT_INITIALIZE_SIGNATURE,
        unsafeAllow: ["constructor"],
      },
```

- `initializer`

contracts/src/operational/RollupRevenueVault.sol:L45-L83

```
/**
 * @notice Initializes the contract state.
 * @param _lastInvoiceDate Timestamp of the last invoice.
 * @param _defaultAdmin Address to be granted the default admin role.
 * @param _invoiceSubmitter Address to be granted the invoice submitter role.
 * @param _burner Address to be granted the burner role.
 * @param _invoicePaymentReceiver Address to receive invoice payments.
 * @param _tokenBridge Address of the token bridge contract.
 * @param _messageService Address of the L2 message service contract.
 * @param _l1LineaTokenBurner Address of the L1 LINEA token burner contract.
 * @param _lineaToken Address of the LINEA token contract.
 * @param _dex Address of the DEX contract.
 */
function initialize(
  uint256 _lastInvoiceDate,
  address _defaultAdmin,
  address _invoiceSubmitter,
  address _burner,
  address _invoicePaymentReceiver,
  address _tokenBridge,
  address _messageService,
  address _l1LineaTokenBurner,
  address _lineaToken,
  address _dex
) external initializer {
  __AccessControl_init();
  __RollupRevenueVault_init(
    _lastInvoiceDate,
    _defaultAdmin,
    _invoiceSubmitter,
    _burner,
    _invoicePaymentReceiver,
    _tokenBridge,
    _messageService,
    _l1LineaTokenBurner,
    _lineaToken,
    _dex
  );
}
```

- `reinitializer(2)`

contracts/src/operational/RollupRevenueVault.sol:L85-L123

```
/**
 * @notice Reinitializes the contract state for upgrade.
 * @param _lastInvoiceDate Timestamp of the last invoice.
 * @param _defaultAdmin Address to be granted the default admin role.
 * @param _invoiceSubmitter Address to be granted the invoice submitter role.
 * @param _burner Address to be granted the burner role.
 * @param _invoicePaymentReceiver Address to receive invoice payments.
 * @param _tokenBridge Address of the token bridge contract.
 * @param _messageService Address of the L2 message service contract.
 * @param _l1LineaTokenBurner Address of the L1 LINEA token burner contract.
 * @param _lineaToken Address of the LINEA token contract.
 * @param _dex Address of the DEX contract.
 */
function initializeRolesAndStorageVariables(
  uint256 _lastInvoiceDate,
  address _defaultAdmin,
  address _invoiceSubmitter,
  address _burner,
  address _invoicePaymentReceiver,
  address _tokenBridge,
  address _messageService,
  address _l1LineaTokenBurner,
  address _lineaToken,
  address _dex
) external reinitializer(2) {
  __AccessControl_init();
  __RollupRevenueVault_init(
    _lastInvoiceDate,
    _defaultAdmin,
    _invoiceSubmitter,
    _burner,
    _invoicePaymentReceiver,
    _tokenBridge,
    _messageService,
    _l1LineaTokenBurner,
    _lineaToken,
    _dex
  );
}
```

contracts/src/operational/RollupRevenueVault.sol:L125-L160

```
function __RollupRevenueVault_init(
  uint256 _lastInvoiceDate,
  address _defaultAdmin,
  address _invoiceSubmitter,
  address _burner,
  address _invoicePaymentReceiver,
  address _tokenBridge,
  address _messageService,
  address _l1LineaTokenBurner,
  address _lineaToken,
  address _dex
) internal onlyInitializing {
  require(_lastInvoiceDate != 0, ZeroTimestampNotAllowed());
  require(_defaultAdmin != address(0), ZeroAddressNotAllowed());
  require(_invoiceSubmitter != address(0), ZeroAddressNotAllowed());
  require(_burner != address(0), ZeroAddressNotAllowed());
  require(_invoicePaymentReceiver != address(0), ZeroAddressNotAllowed());
  require(_tokenBridge != address(0), ZeroAddressNotAllowed());
  require(_messageService != address(0), ZeroAddressNotAllowed());
  require(_l1LineaTokenBurner != address(0), ZeroAddressNotAllowed());
  require(_lineaToken != address(0), ZeroAddressNotAllowed());
  require(_dex != address(0), ZeroAddressNotAllowed());

  _grantRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);
  _grantRole(INVOICE_SUBMITTER_ROLE, _invoiceSubmitter);
  _grantRole(BURNER_ROLE, _burner);

  lastInvoiceDate = _lastInvoiceDate;

  invoicePaymentReceiver = _invoicePaymentReceiver;
  tokenBridge = TokenBridge(_tokenBridge);
  messageService = L2MessageService(_messageService);
  l1LineaTokenBurner = _l1LineaTokenBurner;
  lineaToken = _lineaToken;
  dex = _dex;
}
```

Recommendation

- Consider **removing the `initialize()` function** from future contract versions to reduce confusion and prevent misuse.
- Maintain clear, versioned contracts (`RollupRevenueVaultV1` , `RollupRevenueVaultV2` , etc.) to isolate initialization logic per version and allow incremental upgrade initializations.
- Ensure all future deployments and upgrades atomically execute `upgradeToAndCall(reinitializer_function)` to avoid reinitialization race windows.
- Restrict reinitialization logic for upgrades to **non-critical variables** only. Critical configuration values should be immutable or changeable only through explicitly controlled governance mechanisms.
- When role or permission updates are necessary, explicitly revoke outdated roles before granting new ones to avoid unintended privilege escalation.
- see issue 4.1 , fix the `initialize` call and remove the fallback function or require `msg.value > 0`
- Consider renaming the `reinitializer` function as an `initializeVx...` to make it obvious, that this is an initialization funciton.

4.3 RollupRevenueVault - Missing Validation for Future `lastInvoiceDate` Minor ✓ Fixed

Resolution

Fixed in commit bd2f0b28f59601d40d4ab63243632ccc16d9cf8a. The Linea team added `require(_lastInvoiceDate < block.timestamp, TimestampInTheFutureNotAllowed());` on reinitialization, invoice submission and when updating invoice arrears.

Description

The contract does not enforce validation to prevent `lastInvoiceDate` from being set to a future timestamp. This can permanently block invoice submissions, affecting normal invoice operations, potentially resulting in a denial of service until manual intervention occurs.

Examples

Several functions allow `lastInvoiceDate` to be set to future values without proper checks:

Invoice Submissions:

contracts/src/operational/RollupRevenueVault.sol:L162-L184

```
/**
 * @notice Submit invoice to pay to the designated receiver.
 * @param _startTimestamp Start of the period the invoice is covering.
 * @param _endTimestamp End of the period the invoice is covering.
 * @param _invoiceAmount New invoice amount.
 */
function submitInvoice(
  uint256 _startTimestamp,
  uint256 _endTimestamp,
  uint256 _invoiceAmount
) external payable onlyRole(INVOICE_SUBMITTER_ROLE) {
  require(_startTimestamp == lastInvoiceDate + 1, TimestampsNotInSequence());
  require(_endTimestamp > _startTimestamp, EndTimestampMustBeGreaterThanStartTimestamp());
  require(_invoiceAmount != 0, ZeroInvoiceAmount());

  address payable receiver = payable(invoicePaymentReceiver);
  uint256 balanceAvailable = address(this).balance;

  uint256 totalAmountOwing = invoiceArrears + _invoiceAmount;
  uint256 amountToPay = (balanceAvailable < totalAmountOwing) ? balanceAvailable : totalAmountOwing;

  invoiceArrears = totalAmountOwing - amountToPay;
  lastInvoiceDate = _endTimestamp;
```

If `lastInvoiceDate` is set to a future timestamp (e.g., year 2030), subsequent invoice submissions must have `_startTimestamp = lastInvoiceDate + 1`, effectively soft-locking the system until that future date is reached or forcing submissions to shift in the future. Setting this to `uint_MAX` will permanently prevent submissions until an admin resolves it.

Initialization:

contracts/src/operational/RollupRevenueVault.sol:L125-L152

```
function __RollupRevenueVault_init(
  uint256 _lastInvoiceDate,
  address _defaultAdmin,
  address _invoiceSubmitter,
  address _burner,
  address _invoicePaymentReceiver,
  address _tokenBridge,
  address _messageService,
  address _l1LineaTokenBurner,
  address _lineaToken,
  address _dex
) internal onlyInitializing {
  require(_lastInvoiceDate != 0, ZeroTimestampNotAllowed());
  require(_defaultAdmin != address(0), ZeroAddressNotAllowed());
  require(_invoiceSubmitter != address(0), ZeroAddressNotAllowed());
  require(_burner != address(0), ZeroAddressNotAllowed());
  require(_invoicePaymentReceiver != address(0), ZeroAddressNotAllowed());
  require(_tokenBridge != address(0), ZeroAddressNotAllowed());
  require(_messageService != address(0), ZeroAddressNotAllowed());
  require(_l1LineaTokenBurner != address(0), ZeroAddressNotAllowed());
  require(_lineaToken != address(0), ZeroAddressNotAllowed());
  require(_dex != address(0), ZeroAddressNotAllowed());

  _grantRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);
  _grantRole(INVOICE_SUBMITTER_ROLE, _invoiceSubmitter);
  _grantRole(BURNER_ROLE, _burner);

  lastInvoiceDate = _lastInvoiceDate;
```

Admin Updates:

contracts/src/operational/RollupRevenueVault.sol:L238-L252

```
/**
 * @notice Update the invoice arrears.
 * @param _newInvoiceArrears New invoice arrears value.
 * @param _lastInvoiceDate Timestamp of the last invoice.
 */
function updateInvoiceArrears(
  uint256 _newInvoiceArrears,
  uint256 _lastInvoiceDate
) external onlyRole(DEFAULT_ADMIN_ROLE) {
  require(_lastInvoiceDate >= lastInvoiceDate, InvoiceDateTooOld());

  invoiceArrears = _newInvoiceArrears;
  lastInvoiceDate = _lastInvoiceDate;
  emit InvoiceArrearsUpdated(_newInvoiceArrears, _lastInvoiceDate);
}
```

Setter: Note - This allows the admin to skip settling existing debt.

contracts/src/operational/RollupRevenueVault.sol:L238-L252

```
/**
 * @notice Update the invoice arrears.
 * @param _newInvoiceArrears New invoice arrears value.
 * @param _lastInvoiceDate Timestamp of the last invoice.
 */
function updateInvoiceArrears(
  uint256 _newInvoiceArrears,
  uint256 _lastInvoiceDate
) external onlyRole(DEFAULT_ADMIN_ROLE) {
  require(_lastInvoiceDate >= lastInvoiceDate, InvoiceDateTooOld());

  invoiceArrears = _newInvoiceArrears;
  lastInvoiceDate = _lastInvoiceDate;
  emit InvoiceArrearsUpdated(_newInvoiceArrears, _lastInvoiceDate);
}
```

Recommendation

Implement timestamp validation in all relevant functions to prevent future dates.

4.4 RollupRevenueVault - Consider Enforcement of Minimum Swap Outputs ✓ Fixed

Resolution

Fixed in commit ac30ec35287763697acb94f72fa3c630bee0b46a. Differential balance check added post swap in RollupRevenueVault.sol

Description

The `burnAndBridge` function performs token swaps via a low-level call rather than through a dedicated adapter interface such as `IV3DexSwap`. This design choice is intentional, allowing flexibility to switch or upgrade adapters (e.g., to a future V4 adapter) without tightly coupling the logic to a specific implementation.

While this is not a direct security issue, it places the responsibility for enforcing correct parameterization, such as slippage limits and output validation, entirely on the caller or the external job that initiates the `burnAndBridge` operation.

For example, the current `V3DexSwap` adapter trusts the configured router to correctly enforce the `minLineaOut` parameter, as it primarily acts as a passthrough. The `burnAndBridge` function then uses whatever token amount is returned from the swap, without performing any sanity checks on the received amount.

This design means that correctness and safety depend heavily on external configuration and on the adapter implementation, rather than being verified within `burnAndBridge` itself. While the configured router may currently be immutable and trustworthy, future adapters or router changes could increase operational risk if output validation or slippage enforcement are not consistently implemented.

Recommendation

- Confirm that the intended design is to retain the low-level swap call for adapter flexibility (as acknowledged by the client).
- If the swap adapter or router configuration is expected to remain under full project control, the current approach is acceptable. Make sure to review the dex adapter and the swapping code for sllippage control.
- To further reduce operational risk and dependency on adapter correctness, consider implementing a sanity check on minimum tokens out within `burnAndBridge` or within the controlled adapter (e.g., `V3DexSwap`). This would provide a secondary safeguard and reduce the likelihood of misconfiguration or inconsistent slippage handling in future adapter versions.

5 Document Change Log

Version	Date	Description
1.0	2025-10-20	Initial report
1.1	2025-10-24	Update: Fix Review
1.2	2025-11-02	Update: Added Deployment Addresses

Appendix 1 - Files in Scope

This review covered the following files:

File	SHA-1 hash
contracts/src/operational/L1LineaTokenBurner.sol	`13ba3c7a17b3dd165ddc44c5ea0518b585a4b9a9`
contracts/src/operational/RollupRevenueVault.sol	`73d186a487f0ed4e89102dac999a185c5f427297`
contracts/src/operational/V3DexSwap.sol	`64a492bdfba3398d83c0210ad420d7c2019b035b`
contracts/src/operational/interfaces/IL1LineaToken.sol	`7a6a9498462ca2089f5eb108696bc479db0a5200`
contracts/src/operational/interfaces/IL1LineaTokenBurner.sol	`82ac876952128efa5eecc780672b60306103a9aa`
contracts/src/operational/interfaces/ILineaSequencerUptimeFeed.sol	`d8080409e50b10b069babf68c9e6badd1de259df`
contracts/src/operational/interfaces/IRollupRevenueVault.sol	`b22ab1656d3fa0967ef653f2ca902227e476f6ae`
contracts/src/operational/interfaces/ISwapRouterV3.sol	`ffe6128110c7e64173ff4a03d97bb004a614a823`

File	SHA-1 hash
contracts/src/operational/interfaces/IV3DexSwap.sol	`e606937635ddc5875fb121acd9018e147e134a39`
contracts/src/operational/interfaces/IWETH9.sol	`866101d0f2d977bf909c1b6e35ab51e92bbe184a`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.