Stake.Link Metis Staking Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Hans

Assisting Auditors

November 18, 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

Current codebase implements a liquid staking solution for Metis sequencers, operating across Ethereum (L1) and Metis (L2) chains. Users can deposit tokens on L2 and receive liquid staking tokens (LST) in return, while their deposits are utilized for sequencer staking on L1.

Key Components

Following are the key contracts on the Metis(L2) Chain:

- L2Strategy: Manages deposits and LST minting on Metis
- WithdrawalPool: Handles queued withdrawals in FIFO order
- PriorityPool: Controls deposit flow and LST distribution

Following are the key contracts on Ethereum(L1) Chain:

- L1Strategy: Manages sequencer vault creation and staking on Ethereum
- SequencerVault: Individual vaults for Metis sequencer staking
- MetisLockingPool: Interface with Metis sequencer staking system

Following are the contracts responsible for cross chain communication:

- L1Transmitter/L2Transmitter: Handle cross-chain messaging via CCIP and native token briding via L1/L2 Standard Bridges. Both these contracts manage state synchronization between L1 and L2

5 Audit Scope

The audit focused on changes introduced in PR #128 implementing the Metis liquid staking protocol's core contracts. The scope included cross-chain communication, strategy management, and sequencer vault implementations.

Changes in the following files were part of the current audit scope

- L1Strategy.sol
- L2Strategy.sol
- SequencerVault.sol
- L1Transmitter.sol
- L2Transmitter.sol
- PriorityPool.sol
- WithdrawalPool.sol
- IL1StandardBridge.sol
- IL1StandardBridgeGasOracle.sol
- IL1Strategy.sol
- IL2StandardBridge.sol
- IL2StandardBridgeGasOracle.sol
- IL2Strategy.sol
- IMetisLockingInfo.sol
- IMetisLockingPool.sol

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the Stake.Link Metis Staking smart contracts provided by Stake.Link. In this period, a total of 6 issues were found.

Stake.Link implemented a liquid staking solution enabling Metis sequencer participation while maintaining token liquidity. Operating across Ethereum (L1) and Metis (L2), it allows users to deposit METIS tokens and receive liquid staking tokens (LST) while their deposits secure the network through sequencer staking. The system utilizes Chainlink's CCIP for cross-chain communication and implements a sophisticated reward distribution mechanism.

Our audit revealed several significant security concerns across the protocol's cross-chain architecture and economic design. The most critical findings involve vulnerabilities in the withdrawal mechanism that could lead to denial of service, potential exploitation of slashing events and cross-chain delays, and token accounting inconsistencies affecting LST valuation.

The codebase demonstrates high quality with comprehensive test coverage across both L1 and L2 components. The smart contracts are well-structured and documented, with clear separation of concerns between components.

All the major issues reported during the audit were successfully mitigated.

Summary

Project Name	Stake.Link Metis Staking
Repository	contracts
Commit	06de237d7785...
Audit Timeline	Oct 14th - Oct 25th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	3
Medium Risk	2
Low Risk	1
Informational	0
Gas Optimizations	0
Total Issues	6

Summary of Findings

[H-1] L1 CCIP messages use incorrect `tokensInTransitToL1` value leading to overvalued LST on Metis	Resolved
[H-2] Slashing loss redistribution vulnerability allows existing depositors to avoid losses at new depositors' expense	Acknowledged
[H-3] Attacker can manipulate queued withdrawal execution timing on withdrawal pool to prevent withdrawals on L1 indefinitely	Resolved
[M-1] Updating `operatorRewardPercentage` incorrectly redistributes already accrued rewards	Resolved
[M-2] Incorrect logic causes that SequencerVaults to not behave as expected when relocking or claiming rewards.	Resolved
[L-1] Transmitter address update can cause DoS for In-flight CCIP Messages	Acknowledged

7 Findings

7.1 High Risk

7.1.1 L1 CCIP messages use incorrect `tokensInTransitToL1` value leading to overvalued LST on Metis

Description: A discrepancy in withdrawal accounting between L1 and L2 chains can lead to an artificial inflation of the share price in the staking system.

In the `L1Transmitter::_executeUpdate` function, note that while actual amount withdrawn from `L1Strategy`is `toWithdraw`, the CCIP message assumes that the full `queuedWithdrawals` is withdrawn. In the scenario where `queuedWithdrawals > canWithdraw`, the CCIP message is sending an inflated value for `tokensInTransitToL1`

```
//L1Transmitter.sol
function _executeUpdate() private {
    // ...
    uint256 toWithdraw = queuedWithdrawals > canWithdraw ? canWithdraw : queuedWithdrawals;
    if (toWithdraw > minWithdrawalThreshold) { //@audit only when amount > min withdrawal
        l1Strategy.withdraw(toWithdraw); //@audit actual amount withdrawn is toWithdraw
        // ... (withdrawal logic)
    }

    Client.EVM2AnyMessage memory evm2AnyMessage = _buildCCIPUpdateMessage(
        totalDeposits,
        claimedRewards + queuedWithdrawals,  // @audit This uses full queuedWithdrawals, not actual
        ↪  withdrawn amount
        depositsSinceLastUpdate,
        opRewardReceivers,
        opRewardAmounts
    );
    // ...
}
```

In L2Transmitter.sol, the inflated withdrawal amount (`tokensInTransitFromL1`) is accepted without validation:

```
//L2Transmitter.sol
function _ccipReceive(Client.Any2EVMMessage memory _message) internal override {
    // ...
    (
        uint256 totalDeposits,
        uint256 tokensInTransitFromL1,
        uint256 tokensReceivedAtL1,
        address[] memory opRewardReceivers,
        uint256[] memory opRewardAmounts
    ) = abi.decode(_message.data, (uint256, uint256, uint256, address[], uint256[]));

    l2Strategy.handleUpdateFromL1(
        totalDeposits,
        tokensInTransitFromL1,  //@audit This value could be inflated
        tokensReceivedAtL1,
        opRewardReceivers,
        opRewardAmounts
    );
    // ...
}
```

3. In L2Strategy.sol, the inflated tokensInTransitFromL1 inflates deposit change calculations:

```
function getDepositChange() public view returns (int) {
    return
```

```
        int256(
            l1TotalDeposits +
                tokensInTransitToL1 +
                tokensInTransitFromL1 +  //@audit This value could be inflated
                token.balanceOf(address(this))
        ) - int256(totalDeposits);
}
```

4. Finally, in StakingPool.sol, the inflated deposit change leads to an artificial increase in totalRewards and share price:

```
function _updateStrategyRewards(uint256[] memory _strategyIdxs, bytes memory _data) private {
    int256 totalRewards;
    // ...
    for (uint256 i = 0; i < _strategyIdxs.length; ++i) {
        IStrategy strategy = IStrategy(strategies[_strategyIdxs[i]]);
        (int256 depositChange, , ) = strategy.updateDeposits(_data);
        totalRewards += depositChange;
    }

    if (totalRewards != 0) {
        totalStaked = uint256(int256(totalStaked) + totalRewards);
    }
    // ...
}
```

Impact: This vulnerability leads to an inflate share price of the liquid staking token.

Recommended Mitigation: Ensure accurate tracking of actual withdrawn amounts on L1:

```
uint256 actualWithdrawn = 0;
if (toWithdraw > minWithdrawalThreshold) {
    l1Strategy.withdraw(toWithdraw);
    actualWithdrawn = toWithdraw;
    // ... (rest of withdrawal logic)
}
```

Use the actual withdrawn amount in the CCIP message to L2:

```
Client.EVM2AnyMessage memory evm2AnyMessage = _buildCCIPUpdateMessage(
    totalDeposits,
    claimedRewards + actualWithdrawn,
    depositsSinceLastUpdate,
    opRewardReceivers,
    opRewardAmounts
);
```

Stake.Link: Fixed in 6a42a8c.

Cyfrin: Verified.

7.1.2 Slashing loss redistribution vulnerability allows existing depositors to avoid losses at new depositors' expense

Description: The protocol's staking system spans L1 (Ethereum) and L2 (Metis) with state updates propagated via Chainlink CCIP. Current design is exposed to timing vulnerability where slashing events on Metis are not

immediately reflected in the share price calculation on L2, allowing earlier depositors to exit at inflated share prices at the expense of new depositors.

The core issue stems from L2Strategy's total deposits calculation which depends on stale L1 state until a CCIP update is received:

```
// L2Strategy.sol
function getTotalDeposits() public view override returns (uint256) {
    return l1TotalDeposits + tokensInTransitToL1 + tokensInTransitFromL1 +
    ↪  token.balanceOf(address(this));
}
```

When slashing occurs on Metis, it's visible in L1Strategy.getDepositChange() but not reflected in totalDeposits until updateDeposits is called via CCIP:

```
// L1Strategy.sol
function getDepositChange() public view returns (int) {
    uint256 totalBalance = token.balanceOf(address(this));
    for (uint256 i = 0; i < vaults.length; ++i) {
        totalBalance += vaults[i].getTotalDeposits();
    }
    return int(totalBalance) - int(totalDeposits);
}
```

This can lead to a potential scenario that spans as follows:

-> Slashing occurs on Metis and is visible in L1Strategy.getDepositChange() -> New deposits on L2 receive shares at an inflated price (using stale L1 state) -> Earlier depositors can withdraw using this new liquidity at pre-slash value -> When the slash is finally reflected via CCIP, the last depositor bears most of the loss

Impact:

- New depositors could get fewer shares than they should even if they deposited after slashing on L1
- Front-running slashing can allow old depositors to exit using liquidity created by new depositors. As a result, a disproportionate slashing impact falls on recent depositors.

Proof of Concept: Run the following test. For this test, `allowInstantWithdrawals` is set to `true` on PriorityPool.

```
  it('front running on metis', async () => {
    const {
      signers,
      accounts,
      l2Strategy,
      l2Metistoken,
      l2Transmitter,
      stakingPool,
      priorityPool,
      metisLockingInfo,
      metisLockingPool,
      l1Metistoken,
      l1Transmitter,
      l1Strategy,
      vaults,
      offRamp,
    } = await loadFixture(deployFixture)

    // // setup Bob and Alice
    const bob = signers[7]
    const bobAddress = accounts[7]
    const alice = signers[8]
```

```
const aliceAddress = accounts[8]
const initialDeposit = toEther(1000)

await l2Metistoken.transfer(bobAddress, initialDeposit)
await l2Metistoken.connect(bob).approve(priorityPool.target, ethers.MaxUint256)
await stakingPool.connect(bob).approve(priorityPool.target, ethers.MaxUint256)

// Bob makes initial deposit
await priorityPool.connect(bob).deposit(initialDeposit, false, ['0x'])
assert.equal(await fromEther(await l2Strategy.getTotalDeposits()), fromEther(initialDeposit))
assert.equal(
  await fromEther(await l2Strategy.getTotalQueuedTokens()),
  fromEther(initialDeposit)
)

// executeUpdate to send message and tokens to L1
await l2Transmitter.executeUpdate({ value: toEther(10) })
assert.equal(await fromEther(await l2Strategy.getTotalDeposits()), fromEther(initialDeposit))
assert.equal(await fromEther(await l2Strategy.getTotalQueuedTokens()), 0)
assert.equal(await fromEther(await l2Metistoken.balanceOf(l2Strategy.target)), 0)
assert.equal(await fromEther(await l2Strategy.tokensInTransitToL1()), fromEther(initialDeposit))

// deposit the tokens received form L2 in L1
await l1Metistoken.transfer(l1Transmitter.target, initialDeposit) // mock the deposit via L2 Bridge
await l1Transmitter.depositTokensFromL2() // deposits balance into L1 Strategy
await l1Transmitter.depositQueuedTokens([1], [initialDeposit]) // deposits balance into vault 1

let vault1 = await ethers.getContractAt('SequencerVault', (await l1Strategy.getVaults())[1])
assert.equal(await fromEther(await vault1.getPrincipalDeposits()), fromEther(initialDeposit))

// Confirm initial deposit via CCIP back to L2
await offRamp
  .connect(signers[5])
  .executeSingleMessage(
    ethers.encodeBytes32String('messageId'),
    777,
    ethers.AbiCoder.defaultAbiCoder().encode(
      ['uint256', 'uint256', 'uint256', 'address[]', 'uint256[]'],
      [initialDeposit, 0, initialDeposit, [], []]
    ),
    l2Transmitter.target,
    []
  )

assert.equal(fromEther(await l2Strategy.l1TotalDeposits()), fromEther(initialDeposit))
assert.equal(fromEther(await l2Strategy.tokensInTransitFromL1()), 0)
assert.equal(fromEther(await l2Strategy.tokensInTransitToL1()), 0)
assert.equal(fromEther(await l2Strategy.getTotalDeposits()), fromEther(initialDeposit))
assert.equal(fromEther(await stakingPool.totalStaked()), fromEther(initialDeposit))

assert.equal(fromEther(await stakingPool.balanceOf(bobAddress)), fromEther(initialDeposit))

// simulate 20% slashing on Metis
const slashAmount = toEther(200) // 20% of 1000
await metisLockingPool.slashPrincipal(1, slashAmount)

assert.equal(fromEther(await l1Strategy.getDepositChange()), -200)

// At this point share price should still reflect 1000 tokens since slash isn't reflected
// Alice deposits after slash but before it's reflected
const aliceDeposit = toEther(1000)
await l2Metistoken.transfer(alice.address, aliceDeposit)
```

```
  await l2Metistoken.connect(alice).approve(priorityPool.target, ethers.MaxUint256)
  await stakingPool.connect(alice).approve(priorityPool.target, ethers.MaxUint256)

  // Alice deposits at inflated share price
  await priorityPool.connect(alice).deposit(aliceDeposit, false, ['0x'])

  assert.equal(
    fromEther(await stakingPool.totalStaked()),
    fromEther(initialDeposit) + fromEther(aliceDeposit)
  )

  assert.equal(
    fromEther(await l2Strategy.getTotalDeposits()),
    fromEther(initialDeposit) + fromEther(aliceDeposit)
  )
  assert.equal(fromEther(await stakingPool.balanceOf(aliceAddress)), fromEther(aliceDeposit)) //1:1
  ↪  conversion even though there is slashing
  assert.equal(fromEther(await l2Strategy.getTotalQueuedTokens()), fromEther(aliceDeposit))
  assert.equal(
    fromEther(await l2Metistoken.balanceOf(l2Strategy.target)),
    fromEther(aliceDeposit)
  )

  // Bob sees the slashing on L2 and places a withdrawal request
  const bobBalanceBefore = await l2Metistoken.balanceOf(bobAddress)
  const bobWithdrawAmount = await stakingPool.balanceOf(bobAddress) // try to redeem all of Bob's
  ↪  shares for metis token

  await priorityPool.connect(bob).withdraw(
    bobWithdrawAmount,
    0,
    0,
    [],
    false,
    false, // Don't queue, withdraw instantly from available liquidity
    ['0x']
  )

  // Process Bob's withdrawal using Alice's new deposits
  const bobBalanceAfter = await l2Metistoken.balanceOf(bob.address)
  assert.equal(fromEther(bobBalanceAfter - bobBalanceBefore), fromEther(bobWithdrawAmount)) //full
  ↪  withdrawal even after slashing

  // Finally CCIP message arrives reflecting the slash
  await offRamp.connect(signers[5]).executeSingleMessage(
    ethers.encodeBytes32String('messageId2'),
    777,
    ethers.AbiCoder.defaultAbiCoder().encode(
      ['uint256', 'uint256', 'uint256', 'address[]', 'uint256[]'],
      [toEther(800), 0, 0, [], []] // Now reflects the slash
    ),
    l2Transmitter.target,
    []
  )

  // Calculate Alice's actual value vs expected
  const aliceShareBalance = fromEther(await stakingPool.sharesOf(alice.address))
  const aliceCurrentValueOfShares = fromEther(await stakingPool.balanceOf(alice.address))
  assert.equal(aliceShareBalance, 1000)
  assert.equal(aliceCurrentValueOfShares, 800)
})
```

Recommended Mitigation: Consider the following recommendations:

- Disallow instant withdrawals on PriorityPool for the L2Strategy.
- Add a new owner controlled boolean state variable called `slashed`. Owner updates this variable to true as soon as slashing event occurs and resets it back to `false` once CCIP update is received on L2. When `slashed= true`, prevent all deposits and withdrawals on the strategy until the CCIP update by overriding the `canDeposit` and `canWithdraw` functions in `L2Strategy` as follows:

```
  // @audit override in L2Strategy.sol
  function canWithdraw() public view override returns (uint256) {
        if(slashed) return 0; //@audit prevent withdrawals until slashing is updated on L2
        super.canWithdraw()
    }

    function canDeposit() public view override returns (uint256) {
        if(slashed) return 0; //@audit prevent deposits until slashing is updated on L2
        super.canDeposit()
    }
```

Stake.Link Acknowleged. We will use the PriorityPool.sol pausing logic as soon as a slash is detected.

Cyfrin Acknowledged. Our recommendation was at a "strategy" level whereas the proposed mitigation is at a "pool" level. In case of multiple strategies, it is note worthy that the proposed mitigation will pause deposits across all strategies.

7.1.3 Attacker can manipulate queued withdrawal execution timing on withdrawal pool to prevent withdrawals on L1 indefinitely

Description: The protocol's `L2Transmitter::executeUpdate` function can be blocked through manipulation of withdrawal timing controls. The vulnerability exists due to a lack of access control on `WithdrawalPool::performUpkeep` combined with shared timing restrictions between withdrawals and L2 transmitter updates.

The key issue stems from the interaction between these functions:

```
// L2Transmitter.sol
function executeQueuedWithdrawals() public {
        uint256 queuedTokens = l2Strategy.getTotalQueuedTokens();
        uint256 queuedWithdrawals = withdrawalPool.getTotalQueuedWithdrawals();
        uint256 toWithdraw = MathUpgradeable.min(queuedTokens, queuedWithdrawals);

        if (toWithdraw == 0) revert CannotExecuteWithdrawals();

        bytes[] memory args = new bytes[](1);
        args[0] = "0x";
        withdrawalPool.performUpkeep(abi.encode(args)); //@audit attacker can front-run this to block
        ↪  execute update
    }

function executeUpdate() external payable {
    if (block.timestamp < timeOfLastUpdate + minTimeBetweenUpdates) { //@audit can only run this once
    ↪  every minTimeBetweenUpdates
        revert InsufficientTimeElapsed();
    }

    if (queuedTokens != 0 && queuedWithdrawals != 0) {
        executeQueuedWithdrawals();  // This calls WithdrawalPool.performUpkeep()
        queuedTokens = l2Strategy.getTotalQueuedTokens();
```

```
        queuedWithdrawals = withdrawalPool.getTotalQueuedWithdrawals();
    }
    // ... CCIP processing ...
}

// WithdrawalPool.sol
function performUpkeep(bytes calldata _performData) external {
    uint256 canWithdraw = priorityPool.canWithdraw(address(this), 0);
    uint256 totalQueued = _getStakeByShares(totalQueuedShareWithdrawals);
    if (
        totalQueued == 0 ||
        canWithdraw == 0 ||
        block.timestamp <= timeOfLastWithdrawal + minTimeBetweenWithdrawals
    ) revert NoUpkeepNeeded();

    timeOfLastWithdrawal = uint64(block.timestamp);
    // ... withdrawal processing ...
}
```

An attacker can block `L2Transmitter::executeUpdate` by doing the following:

-> Wait for `minTimeBetweenWithdrawals` to elapse -> Call `L2Transmitter::executeQueuedWithdrawals()` right after time elapses -> If `queuedWithdrawals` becomes 0: Queue a minimum withdrawal to restore `queuedWithdrawals > 0`

Eventually when `L2Transmitter::executeUpdate` is called, it will try to process withdrawals via `executeQueuedWithdrawals` since `queuedTokens` and `queuedWithdrawals` are both non-zero. This fails because `minTimeBetweenWithdrawals` has not elapsed causing the entire `executeUpdate` to revert.

Impact: This attack can be repeated to continuously prevent CCIP updates from being processed. This can potentially prevent withdrawals on L1 indefinitely.

Proof of Concept: Run the following POC, with `minTimeBetweenWithdrawals` for withdrawal pool as 50,000 and `minTimeBetweenUpdates` for L2Transmitter as 86400. Also, set `allowInstantWithdrawals` to false in PriorityPool.

```
  it('DOS executeUpdate on L1Transmitter', async () => {
    const {
      signers,
      accounts,
      l2Strategy,
      l2Metistoken,
      l2Transmitter,
      stakingPool,
      priorityPool,
      withdrawalPool,
      offRamp,
    } = await loadFixture(deployFixture)

    // // setup Bob and Alice
    const bob = signers[7]
    const bobAddress = accounts[7]
    const alice = signers[8]
    const aliceAddress = accounts[8]

    // 1.0 Fund Alice and Bob token and give necessary approvals
    await l2Metistoken.transfer(bobAddress, toEther(100))
    await l2Metistoken.connect(bob).approve(priorityPool.target, ethers.MaxUint256)
    await stakingPool.connect(bob).approve(priorityPool.target, ethers.MaxUint256)

    await l2Metistoken.transfer(aliceAddress, toEther(100))
    await l2Metistoken.connect(alice).approve(priorityPool.target, ethers.MaxUint256)
```

```
await stakingPool.connect(alice).approve(priorityPool.target, ethers.MaxUint256)

// 2.0 Bob deposits 50 METIS and calls executeUpdate on L2Transmitter
await priorityPool.connect(bob).deposit(toEther(50), false, ['0x'])
assert.equal(fromEther(await l2Strategy.getTotalDeposits()), 50)
assert.equal(fromEther(await l2Strategy.getTotalQueuedTokens()), 50)

await l2Transmitter.executeUpdate({ value: toEther(10) })

assert.equal(fromEther(await l2Strategy.getTotalDeposits()), 50)
assert.equal(fromEther(await l2Strategy.tokensInTransitToL1()), 50)

// 3.0 Assume tokens are invested in sequencer vaults on L1 and a message is returned
await offRamp
  .connect(signers[5]) //@note executes message on L2
  .executeSingleMessage(
    ethers.encodeBytes32String('messageId'),
    777,
    ethers.AbiCoder.defaultAbiCoder().encode(
      ['uint256', 'uint256', 'uint256', 'address[]', 'uint256[]'],
      [toEther(50), 0, toEther(50), [], []]
    ),
    l2Transmitter.target,
    []
  )
assert.equal(fromEther(await l2Strategy.getTotalDeposits()), 50)
assert.equal(fromEther(await l2Strategy.tokensInTransitToL1()), 0)
assert.equal(fromEther(await l2Strategy.tokensInTransitFromL1()), 0)
assert.equal(fromEther(await l2Strategy.l1TotalDeposits()), 50)

// 4.0 At this point Alice deposits 25 metis
await priorityPool.connect(alice).deposit(toEther(25), false, ['0x'])

assert.equal(fromEther(await l2Strategy.getTotalDeposits()), 75)
assert.equal(fromEther(await l2Strategy.getTotalQueuedTokens()), 25)
assert.equal(fromEther(await l2Metistoken.balanceOf(l2Strategy.target)), 25)

// 5.0 Right before executeUpdate is called on L2Transmitter, Bob withdraws 10 metis
await priorityPool.connect(bob).withdraw(
  toEther(10),
  0,
  0,
  [],
  false,
  true, // queue withdrawal
  ['0x']
)

assert.equal(fromEther(await withdrawalPool.getTotalQueuedWithdrawals()), 10)

// 6.0 Bob then calls executeQueuedWithdrawals
await time.increase(50001) // 50000 seconds is min time between withdrawals
await l2Transmitter.executeQueuedWithdrawals()

// At this point queued withdrawals is 5, queued tokens is 0
assert.equal(fromEther(await l2Strategy.getTotalQueuedTokens()), 15)
assert.equal(fromEther(await withdrawalPool.getTotalQueuedWithdrawals()), 0)

// // 7.0 Bob again places a 10 metis withdrawal
await priorityPool.connect(bob).withdraw(
  toEther(10),
  0,
```

```
      0,
      [],
      false,
      true, // queue withdrawal
      ['0x']
    )
    assert.equal(fromEther(await l2Strategy.getTotalQueuedTokens()), 15)
    assert.equal(fromEther(await withdrawalPool.getTotalQueuedWithdrawals()), 10)

    await time.increase(36401) // 50001 + 36401 > 86400. So from the initial time, we are now 86401
    ↪  seconds
    // we should be able to again run execute update

    await expect(l2Transmitter.executeUpdate({ value: toEther(10) })).to.be.revertedWithCustomError(
      withdrawalPool,
      'NoUpkeepNeeded'
    )
  })
```

Recommended Mitigation: In `L2Transmitter::executeQueuedWithdrawals`, consider calling `WithdrawalPool::performUpkeep` only when `WithdrawalPool::checkUpkeep` is `true`.

Stake.Link Fixed in 835ed4e.

Cyfrin Verified.

7.2 Medium Risk

7.2.1 Updating `operatorRewardPercentage` incorrectly redistributes already accrued rewards

Description: The `operatorRewardPercentage` in L1Strategy determines how rewards are split between operators and other recipients. When this percentage is changed, previously generated but undistributed rewards are calculated using the new percentage rather than the percentage that was in effect when they were generated.

Key code in `SequencerVault.sol`where rewards are calculated:

```
// SequencerVault.sol
function updateDeposits(
    uint256 _minRewards,
    uint32 _l2Gas
) external payable onlyVaultController returns (uint256, uint256, uint256) {
    // Calculate total deposits and changes
    uint256 principal = getPrincipalDeposits();
    uint256 rewards = getRewards();
    uint256 totalDeposits = principal + rewards;
    int256 depositChange = int256(totalDeposits) - int256(uint256(trackedTotalDeposits));

    uint256 opRewards;
    if (depositChange > 0) {
        // Uses current operatorRewardPercentage for all accrued rewards
        opRewards = (uint256(depositChange) * vaultController.operatorRewardPercentage()) / 10000;
        trackedTotalDeposits = SafeCastUpgradeable.toUint128(totalDeposits);
    }
    //...
}
```

Consider the following scenario:

- operatorRewardPercentage = 10%
- 100 tokens of rewards accrue (10 for operators, 90 for others)
- operatorRewardPercentage changed to 25%
- On next update, same 100 tokens are split: 25 for operators, 75 for others
- Recipients lose 15 tokens they had effectively earned

Impact: *Changing percentage up*: Other recipients lose already earned rewards *Changing percentage down*: Operators lose already earned rewards

Recommended Mitigation: Consider distributing the already earned rewards (if any) using the current `operatorRewardPercentage` before updating it to a new value. This would involve updating the deposits on the Strategy (which in turn would update the deposits on the SequencerVaults), and, send an update the L2 to mint the corresponding shares for the earned rewards.

Stake.Link Fixed in 7c5d0aa.

Cyfrin Verified. Explicit In-line comments added to run `executeUpdate` before updating operator reward percentage.

7.2.2 Incorrect logic causes that SequencerVaults to not behave as expected when relocking or claiming rewards.

Description: A logic error exists in SequencerVault.updateDeposits() where rewards are not processed according to the function's documented behavior. According to the function in-line comments below, the comment `(set 0 to skip reward claiming)` indicates that when minRewards=0, the function should skip claiming but still process re-locking of rewards.

```
//SequencerVault.sol
/**
 * @notice Updates the deposit and reward accounting for this vault
 * @dev will only pay out rewards if the vault is net positive when accounting for lost deposits
 * @param _minRewards min amount of rewards to relock/claim (set 0 to skip reward claiming) --> @audit
 * @param _l2Gas L2 gasLimit for bridging rewards
 * @return the current total deposits in this vault
 * @return the operator rewards earned by this vault since the last update
 * @return the rewards that were claimed in this update
 */
 function updateDeposits(
        uint256 _minRewards,
        uint32 _l2Gas
    ) external payable onlyVaultController returns (uint256, uint256, uint256) {
 // logic
}
```

However, the actual implementation:

```
//SequencerVault.sol
function updateDeposits(
    uint256 _minRewards,
    uint32 _l2Gas
) external payable onlyVaultController returns (uint256, uint256, uint256) {
    uint256 principal = getPrincipalDeposits();
    uint256 rewards = getRewards();
    uint256 totalDeposits = principal + rewards;

    // ... code

    // Incorrectly skips all reward processing if minRewards=0
    if (_minRewards != 0 && rewards >= _minRewards) {
        if (
            (principal + rewards) <= vaultController.getVaultDepositMax() &&
            exitDelayEndTime == 0
        ) {
            lockingPool.relock(seqId, 0, true);
        } else {
            lockingPool.withdrawRewards{value: msg.value}(seqId, _l2Gas);
            trackedTotalDeposits -= SafeCastUpgradeable.toUint128(rewards);
            totalDeposits -= rewards;
            claimedRewards = rewards;
        }
    }
    // @audit No else block to handle relocking when minRewards=0

   // code..
    return (totalDeposits, opRewards, claimedRewards);
}
```

The implementation diverges from intended behavior:

When `minRewards=0`: Expected: Skip claiming but process relocking Actual: Skips all reward processing

Impact: When `minRewards == 0`, rewards are neither relocked nor claimed. This effectively leads to stuck rewards in `MetisLockingPool` until the sequencer is active.

Proof of Concept: Add the following to `l1-strategy.test.ts`

```
 const setupSequencerWithRewards = async () => {
```

```
  const { strategy, metisLockingPool, token, accounts } = await loadFixture(deployFixture)

  // Setup vault and deposit initial amount
  await strategy.addVault('0x5555', accounts[1], accounts[2])
  const vaults = await strategy.getVaults()
  const vault0 = await ethers.getContractAt('SequencerVault', vaults[0])

  await strategy.deposit(toEther(500))

  await strategy.depositQueuedTokens([0], [toEther(500)])
  assert.equal(fromEther(await vault0.getPrincipalDeposits()), 500)
  assert.equal(fromEther(await vault0.getTotalDeposits()), 500)

  // Add rewards to sequencer
  await metisLockingPool.addReward(1, toEther(50))
  assert.equal(fromEther(await vault0.getTotalDeposits()), 550)

  return { strategy, vault0, metisLockingPool }
}

it('rewards not processed when minRewardsToClaim is 0', async () => {
  const { strategy, vault0, metisLockingPool } = await setupSequencerWithRewards()

  // Set minRewardsToClaim to 0
  await strategy.setMinRewardsToClaim(0)

  const rewardsBefore = await vault0.getRewards()
  assert.equal(fromEther(rewardsBefore), 50)

  // Update deposits
  await strategy.updateDeposits(0, 0)

  // Rewards should be relocked but aren't
  const rewardsAfter = await vault0.getRewards()
  assert.equal(fromEther(rewardsAfter), 50)
})
```

Recommended Mitigation: Consider updating the current logic regarding reward management:

- rewards should be claimed when `minRewards > 0` && rewards exceed the minimum threshold!
- rewards should be relocked when `minRewards == 0`

Stake.Link Fixed in commit 0ec7685

Cyfrin Verified. In-line comments updated to ensure consistency with implemented logic.

7.3 Low Risk

7.3.1 Transmitter address update can cause DoS for In-flight CCIP Messages

Description: The L1Transmitter and L2Transmitter contracts allow for updating the address of their counterpart transmitter on the other chain. However, this update mechanism does not account for in-flight CCIP messages, potentially causing them to be rejected upon arrival.

In `L1Transmitter.sol`:

```
//L1Transmitter.sol
function setL2Transmitter(address _l2Transmitter) external onlyOwner {
    l2Transmitter = _l2Transmitter;
}

function _ccipReceive(Client.Any2EVMMessage memory _message) internal override {
    if (_message.sourceChainSelector != l2ChainSelector) revert InvalidSourceChain();
    if (abi.decode(_message.sender, (address)) != l2Transmitter) revert InvalidSender();
    // ... rest of the function
}
```

Similarly, in `L2Transmitter.sol`:

```
function setL1Transmitter(address _l1Transmitter) external onlyOwner {
    l1Transmitter = _l1Transmitter;
}

function _ccipReceive(Client.Any2EVMMessage memory _message) internal override {
    if (_message.sourceChainSelector != l1ChainSelector) revert InvalidSourceChain();
    if (abi.decode(_message.sender, (address)) != l1Transmitter) revert InvalidSender();
    // ... rest of the function
}
```

The issue arises because the `_ccipReceive` function in both contracts checks the sender against the stored transmitter address. If this address is updated while there are messages in transit, those messages will be rejected upon arrival because their sender no longer matches the updated transmitter address.

Impact: Can lead to a Denial of Service for cross-chain communication. Any in-flight messages at the time of a transmitter address update will be rejected, potentially causing:

- Loss of critical updates or withdrawals
- Desynchronization of state between L1 and L2

Recommended Mitigation: Since both the transmitters are upgradeable, consider removing this function. When needed, the implementation of transmitter can be changed without having a need to update the address. Alternatively, make sure there are no in-flight messages when the owner is calling these functions to set a new address.

Stake.Link Acknowledged. Owner will ensure there are no in-flight messages when calling these functions.

Cyfrin Acknowledged.