Linea Audit Report

Prepared by Cyfrin
Version 2.0

Lead Auditor
Dacian

May 24, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Linea is a Type 2 zkEVM L2 rollup with full EVM equivalence that aims to provide the "Ethereum experience" at scale.

5 Audit Scope

The audit scope was limited to Solidity smart contracts primarily covering the messaging service, token bridge and rollup, aiming to audit the following new developments:

- gas optimizations to blob submission and finalization for existing implementation of EIP-4844
- deprecation of previously used messaging fields and storage slots
- use of transient storage for L1 messaging and claiming of messages

All code relating to zk provers and other code outside the smart contract layer was out of scope. The following contracts were included in the scope for this audit:

```
contracts/LineaRollup.sol
contracts/ZkEvmV2.sol
contracts/interfaces/IGenericErrors.sol
contracts/interfaces/IMessageService.sol
contracts/interfaces/IPauseManager.sol
contracts/interfaces/IRateLimiter.sol
contracts/interfaces/l1/IL1MessageManager.sol
contracts/interfaces/l1/IL1MessageManagerV1.sol
contracts/interfaces/l1/IL1MessageService.sol
contracts/interfaces/l1/ILineaRollup.sol
contracts/interfaces/l1/IZkEvmV2.sol
contracts/interfaces/l2/IL2MessageManager.sol
contracts/interfaces/l2/IL2MessageManagerV1.sol
contracts/lib/Utils.sol
```

```
contracts/messageService/MessageServiceBase.sol
contracts/messageService/l1/L1MessageManager.sol
contracts/messageService/l1/L1MessageService.sol
contracts/messageService/l1/TransientStorageReentrancyGuardUpgradeable.sol
contracts/messageService/l1/v1/L1MessageManagerV1.sol
contracts/messageService/l1/v1/L1MessageServiceV1.sol
contracts/messageService/l2/L2MessageManager.sol
contracts/messageService/l2/L2MessageService.sol
contracts/messageService/l2/v1/L2MessageManagerV1.sol
contracts/messageService/l2/v1/L2MessageServiceV1.sol
contracts/messageService/lib/PauseManager.sol
contracts/messageService/lib/RateLimiter.sol
contracts/messageService/lib/SparseMerkleTreeVerifier.sol
contracts/messageService/lib/TimeLock.sol
contracts/messageService/lib/TransientStorageHelpers.sol
contracts/tokenBridge/BridgedToken.sol
contracts/tokenBridge/TokenBridge.sol
contracts/tokenBridge/interfaces/ITokenBridge.sol
```

6 Executive Summary

Over the course of 15 days, the Cyfrin team conducted an audit on the Linea smart contracts provided by Consensys / Linea. In this period, a total of 27 issues were found.

The findings consist of 2 High, 2 Medium and 4 Low severity issues with the remainder being informational and gas optimizations.

The first High was related to inconsistent usage of transient storage combined with the continued use of a deprecated storage slot which could result in token bridging reverting on the destination chain preventing some users from receiving bridged tokens.

The second High was not related to the new functionality but concerns the token bridge inadvertently allowing 1-way bridging of ERC721 tokens when it should only allow bridging of ERC20 tokens.

Common issues with first-generation L2 zkEVMs made up the 2 Mediums which related to downgrading from Ethereum's censorship resistance and the inability to cancel/refund transactions which continually revert on the destination chain.

A variety of issues composed the 4 Lows of which the most interesting was that an invariant-breaking state could be reached where multiple L2 `BridgeToken` are associated with the same L1 `NativeToken`; this issue was not related to the new changes and was previously unknown having escaped detection in all prior audits.

At the conclusion of the audit all verified mitigations were merged into the linea-contracts repository in commit b17e7c7.

Summary

Project Name	Linea
Repository	zkevm-monorepo
Commit	364b64d2a270...
Audit Timeline	April 29th - May 17th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	2
Medium Risk	2
Low Risk	4
Informational	6
Gas Optimizations	13
Total Issues	27

Summary of Findings

[H-1] Deprecated variable `L1MessageServiceV1::__messageSender` is still used making `TokenBridge::completeBridging` revert for older messages preventing users from receiving bridged tokens	Resolved
[H-2] `TokenBridge::bridgeToken` allows 1-way `ERC721` bridging causing users to permanently lose their nfts	Resolved
[M-1] Linea can permanently lock user ETH on L2 by censoring user transactions through the centralized sequencer	Acknowledged
[M-2] Users can't cancel or refund their source chain transaction if the destination chain transaction continually fails, causing loss of bridged funds	Acknowledged
[L-1] Multiple L2 `BridgeToken` can be associated with the same L1 `NativeToken` by using `TokenBridge::setCustomContract`	Resolved
[L-2] Linea can bypass validity proof verification for L2->L1 block finalization	Acknowledged
[L-3] Associated `finalBlockNumber` for an existing shnarf can be overwritten when submitting data using `LineaRollup::submitDataAsCalldata`	Resolved
[L-4] Potential L1->L2 `ERC20` decimal inconsistency can result in loss of user bridged tokens	Resolved
[I-1] Add sanity check to revert if `messageHashesLength == 0` in `L2MessageManager::anchorL1L2MessageHashes`	Acknowledged
[I-2] Add sanity check to revert if `_treeDepth == 0` in `L1MessageManager::_addL2MerkleRoots`	Resolved
[I-3] `SparseMerkleTreeVerifier` should use `Utils::_efficientKeccak` instead of duplicating the same function	Acknowledged
[I-4] Emit event in `TokenBridge::removeReserved` when updating storage	Resolved
[I-5] Emit event in `MessageServiceBase::_setRemoteSender` when updating `remoteSender` storage	Acknowledged
[I-6] Add sanity check to `LineaRollup::submitBlobs` to ensure all blobs are validated	Acknowledged
[G-01] `TokenBridge::deployBridgedToken` can use named return variable to avoid additional local variable	Acknowledged

[G-02] Reverse order of checks in `TokenBridge::isNewToken` to save 2 storage reads	Resolved
[G-03] Optimize away `bridgedMappingValue` variable in `TokenBridge::bridgeToken`	Resolved
[G-04] Save 2 storage reads in `TokenBridge::bridgeToken` by caching `sourceChainId`	Resolved
[G-05] Remove redundant modifiers from `TokenBridge::bridgeTokenWithPermit`	Resolved
[G-06] Optimize away `nativeMappingValue` variable in `TokenBridge::completeBridging`	Acknowledged
[G-07] Optimize away `nativeToken` variable in `TokenBridge::setDeployed`	Resolved
[G-08] Save 1 storage read in `TokenBridge::removeReserved` by caching `sourceChainId`	Resolved
[G-09] Save 1 storage read in `L2MessageServiceV1::setMinimumFee` by emitting `MinimumFeeChanged` event using `_feeInWei` parameter	Acknowledged
[G-10] Optimize away `currentPeriodAmountTemp` variable in `RateLimiter::_addUsedAmount`	Acknowledged
[G-11] Optimize away `usedLimitAmountToSet`, `amountUsedLoweredToLimit` and `usedAmountResetToZero` variables in `RateLimiter::resetRateLimitAmount`	Acknowledged
[G-12] Remove deprecated `SubmissionDataV2::dataParentHash` and `SupportingSubmissionDataV2::dataParentHash` from `ILineaRollup.sol`	Resolved
[G-13] Remove deprecated check from `LineaRollup::submitDataAsCalldata`	Resolved

7 Findings

7.1 High Risk

7.1.1 Deprecated variable `L1MessageServiceV1::__messageSender` is still used making `TokenBridge::completeBridging` revert for older messages preventing users from receiving bridged tokens

Description: In `L1MessageServiceV1`, this comment indicates that `_messageSender` storage variable is deprecated in favor of transient storage.

However the deprecated `_messageSender` is updated in `L1MessageServiceV1::claimMessage` and also set in `L1MessageService::__MessageService_init`. But `L1MessageService::claimMessageWithProof` does use the transient storage method.

`MessageServiceBase::onlyAuthorizedRemoteSender` calls `L1MessagingService::sender` to retrieve the message sender and this function always fetches it from transient storage and not from the deprecated `_messageSender`.

`MessageServiceBase::onlyAuthorizedRemoteSender` in turn is used by `TokenBridge::`setDeployed and `completeBridging`.

Impact: Older L2->L1 messages using `claimMessage` to call `TokenBridge::setDeployed` and `completeBridging` will always revert since:

- the message sender will be fetched from transient storage
- `claimMessage` never sets message sender in transient storage as it still updates the deprecated `_messageSender` storage variable

As `completeBridging` will always revert for these transactions, users will never receive their bridged tokens.

Recommended Mitigation: Change `L1MessageServiceV1::claimMessage` to store message sender in transient storage and remove all uses of deprecated `_messageSender`.

Linea: Fixed in commit 55d936a merged in PR3191.

Cyfrin: Verified.

7.1.2 `TokenBridge::bridgeToken` allows 1-way `ERC721` bridging causing users to permanently lose their nfts

Description: `TokenBridge::bridgeToken` is only intended to support `ERC20` tokens however it quite happily accepts `ERC721` tokens and is able to successfully bridge them over to the L2. But when users attempt to bridge back to the L1 this always reverts resulting in user nfts being permanently stuck inside the `TokenBridge` contract.

This was not introduced in the latest changes but is present in the current mainnet `TokenBridge` code.

Proof of Concept: Add new file `contracts/tokenBridge/mocks/MockERC721.sol`:

```solidity
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.0;

import { ERC721 } from "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract MockERC721 is ERC721 {
  constructor(string memory _tokenName, string memory _tokenSymbol) ERC721(_tokenName, _tokenSymbol) {}

  function mint(address _account, uint256 _tokenId) public returns (bool) {
    _mint(_account, _tokenId);
    return true;
  }
}
```

Add this new test to `test/tokenBridge/E2E.ts` under the `Bridging` section:

```
it("Can erroneously bridge ERC721 and can't bridge it back", async function () {
  const {
    user,
    l1TokenBridge,
    l2TokenBridge,
    tokens: { L1DAI },
    chainIds,
  } = await loadFixture(deployContractsFixture);

  // deploy ERC721 contract
  const ERC721 = await ethers.getContractFactory("MockERC721");
  let mockERC721 = await ERC721.deploy("MOCKERC721", "M721");
  await mockERC721.waitForDeployment();

  // mint user some NFTs
  const bridgedNft = 5;
  await mockERC721.mint(user.address, 1);
  await mockERC721.mint(user.address, 2);
  await mockERC721.mint(user.address, 3);
  await mockERC721.mint(user.address, 4);
  await mockERC721.mint(user.address, bridgedNft);

  const mockERC721Address = await mockERC721.getAddress();
  const l1TokenBridgeAddress = await l1TokenBridge.getAddress();
  const l2TokenBridgeAddress = await l2TokenBridge.getAddress();

  // user approves L1 token bridge to spend nft to be bridged
  await mockERC721.connect(user).approve(l1TokenBridgeAddress, bridgedNft);

  // user bridges their nft
  await l1TokenBridge.connect(user).bridgeToken(mockERC721Address, bridgedNft, user.address);

  // verify user has lost their nft which is now owned by L1 token bridge
  expect((await mockERC721.ownerOf(bridgedNft))).to.be.equal(l1TokenBridgeAddress);

  // verify user has received 1 token on the bridged erc20 version which
  // the token bridge created
  const l2TokenAddress = await l2TokenBridge.nativeToBridgedToken(chainIds[0], mockERC721Address);
  const bridgedToken = await ethers.getContractFactory("BridgedToken");
  const l2Token = bridgedToken.attach(l2TokenAddress) as BridgedToken;

  const l2ReceivedTokenAmount = 1;

  expect(await l2Token.balanceOf(user.address)).to.be.equal(l2ReceivedTokenAmount);
  expect(await l2Token.totalSupply()).to.be.equal(l2ReceivedTokenAmount);

  // now user attempts to bridge back which fails
  await l2Token.connect(user).approve(l2TokenBridgeAddress, l2ReceivedTokenAmount);
  await expectRevertWithReason(
    l2TokenBridge.connect(user).bridgeToken(l2TokenAddress, l2ReceivedTokenAmount, user.address),
    "SafeERC20: low-level call failed",
  );
});
```

Run the test with: `npx hardhat test --grep "Can erroneously bridge ERC721"`

Recommended Mitigation: `TokenBridge` should not allow users to bridge `ERC721` tokens. The reason it currently does is because the `ERC721` interface is very similar to the `ERC20` interface. One option for preventing this is to change `TokenBridge::_safeDecimals` to revert if the call fails as:

- `ERC20` tokens should always provide their decimals
- `ERC721` tokens don't have decimals

Another option would be using `ERC165` interface detection but not all nfts support this.

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Linea can permanently lock user ETH on L2 by censoring user transactions through the centralized sequencer

Description: Linea can permanently lock user ETH on L2 by censoring user transactions through the centralized sequencer; as the Linea team control the sequencer they can choose (or be forced by a government entity) to censor any users by not including their L2 transactions when building L2 blocks.

Additionally, there is no way for users to withdraw their L2 assets by initiating an L1 transaction which would avoid the L2 censorship.

The combination of these factors means that users bridging assets over to Linea are completely at the mercy of the Linea team for their continued freedom to transact.

This represents a significant downgrade from Ethereum's censorship resistance where even the OFAC censorship regime only had 38% enforcement over the last 30 days.

It should be noted however that this is a common defect [1, 2] with first-generation zkEVMs and de-centralized zk L2 sequencers are an active area of research [1, 2], so Linea is not an exceptional case in this regard.

Recommended Mitigation: Migrate to a decentralized sequencer and/or implement a mechanism that would allow users to retrieve their ETH (and ideally also bridged tokens) purely by initiating an L1 transaction completely bypassing L2 censorship.

Linea: Acknowledged; this will be addressed when we decentralize.

7.2.2 Users can't cancel or refund their source chain transaction if the destination chain transaction continually fails, causing loss of bridged funds

Description: Linea's messaging service allows users to bridge ETH between Ethereum L1 mainnet and Linea L2.

The bridging process operates in 2 distinct transactions where the first can succeed and the second can fail completely independently of each other.

In the first step of the bridging process on the source chain the user supplies their ETH to the messaging service contract on that chain together with some parameters.

In the last step of the bridging process on the destination chain a delivery service ("postman") or the user themselves calls `claimMessage` or `claimMessageWithProof` to complete the bridging process.

The last step of claiming the message is an arbitrary call with arbitrary calldata made to an address of the user's choosing. If this call fails for any reason the user may re-try claiming the message an infinite number of times. However if this call continues to revert (or if the `claim` transaction continually reverts for any other reason), the user:

- will never receive their bridged ETH on the destination chain
- has no way of initiating a cancel or refund transaction on the source chain

Impact: This scenario will result in a loss of bridged funds to the user.

Recommended Mitigation: Implement a cancel or refund mechanism that users can initiate on the source chain to retrieve their funds if the claim transaction continually reverts on the destination chain.

Linea: Acknowledged. Currently there is no cancellation or refund feature but users can re-try the destination chain delivery as many times as they like. We may consider implementing this in the future.

7.3 Low Risk

7.3.1 Multiple L2 `BridgeToken` can be associated with the same L1 `NativeToken` by using `TokenBridge::setCustomContract`

Description: Multiple L2 `BridgeToken` can be associated with the same L1 `NativeToken` by using `TokenBridge::setCustomContract`.

This was not introduced in the latest changes but is present in the current mainnet `TokenBridge` code.

Impact: This breaks the `TokenBridge` invariant that one L1 `NativeToken` should only be associated with one L2 `BridgeToken`.

Proof of Concept: Add this PoC to `test/tokenBridge/E2E.ts` under the `setCustomsContract` section:

```
  it("Multiple L2 BridgeTokens can be associated with a single L1 NativeToken by using
  ↪  setCustomContract", async function () {
    const {
      user,
      l1TokenBridge,
      l2TokenBridge,
      tokens: { L1USDT },
    } = await loadFixture(deployContractsFixture);

    // @audit the setup of this PoC has been copied from the test
    // "Should mint and burn correctly from a target contract"
    const amountBridged = 100;

    // Deploy custom contract
    const CustomContract = await ethers.getContractFactory("MockERC20MintBurn");
    const customContract = await CustomContract.deploy("CustomContract", "CC");

    // Set custom contract
    await l2TokenBridge.setCustomContract(await L1USDT.getAddress(), await
    ↪  customContract.getAddress());

    // @audit save user balance before both bridging attempts
    const USDTBalanceBefore = await L1USDT.balanceOf(user.address);

    // Bridge token (allowance has been set in the fixture)
    await l1TokenBridge.connect(user).bridgeToken(await L1USDT.getAddress(), amountBridged,
    ↪  user.address);

    expect(await L1USDT.balanceOf(await l1TokenBridge.getAddress())).to.be.equal(amountBridged);
    expect(await customContract.balanceOf(user.address)).to.be.equal(amountBridged);
    expect(await customContract.totalSupply()).to.be.equal(amountBridged);

    // @audit deploy a new second custom contract
    const customContract2 = await CustomContract.deploy("CustomContract2", "CC2");

    // @audit using setCustomContract, this second L2 BridgeToken can be associated
    // with L1USDT Token even though it should be impossible to do this as L1USDT has
    // already been bridged and already has a paired L2 BridgeToken
    //
    // @audit Once the suggested fix is applied, this should revert
    await l2TokenBridge.setCustomContract(await L1USDT.getAddress(), await
    ↪  customContract2.getAddress());

    // @audit same user now bridges L1->L2 again
    await l1TokenBridge.connect(user).bridgeToken(await L1USDT.getAddress(), amountBridged,
    ↪  user.address);

    // @audit L1 TokenBridge has twice the balance
    expect(await L1USDT.balanceOf(await l1TokenBridge.getAddress())).to.be.equal(amountBridged*2);

    // @audit first L2 BridgeToken still has the same balance and supply
    expect(await customContract.balanceOf(user.address)).to.be.equal(amountBridged);
    expect(await customContract.totalSupply()).to.be.equal(amountBridged);

    // @audit second L2 BridgeToken received new balance and supply from second bridging
    expect(await customContract2.balanceOf(user.address)).to.be.equal(amountBridged);
    expect(await customContract2.totalSupply()).to.be.equal(amountBridged);

    // @audit user L1 USDT balance deducted due to both bridging attempts
    const USDTBalanceAfterTwoL1ToL2Bridges = await L1USDT.balanceOf(user.address);
    expect(USDTBalanceBefore - USDTBalanceAfterTwoL1ToL2Bridges).to.be.equal(amountBridged*2);

    // @audit user now bridges back from first L2 BridgeToken
    await l2TokenBridge.connect(user).bridgeToken(await customContract.getAddress(), amountBridged,
    ↪  user.address);

    // @audit first L2 BridgeToken has 0 balance and supply
    expect(await customContract.balanceOf(user.address)).to.be.equal(0);
    expect(await customContract.totalSupply()).to.be.equal(0);

    // @audit user then bridges back from second L2 BridgeToken
    await l2TokenBridge.connect(user).bridgeToken(await customContract2.getAddress(), amountBridged,
    ↪  user.address);

    // @audit second L2 BridgeToken has 0 balance and supply
    expect(await customContract2.balanceOf(user.address)).to.be.equal(0);
    expect(await customContract2.totalSupply()).to.be.equal(0);

    // @audit user should have received back both bridged amounts
    const USDTBalanceAfter = await L1USDT.balanceOf(user.address);
    expect(USDTBalanceAfter - USDTBalanceAfterTwoL1ToL2Bridges).to.be.equal(amountBridged*2);

    // @audit user ends up with initial token amount
    expect(USDTBalanceAfter).to.be.equal(USDTBalanceBefore);

    // @audit user was able to bridge L1->L2 to two different L2 BridgeTokens,
    // and was able to bridge back L2->L1 from both different L2 BridgeTokens back
    // into the same L1 NativeToken. This breaks the invariant that one L1 NativeToken
    // should only be associated with one L2 BridgeToken
  });
```

Run with: `npx hardhat test --grep "Multiple L2 BridgeTokens can be associated with"`

Recommended Mitigation: `TokenBridge::setCustomContract` should prevent multiple L2 `BridgeToken` from being associated with the same L1 `NativeToken`. One possible implementation is to add this check:

```
function setCustomContract(
  address _nativeToken,
  address _targetContract
) external nonZeroAddress(_nativeToken) nonZeroAddress(_targetContract) onlyOwner
↪  isNewToken(_nativeToken) {
  if (bridgedToNativeToken[_targetContract] != EMPTY) {
    revert AlreadyBrigedToNativeTokenSet(_targetContract);
  }
  if (_targetContract == NATIVE_STATUS || _targetContract == DEPLOYED_STATUS || _targetContract ==
  ↪  RESERVED_STATUS) {
    revert StatusAddressNotAllowed(_targetContract);
  }
```

```
    // @audit proposed fix
    uint256 targetChainIdCache = targetChainId;

    if (nativeToBridgedToken[targetChainIdCache][_nativeToken] != EMPTY) {
      revert("Already set a custom contract for this native token");
    }

    nativeToBridgedToken[targetChainIdCache][_nativeToken] = _targetContract;
    bridgedToNativeToken[_targetContract] = _nativeToken;
    emit CustomContractSet(_nativeToken, _targetContract, msg.sender);
  }
```

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.3.2 Linea can bypass validity proof verification for L2->L1 block finalization

Description: One major advantage of using zk rollups over Optimistic rollups is that zk rollups use math & cryptography to verify L2->L1 block finalization via a validity proof which must be submitted and verified.

In the current codebase the Linea team has two options to bypass validity proof verification during block finalization:

1) Explicitly by calling `LineaRollup::finalizeBlocksWithoutProof` which takes no proof parameter as input and doesn't call the `_verifyProof` function.
2) More subtly by calling `LineaRollup::setVerifierAddress` to associate a `_proofType` with a `_newVerifierAddress` that implements the `IPlonkVerifier::Verify` interface, but which simply returns `true` and doesn't perform any actual verification. Then call `LineaRollup::finalizeBlocksWithProof` passing `_proofType` associated with `_newVerifierAddress` which will always return true.

Impact: Using either option makes Linea roughly equivalent to an Optimistic roll-up but without Watchers and the ability to challenge. That said if either option is used, `LineaRollup::_finalizeBlocks` always executes which enforces many "sanity checks" that significantly limit how these options can be abused.

The "without proof" functionality can also be used to add false L2 merkle roots which could then be used to call `L1MessageService::claimMessageWithProof` to drain ETH from the L1.

Recommended Mitigation: Linea is still at the alpha stage so these functions are likely needed as a last resort. Ideally once Linea is more mature such functionality would be removed.

Linea: Acknowledged; the ability to finalize blocks without proof is primarily part of our "training wheels" controlled via the Security Council, analyzed by Security partners and reserved for particular cases like we had when we upgraded our state manager to another hashing algorithm (the last time it was used).

7.3.3 Associated `finalBlockNumber` for an existing shnarf can be overwritten when submitting data using `LineaRollup::submitDataAsCalldata`

Description: When submitting data using `LineaRollup::submitDataAsCalldata`, if the `computedShnarf` for the current submission turns out to be a shnarf that has already been submitted before, the associated `finalBlockNumber` to this shnarf will be overwritten by the new `submissionData.finalBlockInData`.

The problem is caused because the existing check is comparing if `finalBlockInNumber` associated with the shnarf is the same as the new `submissionData.finalBlockInData`, instead of checking if the shnarf has already been associated with any other `finalBlockNumber`.

Impact: Historical data associated with an existing shnarf can be overwritten by the newest shnarf's submission data for blocks which have not yet been finalized.

Proof of Concept: Add the PoC to `test/LineaRollup.ts` inside the section `describe("Data submission tests", () => {`:

```
it.only("Submitting data on a range of blocks where data had already been submitted", async () => {
  let [firstSubmissionData, secondSubmissionData] = generateCallDataSubmission(0, 2);
  const firstParentShnarfData = generateParentShnarfData(0);

  // @audit Assume this is the lastFinalizedBlock, in the sense that any of the
  //        next submissions can't submit data to a block prior to this one
  // @audit The `firstBlockInData` of the next submissions must start at block 550
  firstSubmissionData.finalBlockInData = 549n;
  await expect(
    lineaRollup
      .connect(operator)
      .submitDataAsCalldata(firstParentShnarfData, firstSubmissionData, { gasLimit: 30_000_000 }),
  ).to.not.be.reverted;

  parentSubmissionData = generateParentSubmissionDataForIndex(1);
  const secondParentShnarfData = generateParentShnarfData(1);

  // @audit Submission for block range 550 - 599
  secondSubmissionData.firstBlockInData = 550n;
  secondSubmissionData.finalBlockInData = 599n;
  await expect(
    lineaRollup.connect(operator).submitDataAsCalldata(secondParentShnarfData,
    ↪  secondSubmissionData, {
      gasLimit: 30_000_000,
    }),
  ).to.not.be.reverted;

  let finalBlockNumber = await lineaRollup.shnarfFinalBlockNumbers(secondExpectedShnarf);
  expect(finalBlockNumber == 599n).to.be.true;
  console.log("finalBlockNumber: ", finalBlockNumber);

  // @audit Submission for block range 550 - 649
  //        Instead of continuing from the "lastSubmitedBlock (599)"
  let secondSubmissionOnSameInitialBlockData = Object.assign({}, secondSubmissionData);
  secondSubmissionOnSameInitialBlockData.firstBlockInData = 550n;
  secondSubmissionOnSameInitialBlockData.finalBlockInData = 649n;
  await expect(
    lineaRollup.connect(operator).submitDataAsCalldata(secondParentShnarfData,
    ↪  secondSubmissionOnSameInitialBlockData, {
      gasLimit: 30_000_000,
    }),
  ).to.not.be.reverted;

  finalBlockNumber = await lineaRollup.shnarfFinalBlockNumbers(secondExpectedShnarf);
  expect(finalBlockNumber == 649n).to.be.true;
  console.log("finalBlockNumber: ", finalBlockNumber);

  // @audit Submission for block range 550 - 575
  let thirdSubmissionOnSameInitialBlockData = Object.assign({}, secondSubmissionData);;
  thirdSubmissionOnSameInitialBlockData.firstBlockInData = 550n;
  thirdSubmissionOnSameInitialBlockData.finalBlockInData = 575n;
  await expect(
    lineaRollup.connect(operator).submitDataAsCalldata(secondParentShnarfData,
    ↪  thirdSubmissionOnSameInitialBlockData, {
      gasLimit: 30_000_000,
    }),
  ).to.not.be.reverted;

  finalBlockNumber = await lineaRollup.shnarfFinalBlockNumbers(secondExpectedShnarf);
  expect(finalBlockNumber == 575n).to.be.true;
  console.log("finalBlockNumber: ", finalBlockNumber);
```

```
    // @audit Each time a duplicated shnarf was computed the associated
    // `finalBlockNumber` was updated for the new `finalBlockNumber` of the newest submission.
  });
```

Run with: `npx hardhat test --grep "Submitting data on a range of blocks where data had"`

When the test is executed, observe on the console how the `finalBlockNumber` associated with the same shnarf is updated on each of the submissions that generates an existing shnarf which had already been submitted before.

Recommended Mitigation: Check if the `computedShnarf` for the current submission has already been associated with a previous `finalBlockInNumber`:

```
function submitDataAsCalldata(
  ...
) external whenTypeAndGeneralNotPaused(PROVING_SYSTEM_PAUSE_TYPE) onlyRole(OPERATOR_ROLE) {
  ...

- if (shnarfFinalBlockNumbers[shnarf] == _submissionData.finalBlockInData) {
-   revert DataAlreadySubmitted(shnarf);
- }

+ if (shnarfFinalBlockNumbers[shnarf] != 0) {
+   revert DataAlreadySubmitted(shnarf);
+ }

  ...
}
```

Linea: Fixed in commit 1816c80 merged in PR3191.

Cyfrin: Verified.

7.3.4 Potential L1->L2 `ERC20` decimal inconsistency can result in loss of user bridged tokens

Description: The first time a new ERC20 token is bridged, `TokenBridge::bridgeToken` uses the `IERC20MetadataUpgradeable` interface to fetch the token's name, symbol and decimals in order to create a bridged version of the token.

If the `staticcall` to `IERC20MetadataUpgradeable::decimals` fails to return a valid value then `TokenBridge::_safeDecimals` defaults to 18 decimals.

Impact: This is potentially dangerous as:

- a non-standard ERC20 token could be using a non-18 decimal value internally but not have a `decimals` function
- the newly created bridged version will have 18 decimals different to the native token

When the user attempts to bridge back, the user's bridged tokens will be burned and then on the native chain:

- if the native token has less than 18 decimals, the transfer call will revert resulting in the user's originally bridged tokens stuck in the token bridge
- if the native token has greater than 18 decimals, the transfer call will succeed resulting in the user receiving less tokens than they initially bridged over

Recommended Mitigation: `TokenBridge::_safeDecimals` should revert if the token being bridged does not return its decimals from the call to `IERC20MetadataUpgradeable::decimals`. Interestingly Consensys itself reported the same finding as issue 5.4 in their Nov 2021 Arbitrum audit.

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.4 Informational

7.4.1 Add sanity check to revert if `messageHashesLength == 0` in `L2MessageManager::anchorL1L2MessageHashes`

Description: Add sanity check to revert if `messageHashesLength == 0` in `L2MessageManager::anchorL1L2MessageHashes`.

Linea: Acknowledged; will be part of a future release.

7.4.2 Add sanity check to revert if `_treeDepth == 0` in `L1MessageManager::_addL2MerkleRoots`

Description: Add sanity check to revert if `_treeDepth == 0` in `L1MessageManager::_addL2MerkleRoots` to prevent adding an L2 Merkle root with `treeDepth == 0` as this would create a state where subsequent calls to `L1MessagingService::claimMessageWithProof` will revert with error `L2MerkleRootDoesNotExist`.

Linea: The depth is always enforced in the circuit and as it forms part of the public input it can never be set to zero - there is a fixed size in the circuit. We have added a comment to note this in commit 54c4670 merged in PR3214.

7.4.3 `SparseMerkleTreeVerifier` should use `Utils::_efficientKeccak` instead of duplicating the same function

Description: `SparseMerkleTreeVerifier` should use `Utils::_efficientKeccak` instead of duplicating the same function.

Linea: Acknowledged; will be part of a future release.

7.4.4 Emit event in `TokenBridge::removeReserved` when updating storage

Description: `TokenBridge::setReserved` emits an event so the opposite function `removeReserved` should likely also emit an event when updating storage.

Linea: Fixed in commit 796f2d9 merged in PR3249.

Cyfrin: Verified.

7.4.5 Emit event in `MessageServiceBase::_setRemoteSender` when updating `remoteSender` storage

Description: Emit event in `MessageServiceBase::_setRemoteSender` when updating `remoteSender` storage.

Linea: Acknowledged; may be included in a future release. `MessageServiceBase::_setRemoteSender` will only ever be used again on new testnet chains and will not be usable for existing Sepolia and Mainnet deploys.

7.4.6 Add sanity check to `LineaRollup::submitBlobs` to ensure all blobs are validated

Description: In `LineaRollup::submitBlobs`, if `_blobSubmissionData.length` is smaller than the actual number of attached blobs the code will successfully execute and store an intermediate value to `shnarfFinalBlockNumbers[computedShnarf]` because the `for` loop did not iterate through all the blobs.

To prevent this edge-case from occurring add an early fail sanity check before the `for` loop eg:

```
if(blobhash(blobSubmissionLength) != EMPTY_HASH) {
      revert MoreBlobsThanData();
}
```

Linea: Acknowledged; will be resolved in a future release focused on decentralization. Currently the Operator Service is written in such a way this cannot happen, and there is always a 1:1 mapping of data:blob.

7.5 Gas Optimization

7.5.1 `TokenBridge::deployBridgedToken` can use named return variable to avoid additional local variable

Description: `TokenBridge::deployBridgedToken` can use named return variable to avoid additional local variable, eg:

```
function deployBridgedToken(
  address _nativeToken,
  bytes calldata _tokenMetadata,
  uint256 _chainId
) internal returns (address bridgedTokenAddress) { // @audit named return
  bytes32 _salt = keccak256(abi.encode(_chainId, _nativeToken));
  BeaconProxy bridgedToken = new BeaconProxy{ salt: _salt }(tokenBeacon, "");
  bridgedTokenAddress = address(bridgedToken); // @audit assign to named return

  (string memory name, string memory symbol, uint8 decimals) = abi.decode(_tokenMetadata, (string,
↪  string, uint8));
  BridgedToken(bridgedTokenAddress).initialize(name, symbol, decimals);
  emit NewTokenDeployed(bridgedTokenAddress, _nativeToken);
  // @audit removed explicit return statement
}
```

Linea: Acknowledged; may be part of a future release.

7.5.2 Reverse order of checks in `TokenBridge::isNewToken` to save 2 storage reads

Description: `TokenBridge::isNewToken` can often save 2 storage reads by reversing the order of checks, eg:

```
modifier isNewToken(address _token) {
  // @audit swapped order of checks to save 2 storage reads by often failing on first check
  if (bridgedToNativeToken[_token] != EMPTY || nativeToBridgedToken[sourceChainId][_token] != EMPTY)
    revert AlreadyBridgedToken(_token);
  _;
}
```

This way the modifier will almost always fail on the first check having done only 1 storage read, not triggering the 2 storage reads that occur in the second check.

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.5.3 Optimize away `bridgedMappingValue` variable in `TokenBridge::bridgeToken`

Description: Optimize away `bridgedMappingValue` variable in `TokenBridge::bridgeToken` by for example:

```
function bridgeToken(
  address _token,
  uint256 _amount,
  address _recipient
) public payable nonZeroAddress(_token) nonZeroAddress(_recipient) nonZeroAmount(_amount)
↪  whenNotPaused nonReentrant {
  address nativeMappingValue = nativeToBridgedToken[sourceChainId][_token];

  if (nativeMappingValue == RESERVED_STATUS) {
    // Token is reserved
    revert ReservedToken(_token);
  }
```

```
    // @audit remove `bridgedMappingValue` and
    // instead initialize `nativeToken`
    address nativeToken = bridgedToNativeToken[_token];
    uint256 chainId;
    bytes memory tokenMetadata;

    // @audit use `nativeToken` in the comparison
    if (nativeToken != EMPTY) {
      // Token is bridged
      BridgedToken(_token).burn(msg.sender, _amount);
      // @audit remove assignment `nativeToken = bridgedMappingValue`
      chainId = targetChainId;
    } else {
   // @audit remaining code continues unchanged
```

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.5.4 Save 2 storage reads in `TokenBridge::bridgeToken` by caching `sourceChainId`

Description: `sourceChainId` is always read once at the beginning of `TokenBridge::bridgeToken` L153.

Then in the `else` branch of `TokenBridge::bridgeToken`, `sourceChainId` is always read at least once L191 but can also be read a second time L183.

Hence it is more efficient to cache `sourceChainId` at the beginning of `TokenBridge::bridgeToken` to save 2 storage reads. The optimized version of `bridgeToken` found below also contains the optimization from the previous issue *"Optimize away bridgedMappingValue variable in TokenBridge::bridgeToken"* which is required to get around the "stack too deep" error this optimization would otherwise create:

```
  function bridgeToken(
    address _token,
    uint256 _amount,
    address _recipient
  ) public payable nonZeroAddress(_token) nonZeroAddress(_recipient) nonZeroAmount(_amount)
  ↪  whenNotPaused nonReentrant {
    // @audit cache `sourceChainId` and use it when setting `nativeMappingValue`
    uint256 sourceChainIdCache = sourceChainId;
    address nativeMappingValue = nativeToBridgedToken[sourceChainIdCache][_token];

    if (nativeMappingValue == RESERVED_STATUS) {
      // Token is reserved
      revert ReservedToken(_token);
    }

    // @audit remove `bridgedMappingValue` and
    // instead initialize `nativeToken`
    address nativeToken = bridgedToNativeToken[_token];
    uint256 chainId;
    bytes memory tokenMetadata;

    // @audit use `nativeToken` in the comparison
    if (nativeToken != EMPTY) {
      // Token is bridged
      BridgedToken(_token).burn(msg.sender, _amount);
      // @audit remove assignment `nativeToken = bridgedMappingValue`
      chainId = targetChainId;
    } else {
      // Token is native
      // For tokens with special fee logic, ensure that only the amount received
      // by the bridge will be minted on the target chain.
      uint256 balanceBefore = IERC20Upgradeable(_token).balanceOf(address(this));
      IERC20Upgradeable(_token).safeTransferFrom(msg.sender, address(this), _amount);
      _amount = IERC20Upgradeable(_token).balanceOf(address(this)) - balanceBefore;

      nativeToken = _token;

      if (nativeMappingValue == EMPTY) {
        // New token
        // @audit using cached `sourceChainIdCache`
        nativeToBridgedToken[sourceChainIdCache][_token] = NATIVE_STATUS;
        emit NewToken(_token);
      }

      // Send Metadata only when the token has not been deployed on the other chain yet
      if (nativeMappingValue != DEPLOYED_STATUS) {
        tokenMetadata = abi.encode(_safeName(_token), _safeSymbol(_token), _safeDecimals(_token));
      }
      // @audit using cached `sourceChainIdCache`
      chainId = sourceChainIdCache;
    }

    messageService.sendMessage{ value: msg.value }(
      remoteSender,
      msg.value, // fees
      abi.encodeCall(ITokenBridge.completeBridging, (nativeToken, _amount, _recipient, chainId,
      ↪  tokenMetadata))
    );
    emit BridgingInitiated(msg.sender, _recipient, _token, _amount);
  }
```

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.5.5 Remove redundant modifiers from `TokenBridge::bridgeTokenWithPermit`

Description: Since `TokenBridge::bridgeTokenWithPermit` calls `bridgeToken`, it doesn't need to repeat the same modifiers that `bridgeToken` contains; this just incurs additional gas costs.

The only benefit to the redundant modifiers is they make the transaction revert faster but in the majority of normal cases where non-zero inputs are passed these additional redundant modifiers simply increase the gas cost of every successful bridging transaction for regular users.

Remove the `nonZeroAddress`, `nonZeroAmount` and `whenNotPaused` modifiers from `bridgeTokenWithPermit` since they are already enforced by `bridgeToken`.

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.5.6 Optimize away `nativeMappingValue` variable in `TokenBridge::completeBridging`

Description: Optimize away `nativeMappingValue` variable in `TokenBridge::completeBridging` by for example:

```
function completeBridging(
  address _nativeToken,
  uint256 _amount,
  address _recipient,
  uint256 _chainId,
  bytes calldata _tokenMetadata
) external nonReentrant onlyMessagingService onlyAuthorizedRemoteSender whenNotPaused {
  // @audit remove `nativeMappingValue` and instead
  // initialize `bridgedToken`
  address bridgedToken = nativeToBridgedToken[_chainId][_nativeToken];

  // @audit use `bridgedToken` for the check
  if (bridgedToken == NATIVE_STATUS || bridgedToken == DEPLOYED_STATUS) {
    // Token is native on the local chain
    IERC20Upgradeable(_nativeToken).safeTransfer(_recipient, _amount);
  } else {
     // @audit remove assignment `bridgedToken = nativeMappingValue;`
     //
     // @audit use `bridgedToken` for the check
    if (bridgedToken == EMPTY) {
      // New token
      bridgedToken = deployBridgedToken(_nativeToken, _tokenMetadata, sourceChainId);
      bridgedToNativeToken[bridgedToken] = _nativeToken;
      nativeToBridgedToken[targetChainId][_nativeToken] = bridgedToken;
    }
    BridgedToken(bridgedToken).mint(_recipient, _amount);
  }

  emit BridgingFinalized(
    _nativeToken,
    // @audit return 0 address as before if token was native on local chain
    bridgedToken == NATIVE_STATUS || bridgedToken == DEPLOYED_STATUS ? address(0x0) : bridgedToken,
    _amount,
    _recipient);
}
```

Linea: Acknowledged; will be part of a future release.

7.5.7 Optimize away `nativeToken` variable in `TokenBridge::setDeployed`

Description: Optimize away `nativeToken` variable in `TokenBridge::setDeployed` which is written to but never read, eg:

```
  function setDeployed(address[] calldata _nativeTokens) external onlyMessagingService
  ↪  onlyAuthorizedRemoteSender {
    // @audit remove `nativeToken`
    unchecked {
      for (uint256 i; i < _nativeTokens.length; ) {
        // @audit remove assignment nativeToken = _nativeTokens[i]
        nativeToBridgedToken[sourceChainId][_nativeTokens[i]] = DEPLOYED_STATUS;
        emit TokenDeployed(_nativeTokens[i]);
        ++i;
      }
    }
  }
```

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.5.8 Save 1 storage read in `TokenBridge::removeReserved` by caching `sourceChainId`

Description: `sourceChainId` is read from storage twice in `TokenBridge::removeReserved` so caching it can save 1 storage read, eg:

```
function removeReserved(address _token) external nonZeroAddress(_token) onlyOwner {
  // @audit cache `sourceChainId`
  uint256 sourceChainIdCache = sourceChainId;

  // @audit used cached version
  if (nativeToBridgedToken[sourceChainIdCache][_token] != RESERVED_STATUS) revert NotReserved(_token);
  nativeToBridgedToken[sourceChainIdCache][_token] = EMPTY;
}
```

Linea: Fixed in commit 35c7807 merged in PR3249.

Cyfrin: Verified.

7.5.9 Save 1 storage read in `L2MessageServiceV1::setMinimumFee` by emitting `MinimumFeeChanged` event using `_feeInWei` parameter

Description: Save 1 storage read in `L2MessageServiceV1::setMinimumFee` by emitting the `MinimumFeeChanged` event using the `_feeInWei` parameter eg:

```
function setMinimumFee(uint256 _feeInWei) external onlyRole(MINIMUM_FEE_SETTER_ROLE) {
  uint256 previousMinimumFee = minimumFeeInWei;
  minimumFeeInWei = _feeInWei;

  emit MinimumFeeChanged(previousMinimumFee, _feeInWei, msg.sender);
}
```

Linea: Acknowledged; will be part of a future release.

7.5.10 Optimize away `currentPeriodAmountTemp` variable in `RateLimiter::_addUsedAmount`

Description: Optimize away `currentPeriodAmountTemp` in `RateLimiter::_addUsedAmount` by for example:

```
function _addUsedAmount(uint256 _usedAmount) internal {
  // @audit removed `currentPeriodAmountTemp`

  if (currentPeriodEnd < block.timestamp) {
    currentPeriodEnd = block.timestamp + periodInSeconds;
    // @audit removed assignment `currentPeriodAmountTemp = _usedAmount`
  } else {
    // @audit modifying `_usedAmount` instead of `currentPeriodAmountTemp`
    _usedAmount += currentPeriodAmountInWei;
  }

  // @audit use `_usedAmount` in check
  if (_usedAmount > limitInWei) {
    revert RateLimitExceeded();
  }

  // @audit use `_usedAmount` in assignment
  currentPeriodAmountInWei = _usedAmount;
}
```

Linea: Acknowledged; may be part of a future release.

7.5.11 Optimize away `usedLimitAmountToSet`, `amountUsedLoweredToLimit` and `usedAmountResetToZero` variables in `RateLimiter::resetRateLimitAmount`

Description: Optimize away `usedLimitAmountToSet`, `amountUsedLoweredToLimit` and `usedAmountResetToZero` variables in `RateLimiter::resetRateLimitAmount` by for example:

```
function resetRateLimitAmount(uint256 _amount) external onlyRole(RATE_LIMIT_SETTER_ROLE) {
  // @audit remove `usedLimitAmountToSet`, `amountUsedLoweredToLimit` and `usedAmountResetToZero`

  // @audit update this first as this always happens
  limitInWei = _amount;

  if (currentPeriodEnd < block.timestamp) {
    currentPeriodEnd = block.timestamp + periodInSeconds;
    // @audit remove assignment to `usedAmountResetToZero` and instead directly set
    // `currentPeriodAmountInWei` to zero instead of doing it later
    currentPeriodAmountInWei = 0;

    // @audit emitting event here as boolean variables removed
    emit LimitAmountChanged(_msgSender(), _amount, false, true);
  }
  // @audit refactor `else/if` statement to simplify
  else if(_amount < currentPeriodAmountInWei) {
      // @audit remove assignment usedLimitAmountToSet = _amount
      // @audit remove assignemnt to `amountUsedLoweredToLimit` and instead directly
      // set `currentPeriodAmountInWei` to `_amount` instead of doing it later
      currentPeriodAmountInWei = _amount;

      // @audit emitting event here as boolean variables removed
      emit LimitAmountChanged(_msgSender(), _amount, true, false);
  }
  // @audit new `else` condition to emit event for when
  // neither boolean was true
  else{
    emit LimitAmountChanged(_msgSender(), _amount, false, false);
  }
}
```

Alternatively only optimize away the `usedLimitAmountToSet` variable:

```
function resetRateLimitAmount(uint256 _amount) external onlyRole(RATE_LIMIT_SETTER_ROLE) {
  // @audit remove `usedLimitAmountToSet`
  bool amountUsedLoweredToLimit;
  bool usedAmountResetToZero;

  if (currentPeriodEnd < block.timestamp) {
    currentPeriodEnd = block.timestamp + periodInSeconds;
    usedAmountResetToZero = true;
  } else {
    if (_amount < currentPeriodAmountInWei) {
      // @audit remove assignment usedLimitAmountToSet = _amount
      amountUsedLoweredToLimit = true;
    }
  }

  limitInWei = _amount;

  // @audit refactor if statement to handle zero case
  if(usedAmountResetToZero) currentPeriodAmountInWei = 0;
  // @audit use `_amount` for assignment instead of `usedLimitAmountToSet`
  else if(amountUsedLoweredToLimit) currentPeriodAmountInWei = _amount;
  emit LimitAmountChanged(_msgSender(), _amount, amountUsedLoweredToLimit, usedAmountResetToZero);
}
```

Linea: Acknowledged; will be part of a future release.

7.5.12 Remove deprecated `SubmissionDataV2::dataParentHash` and `SupportingSubmissionDataV2::dataParentHash` from `ILineaRollup.sol`

Description: Remove deprecated `SubmissionDataV2::dataParentHash` and `SupportingSubmissionDataV2::dataParentHash` from `ILineaRollup.sol`.

These are hangovers from the previous implementation and should be removed; in the current `LineaRollup.sol` `dataParentHash` is only copied from `calldata` into `memory` in `submitDataAsCalldata` but is not actually used for anything.

Linea: Fixed in commit 2261c78 merged in PR3191.

Cyfrin: Verified.

7.5.13 Remove deprecated check from `LineaRollup::submitDataAsCalldata`

Description: In commit 9840265eeb7d8ed9a4a5107b7fad056093b224e0 the check which reverted with error `WrongParentShnarfFinalBlockNumber` was removed from `submitBlobs`, and the test which caught that error was also updated to catch a different error `DataStartingBlockDoesNotMatch`.

However, that same check is still present in `submitDataAsCalldata` but now there is no test in `LineaRollup.ts` which expects the error `WrongParentShnarfFinalBlockNumber`.

This error is now detected inside `_validateSubmissionData` which reverts with new error `DataStartingBlockDoesNotMatch`. Hence the deprecated check in `submitDataAsCalldata` which reverts with `WrongParentShnarfFinalBlockNumber` is obsolete and can be deleted.

Recommended Mitigation: Remove the deprecated check from `submitDataAsCalldata`.

Linea: Fixed in commit 2eba80b merged in PR3191.

Cyfrin: Verified.