Espresso Staking Strategy Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Kage

Al Qa-qa

February 4, 2026

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

The Espresso Strategy is a liquid staking integration that allows users to stake ESP tokens through Stake.Link's infrastructure while delegating to Espresso Network validators. It follows a vault-per-validator architecture.

Key contracts include

EspressoStrategy - Main strategy managing multiple vaults, token accounting, and fee distribution **EspressoVault** - Individual vault delegating to a single Espresso validator **EspressoFundFlowController** - Orchestrates deposit/withdrawal operations and keeper automation

5 Audit Scope

The audit scope was limited to:

```
contracts/espressoStaking/EspressoFundFlowController.sol
contracts/espressoStaking/EspressoStrategy.sol
contracts/espressoStaking/EspressoVault.sol
```

6 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the Espresso Staking Strategy smart contracts provided by Stake.Link. In this period, a total of 9 issues were found.

Summary

Project Name	Espresso Staking Strategy
Repository	contracts
Commit	74e6e09ccc28...
Fix Commit	e37f39f64e35...
Audit Timeline	Jan 26 - Feb 2nd, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	5
Informational	2
Gas Optimizations	0
Total Issues	9

Summary of Findings

[M-1] Queued tokens can be redeposited before withdrawals are processed	Resolved
[M-2] Denial of service in fee management due to mandatory reward updates	Resolved
[L-1] Not tracking the amount withdrawn from inactive vaults can result in the reuse of already unbonded assets	Acknowledged
[L-2] Unclaimed rewards lost when removing vaults	Resolved
[L-3] Lack of `maxRewardChangeBPS` validation in `withdrawRewards` and `restakeRewards` allows reward limit bypass	Acknowledged
[L-4] Token donation can block reward updates in some scenarios	Acknowledged
[L-5] `EspressoVault::exitIsWithdrawable` does not check if assets were already withdrawn	Resolved
[I-1] `FundFlowController::unbondVaults` can be permanently DoS'ed for Espresso Staking Pool with multiple strategies	Acknowledged
[I-2] Changing `vaultImplementation` can DoS `EspressoStrategy::addVault` function	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Queued tokens can be redeposited before withdrawals are processed

Description: `EspressoFundFlowController::shouldDepositQueuedTokens` does not account for pending withdrawal requests when determining whether to deposit tokens.

`StakeTableV2.exitEscrowPeriod` controls the amount of time users funds are locked after unbonding - as per the contract, this value can be anywhere between 2-14 days. `WithdrawalPool.minTimeBetweenWithdrawals` is the time gap between two `performUpkeep` calls where queued withdrawals are executed (currently set to 3 days in deployment scripts).

When `exitEscrowPeriod < minTimeBetweenWithdrawals`, a gap exists where:

- Tokens have completed escrow and are claimed via `EspressoStrategy::claimUnbond`
- But `WithdrawalPool::performUpkeep` cannot yet execute due to the time restriction
- Tokens sit in `totalQueued` during this gap
- `EspressoFundController::shouldDepositQueuedTokens` returns `true` signaling `depositController` to redeposit

```
//EspressoFundController.sol
function shouldDepositQueuedTokens() external view returns (bool, uint256) {
      uint256 queuedTokens = strategy.totalQueued();
      return (queuedTokens > 0, queuedTokens);  // No check for pending withdrawals
  }
```

`EspressoFundFlowController.sol::withdrawVaults` attempts to process withdrawals immediately after claiming:

```
 // EspressoFundFlowController.sol:155-163
  function withdrawVaults(uint256[] calldata _vaultIds) external {
      strategy.claimUnbond(_vaultIds);  // Tokens → totalQueued

      (bool upkeepNeeded, ) = withdrawalPool.checkUpkeep("");

      if (upkeepNeeded) {
          withdrawalPool.performUpkeep("");  // @audit May not execute if minTime not passed
      }
  }
```

If `WithdrawalPool::checkUpkeep` returns false because `minTimeBetweenWithdrawals` has not elapsed, the tokens remain in `totalQueued` and can be re-deposited.

Impact: Tokens explicitly unbonded for withdrawals can be re-deposited, defeating the purpose of the unbonding operation.

Recommended Mitigation: Consider updating `EspressoFundFlowController::shouldDepositQueuedTokens` to reserve tokens for pending withdrawals.

```
 function shouldDepositQueuedTokens() external view returns (bool, uint256) {
      uint256 queuedTokens = strategy.totalQueued();
--       return (queuedTokens > 0, queuedTokens);
++      uint256 queuedWithdrawals = withdrawalPool.getTotalQueuedWithdrawals();

++      // Reserve tokens for pending withdrawals
++      if (queuedTokens <= queuedWithdrawals) {
++         return (false, 0);
++      }
```

```
++      // Only deposit excess beyond withdrawal needs
++      uint256 excessTokens = queuedTokens - queuedWithdrawals;
++      return (excessTokens > 0, excessTokens);
  }
```

Stake.Link: Fixed in 50768a6.

Cyfrin: Verified.

7.1.2 Denial of service in fee management due to mandatory reward updates

Description

In `EspressoStaking`, the owner can add, remove, or update fees. This includes changing the receiver address and the associated fee percentage. Whenever a fee update occurs, the internal function `_updateStrategyRewards` is called.

contracts/espressoStaking/EspressoStrategy.sol#addFee/updateFee

```
    function addFee(address _receiver, uint256 _feeBasisPoints) external onlyOwner {
        _updateStrategyRewards();
        fees.push(Fee(_receiver, _feeBasisPoints));
        if (_totalFeesBasisPoints() > 3000) revert FeesTooLarge();
        emit AddFee(_receiver, _feeBasisPoints);
    }

    function updateFee( ... ) external onlyOwner {
        _updateStrategyRewards();

        if (_feeBasisPoints == 0) { ... } else { ... }
    }
```

The `_updateStrategyRewards` function triggers `StakingPool.updateStrategyRewards`, which in turn executes `EspressoStaking::updateDeposits`. If any fees have accumulated, the system immediately attempts to distribute them to the registered fee receivers.

When distributing fees, if a receiver is a contract, the system treats it as an `ERC677Receiver` and calls the `onTokenTransfer` hook.

contracts/core/StakingPool.sol

```
    function _updateStrategyRewards(uint256[] memory _strategyIdxs, bytes memory _data) private {
        ...
        // distribute fees to receivers if there are any
        if (totalFeeAmounts > 0) {
            ...
            for (uint256 i = 0; i < receivers.length; i++) {
                for (uint256 j = 0; j < receivers[i].length; j++) {
                    if (feesPaidCount == totalFeeCount - 1) {
                        transferAndCallFrom(
                            address(this),
                            receivers[i][j],
                            balanceOf(address(this)),
                            "0x"
                        );
                    } else {
>>                      transferAndCallFrom(address(this), receivers[i][j], feeAmounts[i][j], "0x");
                        feesPaidCount++;
                    }
                }
            }
```

```
        }
        emit UpdateStrategyRewards(msg.sender, totalStaked, totalRewards, totalFeeAmounts);
    }

    function transferAndCallFrom( ... ) internal returns (bool) {
        _transfer(_sender, _to, _value);
        if (isContract(_to)) {
            contractFallback(_sender, _to, _value, _data);
        }
        return true;
    }

    function contractFallback( ... ) internal {
        IERC677Receiver receiver = IERC677Receiver(_to);
        receiver.onTokenTransfer(_sender, _value, _data);
    }
```

The core issue is that changing or removing a fee **enforces** an update that distributes accumulated fees. If a fee receiver is a malicious contract (or a compromised EOA upgraded to a contract via EIP-7702) that reverts upon receiving a call to `onTokenTransfer`, the entire transaction will fail.

Because `addFee` and `updateFee` both require a successful call to `_updateStrategyRewards`, a single compromised receiver can block its own removal and prevent any further fee configuration changes. This state can only be bypassed if `totalFeeAmounts` is zero, which typically only occurs during a "negative" rebase (slashing)—a rare and non-standard condition.

Note that a zero address fee receiver also causes the same issue.

Impact

- Fees across the entire strategy become stuck if one receiver reverts.
- The owner cannot remove a malicious receiver or adjust percentages because the mandatory distribution check will always revert.
- This can only be mitigated by a full contract upgrade if the logic is not flexible.

Proof of Concept

1. `EspressoStaking` has multiple fee receivers.
2. One receiver is a contract (or an account controlled by an attacker) that is programmed to `revert()` when `onTokenTransfer` is called.
3. Fees accumulate in the strategy.
4. The owner recognizes the malicious receiver and attempts to call `updateFee` to set their basis points to 0 (effectively removing them).
5. `updateFee` calls `_updateStrategyRewards`, which attempts to push the accumulated fees to the malicious receiver.
6. The malicious receiver reverts, causing the owner's transaction to fail. The malicious receiver remains in the system indefinitely.

Recommended Mitigation

- Consider adding zero address validation when adding/updating fee receiver.
- Consider adding a dedicated function to remove a fee receiver that **skips** the `_updateStrategyRewards` call, allowing the owner to update a malicious receiver even when distribution is failing, this is suitable in case `StakingPool` is immutable

If `StakingPool` can be changed:

- Consider modifying `StakingPool` to store accumulated fees in a mapping, allowing receivers to claim their fees individually (Pull pattern). This prevents one receiver from affecting the execution flow of others.
- Alternatively, consider wrapping the `contractFallback` call in a `try/catch` block. If a receiver reverts, store their specific share in a "failed transfers" mapping for later manual recovery, rather than reverting the entire global update.

Note: Scope of Impact

[!IMPORTANT] **This vulnerability is not limited to `EspressoStaking`. The flaw resides within the core `StakingPool` implementation itself:**

- **Live Protocols:** This issue directly affects active deployments, including **Link Staking**, **ETH Staking**, and **Polygon Staking**. Since these contracts are live, a compromised or malicious fee receiver could immediately lock fee management functions.
- **OperatorVCS Integration:** The `OperatorVCS` contract is also vulnerable. Any administrative actions—such as **removing a vault** or **adjusting fee percentages**—trigger the `updateStrategyRewards` function. Consequently, a single reverting fee receiver would effectively DoS these management operations, preventing the protocol from offboarding vaults or updating its economic parameters.

Stake.Link: Zero address validation fixed in 6cf8d4b. Fee receivers are trusted, risk of trusted receiver reverting is acknowledged.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Not tracking the amount withdrawn from inactive vaults can result in the reuse of already unbonded assets

Description: Unbonding process is processed by anyone, whenever there is an amount queued in `WithdrawalPool`, anyone can call `EspressoFundFlowController::unbondVaults` to unbond the assets needed for withdrawing.

When withdrawing from `EspressoStrategy`, we loop through all Vaults, if the Vault is in `inActive` sate, we use vault totalDeposits, otherwise, we unbond the principle

EspressoStrategy::unbond

```
    function unbond(uint256 _toUnbond) external onlyFundFlowController {
        ...
        while (toUnbondRemaining != 0) {
            IEspressoVault vault = vaults[i];
            uint256 principalDeposits = vault.getPrincipalDeposits();

            if (!vault.isActive()) {
                uint256 deposits = vault.getTotalDeposits();
>>              toUnbondRemaining = toUnbondRemaining <= deposits
                    ? 0
                    : toUnbondRemaining - deposits;
            } else if (principalDeposits != 0) {
                ...
>>              vault.unbond(vaultToUnbond);

                toUnbondRemaining -= vaultToUnbond;
                ++numVaultsUnbonded;
            }

            i = (i + 1) % vaults.length;
            if (i == vaultWithdrawalIndex) break;
        }

        ...
    }
```

When calling `unbond` on a given `EspressoVault` we call `EspressoStaking::undelegate`, which decreases delegations (i.e decreases PrincipleDeposit. But in case the vault is in `inActive` state, we simply just subtract the value of that inActive vault, but there is no storing or recording of the value we took from that inActive vault. So if we called `unbond` again, and we reached this vault, it can end up simulating unbonding assets, that are already unbonded, meaning we will not be able to withdraw the unbonded assets we requested to fill `WithdrawalPool` queue.

Impact:

- Duplicating `inActive` vault assets by unbonding the same assets twice.

Proof of Concept: We made a script that do the following:

- There are 2 Vaults each of 100 Ether
- second vault is in `inActive` state
- We unbonded `150` Ether
- A new Vault is Added and there is 100 Eth deposited in it
- We requested another unbond of `100 ETH` before finalizing the `inActive` vault
- Instead of taking `50 ETH` from the newVault to fill the `100 ETH` (50 from prev inActive vault and 50 from the new one), it unbonded `100 ETH` from inactive vault, leaving new Vault deposits as it is.

You can test the following behaviour by adding the following test script to `test/espressoStaking/espresso-strategy.test.ts`

```
it('Cyfrin Audit: unbond uncorrectly unbounds', async () => {
  const { signers, accounts, stakingPool, strategy, vaults, espressoStaking, validators } =
    await loadFixture(deployFixture)

  // Removing The Third Vault, We only Have two Vaults Now
  await strategy.removeVaults([2])


  // Test unbond counts inactive vault deposits toward unbond amount
  await stakingPool.deposit(accounts[0], toEther(200), ['0x'])
  await strategy.depositQueuedTokens([0, 1], [toEther(100), toEther(100)])

  // Exit validator 1
  await espressoStaking.exitValidator(validators[1])
  assert.equal(await vaults[1].isActive(), false)

  // Unbond 150: vault 0 has 100, vault 1 is inactive with 100 (counted but not unbonded)
  // So we need 150, vault 0 gives 100 (unbonded), vault 1 gives 100 (counted), done at 200 >= 150
  await strategy.unbond(toEther(150))

  assert.equal(fromEther(await vaults[0].getQueuedWithdrawals()), 100)
  assert.equal(fromEther(await vaults[1].getQueuedWithdrawals()), 0) // As it is inactive


  // Register a new validator
  const newValidator = accounts[8]
  await espressoStaking.registerValidator(newValidator)
  await strategy.addVault(newValidator)

  const vaultsAfterAdd = await strategy.getVaults()
  assert.equal(vaultsAfterAdd.length, 3)
  const newVaultAddress = vaultsAfterAdd[2]
  const newVault: EspressoVault = await ethers.getContractAt('EspressoVault', newVaultAddress)
  // --------

  // Stake 100 Ether to the new Vault
  await stakingPool.deposit(accounts[0], toEther(100), ['0x'])
  await strategy.depositQueuedTokens([2], [toEther(100)])
  assert.equal(fromEther(await newVault.getPrincipalDeposits()), 100)

  // Complete the unbond cycle
  await time.increase(exitEscrowPeriod)
  await strategy.claimUnbond([0])

  // Current State (Funds Left):
  // Vault 0: 0
  // Vault 1: 50 (It is Inactive and we unbounded 50 from it before)
  // newVault: 100 (We just deposited 100 into it)

  // Validation, the current Index is `v0` which has `0` amount
  assert.equal(Number(await strategy.vaultWithdrawalIndex()), 0)
  assert.equal(fromEther(await vaults[0].getPrincipalDeposits()), 0)
  assert.equal(fromEther(await newVault.getPrincipalDeposits()), 100)

  // Unbond 100: Vault 0 is empty so we skip it, vault 1 has 50 so we take 50 from it
  // and vault 2 has 100 so we take 50 from it
  await strategy.unbond(toEther(100))

  // Expected State After Unbond:
  // Vault 0: 0
```

```
    // Vault 1: 0 (We took 50 from it)
    // Vault 2: 50 (We took 50 from it)

    assert.equal(Number(await strategy.numVaultsUnbonding()), 0)
    assert.equal(fromEther(await vaults[0].getQueuedWithdrawals()), 0)
    // The Third Vault has 100 ether left in it instead of `50` as we double accumilate the `50` from
    ↪  inActive vault (v1)
    assert.equal(fromEther(await newVault.getQueuedWithdrawals()), 0)
    assert.equal(fromEther(await newVault.getPrincipalDeposits()), 100)

  })
```

Recommended Mitigation: We should track the `inActive` vault balance, so that we do not reuse already unbonded assets

Stake.Link: Acknowledged.

7.2.2 Unclaimed rewards lost when removing vaults

Description: `EspressoStrategy::removeVaults` allows the owner to remove vaults from the strategy. While the function checks that principal deposits are withdrawn, it does not verify that rewards have been claimed before removal.

The NatSpec documentation acknowledges this:

```
  /**
   * @dev ... Will not check for unclaimed rewards so rewards must be claimed
   * before removing a vault, otherwise they will be lost.
   */
```

However, this is not enforced in the code.

Impact: If the owner removes a vault without first claiming rewards, there is no mechanism to recover those rewards.

Recommended Mitigation: Consider checking unclaimed rewards via the `EspressoVault::getRewards` and revert if the vault has non zero rewards.

Stake.link: Fixed in a299ca7

Cyfrin: Verified.

7.2.3 Lack of `maxRewardChangeBPS` validation in `withdrawRewards` and `restakeRewards` allows reward limit bypass

Description: When the Reward Oracle updates Vault rewards, it verifies that any increase in rewards does not exceed a specific threshold. If the increase is too large, the update reverts. This safeguard prevents massive reward spikes before the `StakingPool` calls `EspressoStrategy::updateDeposits` to settle fee distribution and update `totalDeposits`.

contracts/espressoStaking/EspressoStrategy.sol#updateLifetimeRewards

```
    function updateLifetimeRewards( ... ) external onlyRewardsOracle {
        if (_vaultIds.length != _lifetimeRewards.length) revert InvalidParamLengths();

        for (uint256 i = 0; i < _vaultIds.length; ++i) {
            vaults[_vaultIds[i]].updateLifetimeRewards(_lifetimeRewards[i]);
        }

        int256 rewards = getDepositChange();
>>      if (rewards > 0 && uint256(rewards) > (totalDeposits * maxRewardChangeBPS) / BASIS_POINTS)
            revert RewardsTooHigh();
```

```
        emit UpdateLifetimeRewards();
    }
```

The issue is that `updateLifetimeRewards` is not the only path to updating rewards. In the Espresso Network `RewardClaim` contract, reward updates are handled by verifying `authData` against the Light Client Root (Merkle Tree Proof). According to Espresso, rewards should be updated at least once per epoch (roughly every 24 hours), allowing anyone to claim them.

https://docs.espressosys.com/network/releases/testnets/decaf-testnet/running-a-node#staking

> With the initial release of Proof-of-stake, participation is limited to a dynamic, permissionless set of 100 nodes. **In each epoch (period of roughly 24 hours)** the 100 nodes with the most delegated stake form the active participation set.

https://docs.espressosys.com/network/concepts/the-espresso-network/internal-functionality/light-client#updating-and-verifying-lightclientstate

> Replica nodes update the snapshot of the stake table at the beginning of an epoch and this snapshot is used to define the set of stakers for the next epoch. **The light client state must be updated at least once per epoch**.

Consequently, a new proof is available daily to claim `EspressoVault` rewards. Any user can trigger a claim by providing a proof via `EspressoFundFlowController::restakeRewards` or `EspressoFundFlowController::withdrawRewards`. These functions call `claimRewards` and `updateLifetimeRewards` directly, but they fail to check if the total rewards exceed the `maxRewardChangeBPS` limit set by the protocol.

```
    // contracts/espressoStaking/EspressoFundFlowController.sol#withdrawRewards
    function withdrawRewards( ... ) external {
        strategy.withdrawRewards(_vaultIds, _lifetimeRewards, _authData);
    }

    //contracts/espressoStaking/EspressoStrategy.sol#withdrawRewards
    function withdrawRewards( ... ) external onlyFundFlowController {
        ...
        for (uint256 i = 0; i < _vaultIds.length; ++i) {
>>          vaults[_vaultIds[i]].withdrawRewards(_lifetimeRewards[i], _authData[i]);
        }

        totalQueued += token.balanceOf(address(this)) - preBalance;

        emit WithdrawRewards();
    }

    // contracts/espressoStaking/EspressoVault.sol#withdrawRewards
    function withdrawRewards( ...  ) external onlyVaultController {
>>      _updateLifetimeRewards(_lifetimeRewards);

        if (getRewards() != 0) {
            espressoRewards.claimRewards(_lifetimeRewards, _authData);

            uint256 balance = token.balanceOf(address(this));
            token.safeTransfer(msg.sender, balance);
        }
    }
```

If the reward oracle fails to post rewards for an extended period, a user can obtain a proof through the Espresso API and call `restakeRewards` or `withdrawRewards`. Even if the accumulated rewards far exceed `maxRewardChangeBPS`, the transaction will process without reverting.

https://docs.espressosys.com/network/api-reference/espresso-api/state-api#get-reward-state

Get a Merkle proof proving the balance of a certain reward account in a given snapshot of the state.

Impact: A large increase in the deposit change value can occur in a single transaction, escaping the `StakingPool` restriction enforced during `updateDeposits`. This results in `updateDeposits` being called with excessively large values or over inaccurate intervals, leading to incorrect fee accumulation and reward distribution.

Proof of Concept:

1. The `maxRewardChangeBPS` is set to `1%`.
2. Vaults accumulate significant fees; earnings reach `2%`, but the Oracle has not yet updated.
3. A standard Oracle update would only allow a `1%` increase before requiring `StakingPool` to call `updateDeposits`.
4. Instead, an attacker (or user) retrieves the latest Merkle Root and calls `EspressoFundFlowController::withdrawRewards`.
5. All rewards are withdrawn, and `DepositChange` increases by `2%` instantly.
6. This bypasses the `1%` `maxRewardChangeBPS` restriction, leading to inaccurate protocol accounting.

Recommended Mitigation: After the execution of `EspressoFundFlowController::withdrawRewards` or `EspressoFundFlowController::restakeRewards`, the protocol should verify that `DepositChange` has not increased beyond the `maxRewardChangeBPS` limit.

Stake.Link: Acknowledged.

7.2.4 Token donation can block reward updates in some scenarios

Description: `EspressoStrategy::updateLifetimeRewards` uses `EspressoStrategy::getDepositChange` to enforce a maximum reward increase per update. However, `EspressoStrategy::getDepositChange` includes token balances that can be manipulated via direct transfers.

An attacker can donate a small amount of tokens to the strategy or vaults to artificially inflate deposit change, causing legitimate reward updates to revert. The attack cost can be minimal when natural reward accrual is close to the `maxRewardChangeBPS` threshold.

The reward update check in `EspressoStrategy::updateLifetimeRewards`:

```
// EspressoStrategy.sol
  function updateLifetimeRewards(
      uint256[] calldata _vaultIds,
      uint256[] calldata _lifetimeRewards
  ) external onlyRewardsOracle {
      for (uint256 i = 0; i < _vaultIds.length; ++i) {
          vaults[_vaultIds[i]].updateLifetimeRewards(_lifetimeRewards[i]);
      }

      int256 rewards = getDepositChange();  //@audit this can be manipulated by donation
      if (rewards > 0 && uint256(rewards) > (totalDeposits * maxRewardChangeBPS) / BASIS_POINTS)
      ↪  //@audit causing this to revert
          revert RewardsTooHigh();
  }
```

The `EspressoStrategy::getDepositChange` function output can be manipulated by donating tokens to the strategy contract:

```
  function getDepositChange() public view returns (int) {
      uint256 totalBalance = token.balanceOf(address(this));  // @audit can inflate this by donation
      for (uint256 i = 0; i < vaults.length; ++i) {
          totalBalance += vaults[i].getTotalDeposits();
      }
      return int(totalBalance) - int(totalDeposits);  //@note can increase this
  }
```

Note that the minimum donation required depends on how close natural rewards are to the threshold. In cases where the natural rewards are very close to the threshold, a small donation can brick the reward update workflow.

Impact: The rewardsOracle cannot update lifetime rewards, causing rewards to go unrecognized in the accounting system. However, it is noted that such an attack is temporary, and will only exist until strategy rewards are updated.

Recommended Mitigation: Consider calculating actual reward increase directly rather than inferring from total deposit change:

```
 function updateLifetimeRewards(
      uint256[] calldata _vaultIds,
      uint256[] calldata _lifetimeRewards
  ) external onlyRewardsOracle {
      if (_vaultIds.length != _lifetimeRewards.length) revert InvalidParamLengths();

++      uint256 rewardsBefore;
++      for (uint256 i = 0; i < _vaultIds.length; ++i) {
++         rewardsBefore += vaults[_vaultIds[i]].getRewards();
++      }


--    int256 rewards = getDepositChange();
      for (uint256 i = 0; i < _vaultIds.length; ++i) {
          vaults[_vaultIds[i]].updateLifetimeRewards(_lifetimeRewards[i]);
      }

++      uint256 rewardsAfter;
++     for (uint256 i = 0; i < _vaultIds.length; ++i) {
++          rewardsAfter += vaults[_vaultIds[i]].getRewards();
++      }

++      uint256 rewards = rewardsAfter - rewardsBefore;
      if (rewards > (totalDeposits * maxRewardChangeBPS) / BASIS_POINTS)
          revert RewardsTooHigh();

      emit UpdateLifetimeRewards();
  }
```

Stake.link: Acknowledged.

7.2.5 `EspressoVault::exitIsWithdrawable` does not check if assets were already withdrawn

Description: If a given validator exits `EspressoStaking`, the delegators' funds become available for withdrawal after the lock period.

`EspressoVault` includes a `view` function, which is also utilized by `EspressoFundFlowController`, to determine whether the exited validator's funds are withdrawable. Currently, this function only checks the value of `validatorExits::unlockAt`.

contracts/espressoStaking/EspressoVault.sol#exitIsWithdrawable

```
    function exitIsWithdrawable() external view returns (bool) {
        uint256 unlocksAt = espressoStaking.validatorExits(validator);

        return unlocksAt != 0 && block.timestamp >= unlocksAt;
    }
```

In the `EspressoStaking` contract, the `validatorExits` mapping remains unchanged even after a withdrawal is claimed. Consequently, if the funds have already been withdrawn, this view function will continue to return `true`. However, any subsequent calls to `claimValidatorExit` will revert because the delegated amount has been reset to zero.

https://github.com/EspressoSystems/espresso-network/blob/main/contracts/src/StakeTableV2.sol#L461-L486

```
    function claimValidatorExit(address validator) public virtual override whenNotPaused {
        address delegator = msg.sender;
        uint256 unlocksAt = validatorExits[validator];
        if (unlocksAt == 0) {
            revert ValidatorNotExited();
        }

        if (block.timestamp < unlocksAt) {
            revert PrematureWithdrawal();
        }

        uint256 amount = delegations[validator][delegator];
        if (amount == 0) {
            revert NothingToWithdraw();
        }

        // Mark funds as spent
        delegations[validator][delegator] = 0;
        // the delegatedAmount is updated here (instead of during deregistration) in v2,
        // it's only decremented during withdrawal
        validators[validator].delegatedAmount -= amount;

        SafeTransferLib.safeTransfer(token, delegator, amount);

        emit ValidatorExitClaimed(delegator, validator, amount);
    }
```

Impact:

- The Stake.Link off-chain system will retrieve incorrect results, identifying vaults that have already been withdrawn as still being withdrawable.

Recommended Mitigation: The function should be updated to verify that a delegation balance still exists for the vault/delegator.

```
     function exitIsWithdrawable() external view returns (bool) {
         uint256 unlocksAt = espressoStaking.validatorExits(validator);
+        uint256 amount = espressoStaking.delegations(validator, address(this));

-        return unlocksAt != 0 && block.timestamp >= unlocksAt;
+        return unlocksAt != 0 && block.timestamp >= unlocksAt && amount > 0;
     }
```

Stake.Link: Fixed in e37f39f.

Cyfrin: Verified.

7.3 Informational

7.3.1 `FundFlowController::unbondVaults` can be permanently DoS'ed for Espresso Staking Pool with multiple strategies

Description: In the Staking pool targeting the `ESP` token, the `StakingPool` is designed to support more than one strategy.

contracts/core/StakingPool.sol

```
contract StakingPool is StakingRewardsPool {
    ...

    // list of all strategies controlled by pool
    address[] private strategies;
    ...
}
```

When a withdrawal is initiated, it is queued in the `WithdrawalPool`. This process occurs when requesting a withdrawal of staked tokens to reclaim the original asset. This is triggered by calling `PriorityPool::_withdraw` (assuming no queued deposits are present and instant withdrawal is not activated).

contracts/core/priorityPool/PriorityPool.sol::_withdraw

```
    function _withdraw( ... ) internal returns (uint256) {
        ...
        if (totalQueued != 0) { ... }

        if (
            toWithdraw != 0 &&
            allowInstantWithdrawals &&
            withdrawalPool.getTotalQueuedWithdrawals() == 0
        ) { ... }

        if (toWithdraw != 0) {
            if (!_shouldQueueWithdrawal) revert InsufficientLiquidity();

            if (toWithdraw >= withdrawalPool.minWithdrawalAmount()) {
>>              withdrawalPool.queueWithdrawal(_account, toWithdraw);
                queued = toWithdraw;
            } else {
                IERC20Upgradeable(address(stakingPool)).safeTransfer(_account, toWithdraw);
            }
        }
        ...
    }
```

When executing a queued withdrawal from the `PriorityPool`, `StakingPool::withdraw` is called. This function withdraws liquidity from different strategies to finalize queued withdrawals in the `WithdrawalPool`.

contracts/core/priorityPool/PriorityPool.sol#executeQueuedWithdrawals

```
    function executeQueuedWithdrawals( ... ) external onlyWithdrawalPool {
        IERC20Upgradeable(address(stakingPool)).safeTransferFrom( ... );
>>      stakingPool.withdraw(address(this), address(this), _amount, _data);
        token.safeTransfer(msg.sender, _amount);
    }
```

If insufficient main tokens are available in the contract, liquidity is withdrawn from the strategies. Since the `StakingPool` can have multiple strategies, it iterates through them to fulfill the queued withdrawal amount.

contracts/core/StakingPool.sol#withdraw

```
    function withdraw( ... ) external onlyPriorityPool {
```

```
        uint256 toWithdraw = _amount;
        if (_amount == type(uint256).max) {
            toWithdraw = balanceOf(_account);
        }

        uint256 balance = token.balanceOf(address(this));
        if (toWithdraw > balance) {
>>          _withdrawLiquidity(toWithdraw - balance, _data);
        }
        require(
            token.balanceOf(address(this)) >= toWithdraw,
            "Not enough liquidity available to withdraw"
        );

        _burn(_account, toWithdraw);
        totalStaked -= toWithdraw;
        token.safeTransfer(_receiver, toWithdraw);
    }
// --------
    function _withdrawLiquidity(uint256 _amount, bytes[] calldata _data) private {
        ...
        for (uint256 i = strategies.length; i > 0; i--) {
            ...
            if (strategyCanWithdrawdraw >= toWithdraw) {
>>              strategy.withdraw(toWithdraw, strategyData);
                break;
            } else if (strategyCanWithdrawdraw > 0) {
>>              strategy.withdraw(strategyCanWithdrawdraw, strategyData);
                ...
            }
        }
    }
```

To withdraw from the `EspressoStrategy`, the contract must have enough `ESP` tokens. This only occurs during the unbonding process. However, when unbonding from a single strategy, the current logic forces the unbonding of the **total global queued withdrawal amount from that specific strategy**.

contracts/espressoStaking/EspressoFundFlowController.sol#unbondVaults

```
    function unbondVaults() external {
>>      uint256 queuedWithdrawals = withdrawalPool.getTotalQueuedWithdrawals();
        uint256 queuedDeposits = strategy.totalQueued();

        if ( ... ) revert NoUnbondingNeeded();

        uint256 toWithdraw = queuedWithdrawals - queuedDeposits;
>>      strategy.unbond(toWithdraw);
        timeOfLastUnbond = uint64(block.timestamp);
    }
```

If there is more than one strategy and the total queued withdrawal amount exceeds the deposits of any single strategy, calling `EspressoFundFlowController#unbondVaults` will always revert. This is because an individual strategy cannot fulfill the global withdrawal requirement alone.

contracts/espressoStaking/EspressoStrategy.sol#unbond

```
    function unbond(uint256 _toUnbond) external onlyFundFlowController {
        ...
        while (toUnbondRemaining != 0) {
            ...
            if (i == vaultWithdrawalIndex) break;
        }
```

```
>>      if (toUnbondRemaining > 0) revert InsufficientDeposits();

        vaultWithdrawalIndex = i;
        numVaultsUnbonding = numVaultsUnbonded;

        emit Unbond(_toUnbond);
    }
```

Impact:

- Users are unable to unbond assets from `EspressoStrategy` (requesting token withdrawals) if the total queued withdrawal amount exceeds the total deposits of the specific strategy being processed.

Proof of Concept:

- The Espresso Staking Pool utilizes two strategies: `Strategy A` and `Strategy B`.
- Each strategy has `10,000` tokens deposited.
- No new deposits are made and instant withdrawals are disabled.
- Users request withdrawals; the `WithdrawalPool` total queue reaches `15,000` tokens.
- Users attempt to unbond assets from `Strategy A` via `EspressoFundFlowController::unbondVaults`.
- The call calculates toWithdraw = `15,000`.
- The call reverts with `InsufficientDeposits` because Strategy A only holds 10,000 tokens, even though the total system liquidity (20,000) is sufficient.

Recommended Mitigation: Introduce a function in `EspressoStrategy` to track available deposited assets by iterating through all Vaults:

- For active Vaults, use `getPrincipalDeposits`.
- For inactive Vaults, use `getTotalDeposits`.

The requested unbonding amount should be capped by the strategy's individual total deposits. This allows one strategy to be fully unbonded while other strategies handle the remaining balance, collectively fulfilling the users' withdrawal requests.

Stake.Link: Acknowledged.

7.3.2 Changing `vaultImplementation` can DoS `EspressoStrategy::addVault` function

Description In `EspressoStrategy`, the Owner can deploy new vaults by calling `addVault`. This function creates an `ERC1967Proxy` pointing to the current `vaultImplementation`.

The deployment process invokes the `initialize` function using a hardcoded signature with exactly five parameters.

contracts/espressoStaking/EspressoStrategy.sol#addVault

```
    function addVault(address _validator) external onlyOwner {
        address vault = address(
            new ERC1967Proxy(
                vaultImplementation,
                abi.encodeWithSignature(
                    "initialize(address,address,address,address,address)",
                    address(token),
                    address(this),
                    address(espressoStaking),
                    address(espressoRewards),
                    _validator
                )
            )
        );
```

```
        token.safeApprove(vault, type(uint256).max);
        vaults.push(IEspressoVault(vault));

        emit AddVault(_validator);
    }
// -------
    function setVaultImplementation(address _vaultImplementation) external onlyOwner {
        if (_vaultImplementation == address(0)) revert InvalidAddress();
        vaultImplementation = _vaultImplementation;
        emit SetVaultImplementation(_vaultImplementation);
    }
```

The `vaultImplementation` can be updated by the Admin at any time. Furthermore, the `upgradeVaults` function allows existing vaults to be upgraded to the new implementation, supporting arbitrary data for the upgrade call.

contracts/espressoStaking/EspressoStrategy.sol#upgradeVaults

```
    function upgradeVaults(address[] calldata _vaults, bytes[] memory _data) external onlyOwner {
        for (uint256 i = 0; i < _vaults.length; ++i) {
            if (_data.length == 0 || _data[i].length == 0) {
                IEspressoVault(_vaults[i]).upgradeTo(vaultImplementation);
            } else {
                IEspressoVault(_vaults[i]).upgradeToAndCall(vaultImplementation, _data[i]);
            }
        }
        emit UpgradedVaults(_vaults);
    }
```

If a new `vaultImplementation` requires a different `initialize` signature (e.g., more/fewer parameters or changed data types), the `addVault` function becomes broken. Because `addVault` enforces the legacy 5-parameter signature, calls to it will either revert due to a signature mismatch or initialize the contract with incorrect data.

Impact

- Changing the `vaultImplementation` to a version with a different initialization schema will permanently DoS the `addVault` function.
- If the signature matches but parameter type changes, vaults may be deployed in an inconsistent or broken state.

Proof of Concept

1. Admin updates `vaultImplementation` to a new version that requires an additional parameter (e.g., `initialize(address,address,address,address,address,uint256)`).
2. Admin attempts to call `addVault`.
3. The transaction reverts because the `ERC1967Proxy` constructor fails to execute the hardcoded 5-parameter `initialize` call on the new implementation.

Recommended Mitigation

Introduce an overloaded `addVault` function (or update the existing one) to accept arbitrary initialization data. This aligns the deployment logic with the flexible upgrade logic found in `upgradeVaults`.

Stake.Link: Acknowledged.