Stake.Link Withdrawals Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Hans

Assisting Auditors

September 17, 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 audit looked into the vault based staking infrastructure centered around Chainlink staking. A system of smart contracts are designed to manage staking, rewards, and fund flows for both community and operator vaults. At its core, the protocol utilizes two primary strategies: the Community Vault Control Strategy (CommunityVCS) and the Operator Vault Control Strategy (OperatorVCS).

These strategies manage multiple vaults that interact with Chainlink's staking contracts. The CommunityVCS handles community stakers, while the OperatorVCS manages operators. Both strategies are responsible for:

- Depositing funds into individual vaults, which then stake in Chainlink's contracts
- Managing the creation and allocation of new vaults
- Handling reward accrual and distribution from Chainlink staking

Rewards generated from Chainlink staking are crucial to the system's incentive structure. The process works as follows:

- Rewards accumulate in the individual vaults managed by CommunityVCS and OperatorVCS.
- Periodically, the `updateStrategyRewards` function is called on the Staking Pool.
- This function, in turn, calls `updateDeposits` on each strategy (CommunityVCS and OperatorVCS).
- The strategies calculate the change in deposits (which includes accrued rewards from Chainlink staking) and return this information to the Staking Pool.
- The Staking Pool then updates its 'totalStaked" value, effectively distributing the rewards among all stakers.

This mechanism ensures that rewards from Chainlink staking are fairly distributed to all participants in the stake.link system, proportional to their stake.

While both the Community and Operator controlled strategies are already deployed on-chain, this audit focused on new contract upgrades centered around supporting withdrawals for stakers. The order of vault withdrawal and the slashing and reward accounting were some of the key areas of focus in this audit.

5 Audit Scope

The audit scope was limited to Solidity contracts that govern the core on-chain operations of stake.link staking infrastructure. This audit focussed on the impact of adding withdrawal support to existing vault strategies. Specifically we looked at the changes made in the PR and their impact to the existing contracts deployed on-chain.

The following contracts were included in the scope of the audit:

```
linkStaking/base/Vault.sol
linkStaking/base/VaultControllerStrategy.sol
linkStaking/CommunityVault.sol
linkStaking/CommunityVCS.sol
linkStaking/OperatorVault.sol
linkStaking/OperatorVCS.sol
linkStaking/FundFlowController.sol
linkStaking/PPKeeper.sol
metisStaking/SequencerVCS.sol
core/priorityPool/PriorityPool.sol
core/priorityPool/WithdrawalPool.sol
core/base/Strategy.sol
core/base/StakingRewardsPool.sol


src/interfaces/IVault.sol
src/interfaces/IStrategy.sol
src/interfaces/IStaking.sol
src/interfaces/IRewardVault.sol
src/interfaces/IStakingPool.sol
src/interfaces/IPriorityPool.sol
src/interfaces/IWithdrawalPool.sol
```

This scope encompasses the core contracts responsible for staking strategies specific to Chainlink, vault management, reward distribution, and overall fund flow control.It includes base contracts, specific implementations for community and operator staking, pool management, and auxiliary components such as the fund flow controller.

6 Executive Summary

Over the course of 15 days, the Cyfrin team conducted an audit on the Stake.Link Withdrawals smart contracts provided by STAKE.LINK. In this period, a total of 21 issues were found.

Stake.link has developed a sophisticated staking protocol that allows users to participate in Chainlink staking through a managed system of vaults and strategies. The protocol incorporates separate strategies for community stakers and operators, along with mechanisms for managing deposits, withdrawals, and reward distribution.

This audit focused on new contract upgrades designed to support withdrawals for stakers, examining the order of vault withdrawals, fund flow controller operations, and share-based accounting related to staking rewards and slashing conditions.

The audit revealed several significant findings across various risk categories:

Critical risk issues that could result in permanent loss of funds include:

- A storage gap discrepancy during upgrades could cause storage collision in vault controller strategies.
- An implicit assumption of sequential filling of community vaults that could cause permanent loss to stakers.

High risk issues include:

- Lack of access control in FundController's performUpkeep function could allow an attacker to block withdrawals.

- A call selector mismatch could cause vault unstaking to revert.
- Incorrect handling of deposit data in FundFlowController could lead to system inconsistencies.
- An attacker could potentially inflate the total deposit room in vault groups to near infinity.
- A flash loan attack could potentially create an unmanageable number of ghost vaults in CommunityVCS.

In addition, the audit discovered multiple medium/low risk issues.

All critical, high, and medium-risk issues identified in the audit have been successfully mitigated.

The auditors noted that the codebase demonstrates very good test coverage with well-written tests that cover many edge case scenarios. This extensive testing framework significantly contributes to the overall security and reliability of the protocol.

Given the discovery of two critical issues by two auditors in a time-boxed audit, it is recommended that this private audit be followed up with an open audit contest. This would leverage the insights of hundreds of leading security researchers, potentially uncovering additional vulnerabilities and further strengthening the protocol's security.

Summary

Project Name	Stake.Link Withdrawals
Repository	contracts
Commit	aa0efea6a93c. . .
Audit Timeline	July 29th - August 16th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	2
High Risk	5
Medium Risk	6
Low Risk	5
Informational	0
Gas Optimizations	3
Total Issues	21

Summary of Findings

[C-1] Implicit assumption of sequentially filling of vaults in `CommunityVCS::getDepositChange` causes direct losses to stakers	Resolved
[C-2] Storage gap discrepancy during upgrade causes storage collision in vault controller strategies	Resolved
[H-1] Lack of access control in `FundController::performUpkeep` can allow an attacker to block withdrawals	Resolved

[H-2] Attacker can potentially inflate total deposit room in vault group to near infinity	Resolved
[H-3] Incorrect Handling of Deposit Data in FundFlowController	Resolved
[H-4] Call selector mismatch will cause unstaking to revert	Resolved
[H-5] Flash loan attack can potentially create an unmanageable number of ghost vaults in CommunityVCS	Resolved
[M-1] `WithdrawalPool::withdrawalBatches` array is only every increasing in size potentially leading to Denial Of Service	Resolved
[M-2] Attacker can deny users from depositing into priority pool by front-running call to StakingPool::depositLiquidity()	Resolved
[M-3] No way to recover principal if an operator is removed by Chainlink	Resolved
[M-4] `Vault::unbondingActive` can be out of sync with Chainlink and `Fund-FlowController` bonding	Resolved
[M-5] Potential stale data in `FundFlowController::performUpkeep` can lead to incorrect state updates	Resolved
[M-6] Staking rewards susceptible to reward multiplier manipulation	Resolved
[L-1] Lack of validation checks in `PriorityPool::depositQueuedTokens` can trigger temporary delays in queued deposits	Acknowledged
[L-2] FundflowController goes out-of-sync when Chainlink changes their un-bonding and claim periods	Acknowledged
[L-3] Chainlink can have different bonding and claim periods for Operator and Community staking pools	Acknowledged
[L-4] User can frontrun operator slashing/update rewards by withdrawing from the pool	Acknowledged
[L-5] Withdrawals when Chainlink pool is paused will take longer than neces-sary	Acknowledged
[G-1] Gas optimisation of `FundFlowController::_getVaultDepositOrder`	Acknowledged
[G-2] Cache storage reads in `FundFlowController::performUpkeep`	Acknowledged
[G-3] Optimize VaultGroup and Fee Structs Using Bitwise Operations	Acknowledged

7 Findings

7.1 Critical Risk

7.1.1 Implicit assumption of sequentially filling of vaults in `CommunityVCS::getDepositChange` causes direct losses to stakers

Description: The `getDepositChange` function in the CommunityVCS contract can lead to significant underreporting of the total balance in the protocol. The function is designed with an assumption that vaults are filled sequentially, but this assumption can easily be violated with withdrawals.

Vault ID's in a given `curUnbondedVaultGroup` are not sequential but separated by the `numVaultGroups` which is set to 5 for Community Vault Controller Strategy. If there is a full withdrawal at any index, the `getDepositChange` will stop including the deposits for all subsequent vaults.

The vulnerable code in CommunityVCS:

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

This function stops counting at the first empty vault it encounters. When StakingPool calls `updateDeposits`, the incorrect change is then used to update the `totalStaked` value. This causes an accounting loss to the users as the `totalStaked` would artificially reduce, similar to what happens during slashing.

Impact: Total deposits of non-sequential vaults are not counted towards `totalBalance` if there is an empty vault with a lower index. This has a similar effect of slashing. Key effects are:

- `TotalStaked` on Staking Pool drops causing a direct loss to stakers
- Fee receivers do not accrue a fee

Proof of Concept: Below POC shows that if the first vault is fully withdrawn, `getDepositChange` stops including the vault deposits for all the subsequent vaults.

```
it(`sequential deposits assumption in Community VCS can be attacked`, async () => {
    const {
      accounts,
      adrs,
      token,
      rewardsController,
      stakingController,
      strategy,
      stakingPool,
      updateVaultGroups,
    } = await deployCommunityVaultFixture()

    await strategy.deposit(toEther(1200), encodeVaults([]))

    await updateVaultGroups([0, 5, 10], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([1, 6, 11], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([2, 7], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([3, 8], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([4, 9], 300)
    await time.increase(claimPeriod)
    await updateVaultGroups([0, 5, 10], 300)

    //@audit withdraw from the first vault
    await strategy.withdraw(toEther(100), encodeVaults([1]))

    let vaults = await strategy.getVaults()

    let totalDeposits = 0

    for (let i = 0; i < 15; i++) {
      let vault = await ethers.getContractAt('CommunityVault', vaults[i])
      totalDeposits += fromEther(await vault.getTotalDeposits())
    }

    console.log(`total deposits`, totalDeposits)
    assert.equal(totalDeposits, 1100) //@audit 1200 (deposit) - 100 (withdrawn)

    // const strategyTokenBalance = fromEther(await token.balanceOf(adrs.strategy))
    // console.log(`strategy token balance`, totalDepositsCalculated)
    const depositChange = fromEther(await strategy.getDepositChange())

    assert.equal(depositChange, -1000) //@audit This is equivalent to slashing
    //@audit -> deposit Change should be 0 at this point, instead it is -1000
    //@audit only the 0'th vault deposit is considered in the deposit change calculation
  })
```

Recommended Mitigation: Consider using the base implementation for `getDepositChange`. With the current withdrawal logic, a sequential deposit assumption can be easily violated.

Stake.link Fixed in e349a7d

Cyfrin Verified. `getDepositChange` override removed in `CommunityVCS`.

7.1.2 Storage gap discrepancy during upgrade causes storage collision in vault controller strategies

Description: Both `CommunityVCS` and `OperatorVCS` are upgradeable contracts that are already deployed on mainnet.

The contracts audited are intended to upgrade these. There is however an issue with the storage gap in the inherited contract `VaultControllerStrategy`:

The `VaultControllerStrategy` used as base in the deployed contracts have its storage setup like this:

```
    IStaking public stakeController;        // 1
    Fee[] internal fees;                    // 2

    address public vaultImplementation;     // 3

    IVault[] internal vaults;               // 4
    uint256 internal totalDeposits;         // 5
    uint256 public totalPrincipalDeposits;  // 6
    uint256 public indexOfLastFullVault;    // 7

    uint256 public maxDepositSizeBP;        // 8

    uint256[9] private __gap;               // 8 + 9 = 17
```

And in the upgraded code from `VaultControllerStrategy`:

```
    IStaking public stakeController;                 // 1
    Fee[] internal fees;                             // 2

    address public vaultImplementation;              // 3

    IVault[] internal vaults;                        // 4
    uint256 internal totalDeposits;                  // 5
    uint256 public totalPrincipalDeposits;           // 6

    uint256 public maxDepositSizeBP;                 // 7

    IFundFlowController public fundFlowController; // 8
    uint256 internal totalUnbonded;                  // 9

    VaultGroup[] public vaultGroups;                 // 10
    GlobalVaultState public globalVaultState;        // 11
    uint256 internal vaultMaxDeposits;               // 12

    uint256[6] private __gap;                        // 12 + 6 = 18!
```

There are four new slots added, `totalUnbonded`, `vaultGroups`, `globalVaultState` and `vaultMaxDeposits` however the gap is only decreased by 3 (`9` -> `6`). Hence the total storage slots used by the upgraded `VaultControllerStrategy` will increase by `1` encroaching on the storage of the `Community` and `OperatorVCS` implenentations.

Note, there is also some variables that have been moved and renamed (`maxDepositSizeBP` and `indexOfLastFullVault`) but that is handled in the `initializer`

Impact: For `CommunityVCS` the impact is quite moderate. This is the storage `CommunityVCS`:

```
contract CommunityVCS is VaultControllerStrategy {
    uint128 public vaultDeploymentThreshold;
    uint128 public vaultDeploymentAmount;
```

The extra gap will lead to that `vaultDeploymentThreshold` will get the value of `vaultDeploymentAmount` (`6` on chain at time of writing). And `vaultDeploymentAmount` will be `0`.

These are used in `CommunityVCS::performUpkeep`:

```
    function performUpkeep(bytes calldata) external {
        if ((vaults.length - globalVaultState.depositIndex) >= vaultDeploymentThreshold)
            revert VaultsAboveThreshold();
        _deployVaults(vaultDeploymentAmount);
    }
```

Since `vaultDeploymentAmount` is `0` no new vaults will be deployed until the issue is discovered and the values updated (using `setVaultDeploymentParams`).

This will at most cause deposits to be blocked for a while since no new vaults will be deployed.

For `OperatorVCS` however, the impact is more severe: Here's the storage layout of `OperatorVCS`:

```
contract OperatorVCS is VaultControllerStrategy {
    using SafeERC20Upgradeable for IERC20Upgradeable;

    uint256 public operatorRewardPercentage;
    uint256 private unclaimedOperatorRewards;
```

`operatorRewardPercentage` will get the value of `unclaimedOperatorRewards` (`4914838471043033862842`, `4.9e21` at time or writing).

This is used in `OperatorVault` to calculate the portion of the rewards earned that should go to the operator:

```
            opRewards =
                (uint256(depositChange) *
                    IOperatorVCS(vaultController).operatorRewardPercentage()) /
                10000;
```

This is ultimately triggered by calling `StakingPool::updateStrategyRewards` that is in-turn called periodically by the protocol. If `updateStrategyRewards` (which in turn calls `OperatorVault::updateDeposits`) is called before the storage collision is discovered, the `OperatorVaults` will be handing out an enormous amount of reward shares to the operators. This will greatly diminish the value of each share held by anyone else and any operator who withdraws these can take a lot of `LINK` from the pool.

`StakingPool::updateStrategyRewards` is also callable by anyone hence a malicious operator could figure this out and backrun the upgrade by calling this themselves.

Recommended Mitigation: Considering reducing the storage gap in the upgraded `VaultControllerStrategy` to `5`

Stake.link Fixed in 3107a31

Cyfrin Verified. Storage gap correctly assigned.

7.2 High Risk

7.2.1 Lack of access control in `FundController::performUpkeep` can allow an attacker to block withdrawals

Description: Anyone can perform upkeep on the `FundFlowController`. This will rotate the unbonded vault groups in the vault controllers through `VaultControllerStrategy::updateVaultGroups`.

The user calling `FundFlowController::performUpkeep` also provides the details to `VaultControllerStrategy::updateVaultGroups`, the vaults to unbond in the Chainlink staking pool and the new `totalUnbonded` amount:

```
    function updateVaultGroups(
        uint256[] calldata _curGroupVaultsToUnbond,
        uint256 _nextGroup,
        uint256 _nextGroupTotalUnbonded
    ) external onlyFundFlowController {
        for (uint256 i = 0; i < _curGroupVaultsToUnbond.length; ++i) {
            vaults[_curGroupVaultsToUnbond[i]].unbond();
        }

        globalVaultState.curUnbondedVaultGroup = uint64(_nextGroup);
        totalUnbonded = _nextGroupTotalUnbonded;
    }
```

None of these two parameters are validated. Hence a user could provide any vaults to unbond or any new `totalUnbonded` amount.

Impact: Providing `0` as `_nextGroupTotalUnbonded` would effectively block withdrawals because of this check in `VaultControllerStrategy::withdraw`:

```
    function withdraw(uint256 _amount, bytes calldata _data) external onlyStakingPool {
        if (!fundFlowController.claimPeriodActive() || _amount > totalUnbonded)
            revert InsufficientTokensUnbonded();
```

Since `FundFlowController::performUpkeep` can only be called at certain times (when new vaults are available to be unbonded) this could block withdrawals until it could be called the next time. At which time the attacker could call this again to continue to block withdrawals.

Setting `totalUnbonded` to `0`, although less impactful, would also break depositing into vaults with active unbonding in `_depositToVaults`:

```
                if (vault.unbondingActive()) {
                    totalRebonded += deposits;
                }
...

        if (totalRebonded != 0) totalUnbonded -= totalRebonded;
```

Providing erroneous vaults could be used to reorder which vaults are available to withdraw from causing disruptions in withdrawals. Since `FundFlowController` and `VaultControllerStrategy` assumes that all vaults with deposits in the group are available when the group is active.

Proof of Concept: Two tests, one for `totalUnbonded` and one for the unbonded vaults, that can be added to `vault-controller-strategy.test.ts`:

```
  it('performUpkeep, updateVaultGroups can be called by anyone and lacks validation for totalUnbonded',
  ↪  async () => {
    const { adrs, strategy, token, stakingController, vaults, updateVaultGroups } =
      await loadFixture(deployFixture)
    // Deposit into vaults
    await strategy.deposit(toEther(500), encodeVaults([]))
    assert.equal(fromEther(await token.balanceOf(adrs.stakingController)), 500)
    for (let i = 0; i < 5; i++) {
      assert.equal(fromEther(await stakingController.getStakerPrincipal(vaults[i])), 100)
    }
    assert.equal(Number((await strategy.globalVaultState())[3]), 5)

    // do the initial rotation to get a vault into claim period
    await updateVaultGroups([0, 5, 10], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([1, 6, 11], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([2, 7], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([3, 8], 0)
    await time.increase(claimPeriod)

    // user calls `fundFlowController.performUpkeep` with erroneous `totalUnbonded`
    await updateVaultGroups([4, 9], 0)

    // causing withdrawals to stop working
    await expect(
      strategy.withdraw(toEther(50), encodeVaults([0, 5]))
    ).to.be.revertedWithCustomError(strategy, 'InsufficientTokensUnbonded()')
  })

  it('performUpkeep, updateVaultGroups can be called by anyone and lacks validation for unbonded
  ↪  vaults', async () => {
    const { adrs, strategy, token, stakingController, vaults, updateVaultGroups } =
      await loadFixture(deployFixture)

    // Deposit into vaults
    await strategy.deposit(toEther(500), encodeVaults([]))
    assert.equal(fromEther(await token.balanceOf(adrs.stakingController)), 500)
    for (let i = 0; i < 5; i++) {
      assert.equal(fromEther(await stakingController.getStakerPrincipal(vaults[i])), 100)
    }
    assert.equal(Number((await strategy.globalVaultState())[3]), 5)

    // calling `fundFlowController.performUpkeep` with vaults in other groups
    await updateVaultGroups([0, 1, 2, 3, 4], 0)
    await time.increase(claimPeriod)
    await expect(
      updateVaultGroups([1, 6, 11], 0)
    ).to.be.revertedWithCustomError(stakingController, 'UnbondingPeriodActive()')
  })
```

Recommended Mitigation: Consider not allowing the user to provide any vaults or amounts at all. `FundFlowController` has a `view` function `checkUpkeep` which collects all this data.

We recommend combining `checkUpkeep` and `performUpkeep` to be linked so that the caller won't provide any data. This would also address issue 7.3.6 *Potential stale data in `FundFlowController::performUpkeep` can lead to incorrect state updates*

Stake.link Fixed in 53d6090

Cyfrin Verified. `performUpkeep` no longer takes encoded inputs.

7.2.2 Attacker can potentially inflate total deposit room in vault group to near infinity

Description: The `VaultControllerStrategy::deposit` function increases `totalDepositRoom` based on the difference between `maxDeposits` (from Chainlink) and `vaultMaxDeposits`, but fails to update `vaultMaxDeposits` afterwards. As a result, on every deposit, vault group deposit room increases by `maxDeposits - vaultMaxDeposits`, regardless of the actual deposit amount.

Furthermore, the `StakingPool` contract, which interacts with `VaultControllerStrategy`, has a public `depositLiquidity` function that can be called by any user. This function uses the current balance of the `StakingPool` contract for deposits, which can be manipulated by transferring small amounts of tokens directly to the contract.

The combination of these issues could allow a malicious actor to:

- Manipulate the deposit process by calling depositLiquidity in StakingPool with carefully crafted data after transferring a small amount of tokens to the contract.
- Artificially inflate `totalDepositRoom` in VaultControllerStrategy by repeatedly triggering the deposit function.

Note that this attack vector is possible whenever Chainlink increases its vault limits.

Impact: Vault group total deposit room can be increased by attacker to near infinity by repeatedly calling deposits. This has 2 severe side-effects

- Unused vaults will never see any deposits even when vaults with ids less than deposit index are full
- Since withdrawals increase deposit room continuously, it is likely that a repeated attack, in the extreme case, can cause an deposit room overflow and thus block all withdrawals.

Proof of Concept:

```
it('inflate deposit room in vault groups to infinity', async () => {
  const {
    token,
    accounts,
    signers,
    adrs,
    strategy,
    priorityPool,
    stakingPool,
    stakingController,
    rewardsController,
    vaults,
  } = await loadFixture(deployFixture)

  const [, totalDepRoomBefore] = await strategy.vaultGroups(1)
  await stakingController.setDepositLimits(toEther(10), toEther(120))
  await stakingPool.deposit(accounts[0], 1 /*dust amount*/, [encodeVaults([0, 1, 2, 3, 4])])
  const [, totalDepRoomAfterFirstDeposit] = await strategy.vaultGroups(0)

  //@audit after first deposit, deposit room increases to 20 ether
  assert.equal(fromEther(totalDepRoomAfterFirstDeposit), 20)
  await stakingPool.deposit(accounts[0], 1 /*dust amount*/, [encodeVaults([4])])

  //@audit after first deposit, deposit room increases to 40 ether
  const [, totalDepRoomAfterSecondDeposit] = await strategy.vaultGroups(0)
  assert.equal(fromEther(totalDepRoomAfterSecondDeposit), 40)

  //@audit this can theoretically increase to infinity - disrupting the deposit logic
})
```

Recommended Mitigation: Update vaultMaxDeposits once all vault group total deposit rooms are updated.

Stake.link Fixed in 04c2f10

Cyfrin Verified. `vaultMaxDeposits` is correctly updated.

7.2.3 Incorrect Handling of Deposit Data in FundFlowController

Description: The `getDepositData` function in the `FundFlowController` contract incorrectly handles the case when operator vaults can accommodate all deposits. This leads to empty deposit data being passed to operator vaults, potentially causing deposits to fail or be misallocated.

Relevant code snippet from `FundFlowController`:

```
function getDepositData(uint256 _toDeposit) external view returns (bytes[] memory) {
    uint256 toDeposit = 2 * _toDeposit;
    bytes[] memory depositData = new bytes[](2);

    (
        uint64[] memory opVaultDepositOrder,
        uint256 opVaultsTotalToDeposit
    ) = _getVaultDepositOrder(operatorVCS, toDeposit);
    depositData[0] = abi.encode(opVaultDepositOrder);

    if (opVaultsTotalToDeposit < toDeposit) {
        (uint64[] memory comVaultDepositOrder, ) = _getVaultDepositOrder(
            communityVCS,
            toDeposit - opVaultsTotalToDeposit
        );
        depositData[1] = abi.encode(comVaultDepositOrder);
    } else {
        depositData[0] = abi.encode(new uint64[](0)); // @audit: Should be depositData[1]
    }

    return depositData;
}
```

Note that the depositData[0] that contains the vault ID order is deleted when operator vaults can handle full deposit. This treatment is also inconsistent with the `getWithdrawalData` function that correctly encodes only `withdrawal-Data[0]` in both the `if-else` logic flow.

```
function getWithdrawalData(uint256 _toWithdraw) external view returns (bytes[] memory) {
    uint256 toWithdraw = 2 * _toWithdraw;
    bytes[] memory withdrawalData = new bytes[](2);

    (
        uint64[] memory comVaultWithdrawalOrder,
        uint256 comVaultsTotalToWithdraw
    ) = _getVaultWithdrawalOrder(communityVCS, toWithdraw);
    withdrawalData[1] = abi.encode(comVaultWithdrawalOrder);

    if (comVaultsTotalToWithdraw < toWithdraw) {
        (uint64[] memory opVaultWithdrawalOrder, ) = _getVaultWithdrawalOrder(
            operatorVCS,
            toWithdraw - comVaultsTotalToWithdraw
        );
        withdrawalData[0] = abi.encode(opVaultWithdrawalOrder);
    } else {
        withdrawalData[0] = abi.encode(new uint64[](0)); //@audit correctly encoding this instead of
        ↪  withdrawalData[1]
    }

    return withdrawalData;
}
```

Impact: `getDepositData` is called by the `PPKeeper` contract to compute the vaultId's that are passed to the priority pool to deposit queued tokens. This bug can cause deposits to fail or be misallocated when operator vaults have sufficient capacity to handle all deposits.

Recommended Mitigation: Consider modifying the `getDepositData` function to correctly handle the case when operator vaults can accommodate all deposits. The corrected code should be:

```
if (opVaultsTotalToDeposit < toDeposit) {
    (uint64[] memory comVaultDepositOrder, ) = _getVaultDepositOrder(
        communityVCS,
        toDeposit - opVaultsTotalToDeposit
    );
    depositData[1] = abi.encode(comVaultDepositOrder);
} else {
-    depositData[0] = abi.encode(new uint64[](0));
+   depositData[1] = abi.encode(new uint64[](0));
}
```

Stake.link Fixed in 7af3ec0

Cyfrin Verified. Fixed as recommended.

7.2.4 Call selector mismatch will cause unstaking to revert

Description: In the Chainlink staking the `StakingPoolBase::unstake` call looks like this:

```
  function unstake(uint256 amount) external {
```

However in `Vault::withdraw` it is called like this:

```
        stakeController.unstake(_amount, false);
```

Impact: Unstaking will not work until all vaults are updated.

Recommended Mitigation: Consider removing the `false` from `unstake`:

```
-       stakeController.unstake(_amount, false);
+       stakeController.unstake(_amount);
```

Stake.link Fixed in c563a62

Cyfrin Verified. Fixed as recommended.

7.2.5 Flash loan attack can potentially create an unmanageable number of ghost vaults in CommunityVCS

Description: The current implementation of the `CommunityVCS` allows for a potential attack vector where an attacker can artificially inflate the number of vaults to an arbitrarily large number. This attack exploits the fact that `globalState.depositIndex` is monotonically increasing and is not reset when earlier vaults become empty. As a result, each iteration of this attack can push the `depositIndex` further, leading to the creation of ghost vaults which will never be used.

Consider following attack vector:

1. Obtain a flash loan for a large amount of tokens. Choose a token amount such that they trigger the `vaultDeploymentThreshold` condition

2. Deposit these tokens into the Priority Pool without specifying vault IDs.
3. This triggers a deposit into the Staking Pool, which in turn calls the deposit function of the CommunityVCS.
4. Since no vault IDs are provided, the deposit starts from `globalState.depositIndex` and continues until the deposit is fully allocated or all vaults are filled. This updates `globalState.depositIndex` to a bigger number
5. Call performUpkeep on CommunityVCS, which deploys new vaults.
6. Repeat steps 1-5
7. Assuming there is enough exit liquidity from all strategies in StakingPool, attacker can withdraw all deposited tokens to repay the flash loan

Impact: Managing an unnecessarily large number of vaults severely complicates protocol operations and maintenance. Critical functions like `updateDeposits`, which are essential for maintaining accurate vault accounting, become extremely expensive to execute. This is because the `getDepositChange` function, which tracks the latest total deposits across all vaults, must loop over every available vault. In extreme cases, this could lead to a denial-of-service condition for functions that need to iterate over all vaults

Proof of Concept:

```
it(`flash loan attack to max out community vaults`, async () => {
  const {
    token,
    accounts,
    signers,
    adrs,
    strategy,
    priorityPool,
    stakingPool,
    stakingController,
    rewardsController,
    updateVaultGroups,
  } = await loadFixture(deployCommunityVaultFixtureWithStrategyMock)
  //@note this fixture uses a strategy mock to simulate a strategy with a max deposit of 900 tokens
  //@note by now, that strategy is already full -> so any further deposits go into the Community VCS

  console.log(`strategies length ${(await stakingPool.getStrategies()).length}`)
  assert.equal((await strategy.getVaults()).length, 20)
  await stakingPool.deposit(accounts[0], toEther(1500), [encodeVaults([]), encodeVaults([])])

  // get initial deposit index
  const [, , , depositIndex] = await strategy.globalVaultState()
  console.log(`deposit index`, depositIndex.toString())

  await strategy.performUpkeep(encodeVaults([]))
  assert.equal((await strategy.getVaults()).length, 40) //@audit 20 new vaults are created

  await updateVaultGroups([0, 5, 10], 0)
  await time.increase(claimPeriod)
  await updateVaultGroups([1, 6, 11], 0)
  await time.increase(claimPeriod)
  await updateVaultGroups([2, 7], 0)
  await time.increase(claimPeriod)
  await updateVaultGroups([3, 8], 0)
  await time.increase(claimPeriod)
  await updateVaultGroups([4, 9], 300)

  //@note a user can flashloan 1100 tokens -> deposit them with no vaultIds -> deposit index moves to
  ↪  26
  await stakingPool.setPriorityPool(adrs.priorityPool)
  await token.approve(adrs.strategy, ethers.MaxUint256)
  await token.connect(signers[2]).approve(priorityPool.target, ethers.MaxUint256)
```

```
    await priorityPool
      .connect(signers[2])
      .deposit(toEther(1100), false, [encodeVaults([]), encodeVaults([])])

    const [, , , newDepositIndex] = await strategy.globalVaultState()

    //@note user can then performUpkeep (or wait for keeper to do this) to further increase vaults to 60
    await strategy.performUpkeep(encodeVaults([]))
    assert.equal((await strategy.getVaults()).length, 60) //@audit 20 new vaults are created again

    //@note 300 is now available for withdrawal
    //@note 800 is withdrawable from first strategy
    //@note creating a mock staking pool to simulate this

    //@note 800 + 300 -> total withdrawable is 1100
    //@note user can then withdraw complete amount to repay flash without queueing

    await stakingPool.connect(signers[2]).approve(adrs.priorityPool, ethers.MaxUint256)
    await priorityPool
      .connect(signers[2])
      .withdraw(toEther(1100), toEther(1100), toEther(1100), [], false, false, [
        encodeVaults([]),
        encodeVaults([0, 5, 10]),
      ])
  })
```

Recommended Mitigation: Consider implementing a mechanism to recycle empty vaults instead of always creating new ones. Also to nullify the flash loan vectors, consider adding a time delay for withdrawals.

Stake.link Fixed in 9dcac24

Cyfrin Verified. Instant staking pool withdrawals disallowed. Also, the current vault group deposit room is also considered while depositing into vaults. Both changes combined mitigate the highlighted risk.

7.3 Medium Risk

7.3.1 `WithdrawalPool::withdrawalBatches` array is only every increasing in size potentially leading to Denial Of Service

Description: Every time withdrawals are finalized, a new batchId gets appended to `withdrawalBatches` array. Over time, this array is ever increasing and can lead to extremely large size. It is important to note that users can trigger a queuing of withdrawal by themselves.

Impact: Potential denial of service of `getBatchIds` and `getFinalizedWithdrawalIdsByOwner`

Recommended Mitigation: If `indexOfLastWithdrawal` for a batch is less than the ID of the first queued withdrawal (queuedWithdrawals[0]), it means all withdrawals in that batch and any preceding batches have been fully processed. This can be used to calculate a cut-off batch `id` on a periodic basis.

Consider the following implementation

- Find the cutoff batch: Iterate through withdrawalBatches from the beginning until you find the last batch where indexOfLastWithdrawal < queuedWithdrawals[0].
- Delete or archive: All batches up to and including this cutoff batch can be safely deleted or archived.

Stake.link Fixed in 7889f75

Cyfrin Verified. Fixed as recommended.

7.3.2 Attacker can deny users from depositing into priority pool by front-running call to StakingPool::depositLiquidity()

Description: The `VaultControllerStrategy` contract is susceptible to a front-running attack that can prevent priority pool from depositing user funds into specific vaults of a strategy. The vulnerability lies in the `deposit` function, which uses a `globalState.groupDepositIndex` to track the current deposit state. An attacker can exploit this by front-running legitimate deposits by calling `depositLiquidity` that deposits unused deposits into a strategy.

The griefing attack works as follows:

1. Priority pool prepares a transaction to deposit funds, specifying a list of vault IDs.
2. An attacker observes this pending transaction in the mempool.
3. The attacker quickly submits their own transaction that uses unused staking pool deposits, using the same vault IDs but with a higher gas price.
4. The attacker's transaction is processed first, updating the `globalState.groupDepositIndex`
5. When the priority pool transaction is processed, it fails due to an `InvalidVaultIds` error, as the `globalState.groupDepositIndex` no longer matches the expected value.

Impact: In certain scenarios, priority pool can be prevented from depositing user funds, effectively creating a denial of service condition for the deposit functionality. It is noteworthy that an attacker need not use his own funds to execute the attack but use the unused staking pool deposits (when available) to grief other depositors.

Proof of Concept:

```
it('front-running deposit to change deposit index', async () => {
    const {
      token,
      accounts,
      signers,
      adrs,
      strategy,
      priorityPool,
      stakingPool,
      stakingController,
      rewardsController,
      vaults,
    } = await loadFixture(deployFixture)

    await token.transfer(await stakingPool.getAddress(), toEther(100)) //depositing to represent unused
    ↪  liquidity in staking pool
    //@audit front-run priority pool deposit
    await stakingPool.connect(signers[2]).depositLiquidity([encodeVaults([0, 1, 2, 3, 4])]) // anyone
    ↪  can deposit this

    //@audit actual deposit is DOSed as the group deposit index has changed
    await expect(
      priorityPool.connect(signers[2]).deposit(toEther(100), false, [encodeVaults([0, 1, 3, 4])])
    ).to.be.revertedWithCustomError(strategy, 'InvalidVaultIds()')
  })
```

Recommended Mitigation: Consider one of the alternatives:

- Gate the `StakingPool::depositLiquidity` function to prevent unauthorized access
- Redesign the deposit mechanism to not rely on sequential vault IDs or a global deposit index. Instead, use a more robust method for managing deposits that is resistant to order manipulation.

Stake.link Fixed in 27b3008

Cyfrin Verified. `depositLiquidity` is now a private function.

7.3.3 No way to recover principal if an operator is removed by Chainlink

Description: Chainlink can remove operators from the OperatorStakingPool. This will stop the Operator from accruing any more rewards by removing their principal. Their principal is not lost however, it is still available though by calling `OperatorStakingPool::unstakeRemovedPrincipal`.

In the `OperatorVault` there is no call like this. If an OperatorVault got removed as operator in the chainlink staking pool the pool principal would be locked.

Impact: The funds could eventually be recovered by an upgrade to the vault but that is a long process and until then, the vault behavior would be imperfect as the removed principal is included in the vault principal:

```
    function getPrincipalDeposits() public view override returns (uint256) {
        return
            super.getPrincipalDeposits() +
            IOperatorStaking(address(stakeController)).getRemovedPrincipal(address(this));
    }
```

Thus the vault would appear as it had the principal but it wouldn't be withdrawable.

Recommended Mitigation: Consider adding a call that the operator or owner can do to `unstakeRemovedPrincipal`

Stake.link Fixed in b19b58c

Cyfrin Verified. Support for unstaking removed principal is added.

7.3.4 `Vault::unbondingActive` can be out of sync with Chainlink and `FundFlowController` bonding

Description: Each vault has a call to query whether its currently in an unbonding period, `Vault::unbondingActive`:

```
    function unbondingActive() external view returns (bool) {
        return block.timestamp < stakeController.getClaimPeriodEndsAt(address(this));
    }
```

Here the comparison is <.

However in the Chainlink vault:

```
function _inClaimPeriod(Staker storage staker) private view returns (bool) {
  if (staker.unbondingPeriodEndsAt == 0 || block.timestamp < staker.unbondingPeriodEndsAt) {
    return false;
  }

  return block.timestamp <= staker.claimPeriodEndsAt;
}
```

and `FundFlowController::claimPeriodActive` the comparison is <=:

```
  function claimPeriodActive() external view returns (bool) {
      uint256 claimPeriodStart = timeOfLastUpdateByGroup[curUnbondedVaultGroup] + unbondingPeriod;
      uint256 claimPeriodEnd = claimPeriodStart + claimPeriod;

      return block.timestamp >= claimPeriodStart && block.timestamp <= claimPeriodEnd;
  }
```

Hence at `block.timestamp == staker.claimPeriodEndsAt` the `unbondingActive` call will give the incorrect answer.

Impact: `Vault::unbondingActive` is used in `FundFlowController` to determine which total unbonded and withdrawal orders for upkeep. And also in `VaultControllerStrategy::withdraw`:

```
  function withdraw(uint256 _amount, bytes calldata _data) external onlyStakingPool {
      // @audit claimPeriodActive() does `<= claimPeriodEnd`
      if (!fundFlowController.claimPeriodActive() || _amount > totalUnbonded)
          revert InsufficientTokensUnbonded();

      // ...

      for (uint256 i = 0; i < vaultIds.length; ++i) {
          // ...

          // @audit unbondingActive() does `< claimPeriodEnd`
          if (deposits != 0 && vault.unbondingActive()) {
              if (toWithdraw > deposits) {
                  vault.withdraw(deposits);
                  unbondedRemaining -= deposits;
                  toWithdraw -= deposits;
              } else if (deposits - toWithdraw > 0 && deposits - toWithdraw < minDeposits) {
                  vault.withdraw(deposits);
                  unbondedRemaining -= deposits;
                  break;
              } else {
                  vault.withdraw(toWithdraw);
                  unbondedRemaining -= toWithdraw;
                  break;
              }
          }
      }
```

Hence at `block.timestamp == staker.claimPeriodEndsAt` the withdraw would incorrectly not withdraw anything.

This also applies to `VaultControllerStrategy::_depositToVaults`:

```
                if (vault.unbondingActive()) {
                    totalRebonded += deposits;
                }
```

Where `totalRebonded`, and in turn `totalUnbonded` would be wrong. However that is much less impactful as the vault is just about to go out of claim period and thus void until `FundFlowController::performUpkeep` is called which resets `totalUnbonded`.

Proof of Concept: Test that can be added to `vault-controller-strategy.test.ts`:

```
  it('should perform withdrawal at claim period end' , async () => {
    const {accounts, adrs, strategy, token, stakingController, vaults, fundFlowController,
    ↪  updateVaultGroups } =
      await loadFixture(deployFixture)

    // Deposit into vaults
    await strategy.deposit(toEther(500), encodeVaults([]))
    assert.equal(fromEther(await token.balanceOf(adrs.stakingController)), 500)
    for (let i = 0; i < 5; i++) {
      assert.equal(fromEther(await stakingController.getStakerPrincipal(vaults[i])), 100)
    }
    assert.equal(Number((await strategy.globalVaultState())[3]), 5)

    await updateVaultGroups([0, 5, 10], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([1, 6, 11], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([2, 7], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([3, 8], 0)
    await time.increase(claimPeriod)
    await updateVaultGroups([4, 9], 100)

    const claimPeriodEnd = Number(await fundFlowController.timeOfLastUpdateByGroup(0)) +
    ↪  unbondingPeriod + claimPeriod

    const balanceBefore = await token.balanceOf(accounts[0])

    // set to claim period end
    await time.setNextBlockTimestamp(claimPeriodEnd)
    await strategy.withdraw(toEther(50), encodeVaults([0, 5]))


    const balanceAfter = await token.balanceOf(accounts[0])

    // nothing was withdrawn
    assert.equal(balanceAfter - balanceBefore, 0n)
  })
```

Recommended Mitigation: Consider changing < to <=:

```
    function unbondingActive() external view returns (bool) {
-       return block.timestamp < stakeController.getClaimPeriodEndsAt(address(this));
+       return block.timestamp <= stakeController.getClaimPeriodEndsAt(address(this));
    }
```

Stake.link Fixed in b19b58c

Cyfrin Verified. Claim period start is correctly accounted for.

7.3.5 Potential stale data in `FundFlowController::performUpkeep` can lead to incorrect state updates

Description: The `FundFlowController::checkUpkeep` function calculates `nextGroupOpVaultsTotalUnbonded` and `nextGroupComVaultsTotalUnbonded` by iterating over vaults and checking their `unbondingActive` status. This data is then encoded and passed to `performUpkeep` by keepers.

However, there's a potential time delay between `checkUpkeep` and `performUpkeep` execution, which could lead to stale data being used to update the system state. The `unbondingActive` status of vaults is time-sensitive, based on the current block timestamp and each vault's claim period end time. If the `performUpkeep` transaction is delayed due to network congestion or other factors, the `totalUnbonded` values used for updates may no longer accurately reflect the current state of the vaults.

`FundFlowController.sol`

```
function checkUpkeep(bytes calldata) external view returns (bool, bytes memory) {
    // ... (other code)
    (
        uint256[] memory curGroupOpVaultsToUnbond,
        uint256 nextGroupOpVaultsTotalUnbonded
    ) = _getVaultUpdateData(operatorVCS, nextUnbondedVaultGroup);
    // ... (similar for community vaults)
    return (
        true,
        abi.encode(
            curGroupOpVaultsToUnbond,
            nextGroupOpVaultsTotalUnbonded,
            curGroupComVaultsToUnbond,
            nextGroupComVaultsTotalUnbonded
        )
    );
}

function performUpkeep(bytes calldata _data) external {
    // ... (decoding and using potentially stale data)
}
```

Impact: The use of stale data in performUpkeep can lead to incorrect state updates. The system may allow more or fewer unbondings than it should, leading to discrepancies between the recorded state and the actual state of the vaults. In a worst-case scenario, funds that should be unbonded might remain locked, or funds that should still be locked might be prematurely released.

Recommended Mitigation: Consider modifying `performUpkeep` to recalculate the `totalUnbonded` values at the time of execution, rather than relying on potentially stale data from checkUpkeep.

Stake.link Fixed in 53d6090

Cyfrin Verified. Fixed as recommended.

7.3.6 Staking rewards susceptible to reward multiplier manipulation

Description: For staking in the Chainlink Operator and Community staking pools a staker is rewarded through the Chainlink RewardVault. Here rewards are accrued per time staked.

There is however a twist to how the rewards work compared to the standard rewards design. To promote staying in your position for an extended period of time, Chainlink uses a `multiplier`. This goes from 0 to 1 over a period of 90 days (at time of writing). Just as you start staking you have a multiplier of 0, then i linearly increases to 1 over 90 days.

Were you to unstake during this period, the multiplier is reset and the forfeited rewards you should have earned are distributed to the other stakers.

This can be used to grief LinkPool stakers to reduce their rewards received rewarding stakers outside of LinkPool.

A user could iterate depositing a full vault (+ 1 LINK) and withdrawing a full vault to iterate through the whole current vault group. Thus resetting all the multipliers for the vaults.

Impact: A quick overview of the impact using current numbers from the community pool:

Current emission rate for community stakers is `~0.05 LINK/s`. The community vault is currently full with a total of `40875000 LINK` staked. This gives a gain per token per second of `0.05 / 40875000 =~ 1.2e-09`

LinkPool has a total stake `1324452 LINK`, ~3% of the pool.

As mentioned above, the multiplier period is `90 days` and the unbonding period is `4 weeks` which is the fastest you can reset the multiplier.

This gives a reward calculation:

```
perTokenPerS*(staked*0.2)*4 weeks =~ 783 LINK
```

To get the total forfeited you'll need to multiply with `1-(staked time/multiplier period)`:

```
perTokenPerS*(staked*0.2)*4 weeks*(1 - (4 weeks/90 days)) =~ 540 LINK
```

Hence in this case a total of 540 LINK would be lost. Note that this only applies to when the vaults start from 0 multiplier. If a vault was previously at `>90 days` there wouldn't be any forfeit hence the gain is only gotten the second time you do this for a vault group. As it is only then the multiplier is completely reset.

To see the profitability of this, the gained `540 LINK` is split into the pool:

```
540/40875000 = 1.3e-5 per held link
```

If a user has 100 vaults (with maximum deposit 15000 LINK each), this would gain them a total of 19 LINK which is in the ballpark of what the gas would probably cost of this.

Hence the attack vector is pretty small. However there is still a possibility for griefing or un-intended loss of rewards due to vaults having their multiplier unnecessarily reset.

Note, that this is taken with the current community staking pool distribution. The attack grows more profitable the larger the LinkPool stake is.

Recommended Mitigation: For the attack vector of resetting all the vaults multipliers it relies on being able to easily iterate through the vaults with deposits + withdrawals. This can be mitigated by implementing a small withdrawal timelock.

For the unintentional loss of rewards that can happen if you withdraw from vaults that have been staking for <90 days, consider enforcing a rotation of the `withdrawalIndex`. Right now any can be chosen but if it always increases (until it loops back) it will hopefully keep vaults "alone" for more than 90 days.

Stake.link Fixed in 9dcac24

Cyfrin Verified. Instant staking pool withdrawals disallowed.

7.4 Low Risk

7.4.1 Lack of validation checks in `PriorityPool::depositQueuedTokens` can trigger temporary delays in queued deposits

Description: `PriorityPool::depositQueuedTokens` lacks a validation check to ensure `_queueDepositMin < _queueDepositMax`. This oversight alllows a griefing attack vector where a user can call `depositQueuedTokens` with `_queueDepositMin` as the `max(strategyDepositRoom, canDeposit)` and `_queueDepositMax=0`.

This forces staking pool to always deposit unused deposits into strategy and bypassing any queued amounts in priority pool. In cases where total depositable amount, ie. total queued amount plus unused deposits exceeds the `queuedDepositMin` but just the total queued amount is less than `queuedDepositMin`, the above manipulation would mean that queued deposits are unnecessarily delayed.

Impact: Griefing of users who would want their queued amounts to be deposited into strategies as early as possible.

Recommended Mitigation: Consider adding a check `_queueDepositMin < _queueDepositMax` in `depositQueuedTokens` function

Stake.link Acknowledged.

Cyfrin Acknowledged.

7.4.2 FundflowController goes out-of-sync when Chainlink changes their unbonding and claim periods

Description: In `FundFlowController` there are two fields: `unbondingPeriod` and `claimPeriod` which should be in sync with the corresponding state in the Chainlink contracts:

```
    uint64 public unbondingPeriod;
    uint64 public claimPeriod;
```

Chainlink can however change these using `StakingPoolBase::setUnbondingPeriod` and `StakingPoolBase::setClaimPeriod`. Since there isn't a call in `FundFlowController` to update the states these changes would not be reflected.

Also, the design of `performUpkeep` doesn't work well if the periods in `FundFlowController` would change:

```
        if (
            timeOfLastUpdateByGroup[nextUnbondedVaultGroup] != 0 &&
            block.timestamp <=
            timeOfLastUpdateByGroup[curUnbondedVaultGroup] + unbondingPeriod + claimPeriod // @audit
            ↪   new periods used
        ) revert NoUpdateNeeded();

        if (
            curUnbondedVaultGroup != 0 &&
            timeOfLastUpdateByGroup[curUnbondedVaultGroup] == 0 &&
            block.timestamp <= timeOfLastUpdateByGroup[curUnbondedVaultGroup - 1] + claimPeriod //
            ↪   @audit new period used
        ) revert NoUpdateNeeded();

        if (block.timestamp < timeOfLastUpdateByGroup[nextUnbondedVaultGroup] + unbondingPeriod) //
        ↪   @audit new period used
            revert NoUpdateNeeded();

...
        // @audit time of unbonding stored, not the resulting unbonding timestamp and claim timestamp
        timeOfLastUpdateByGroup[curUnbondedVaultGroup] = uint64(block.timestamp);
```

Here the time of unbonding is stored and the current state of the unbonding periods is used. The Chainlink contracts applies the delay when unbonding:

```
    staker.unbondingPeriodEndsAt = (block.timestamp + s_pool.configs.unbondingPeriod).toUint128();
    staker.claimPeriodEndsAt = staker.unbondingPeriodEndsAt + s_pool.configs.claimPeriod;
```

Hence at the time of change, `FundFlowController` would go out of sync.

Impact: If the `unbonding` and `claim` periods were to change in Chainlink, the state kept in `FundFlowController` would be out of date.

Even if `FundFlowController` were upgraded or a new one deployed the state of the vault groups unbonding and claim periods would effectively be out of date until all have had their periods expire.

This would cause significant disruptions to the withdrawals in the protocol causing user funds to be locked longer than needed.

Recommended Mitigation: When a user unbonds in Chainlink, Chainlink saves both the unbonding timestamp and the claim timestamp. We suggest LinkPool does the same. At time of unbonding, consider returning the unbonding and claim timestamp as they were saved at chainlink.

Then the `VaultControllerStrategy` propagates this (since they all *must* be the same it can just return the last) to the `FundFlowController` which can save the same unbonding and claim timestamps. Then use them for the rotation logic.

Stake.link Acknowledged.

Cyfrin Acknowledged.

7.4.3 Chainlink can have different bonding and claim periods for Operator and Community staking pools

Description: `FundFlowController` uses the same `unbondingPeriod` and `claimPeriod` for both the Operator and Community pools.

Since the Chainlink Operator and Community pools are different contracts deployed. Chainlink can change the time periods for unbonding and claiming hence there's no guarantee that they'll always be the same.

Impact: Were Chainlink to set different unbonding and claim periods in their operator and community vaults this would affect the ability to withdraw in LinkPool.

Since the state in `FundFlowController` would only be able to accurately track one of them. This could also cause reverts and complications with running `checkUpkeep` since reverts happen in the Chainlink vaults when a claim period is still active. Since the controls in `checkUpkeep` would no longer be up to date this would inevitably happen sometimes.

Recommended Mitigation: Consider, instead of having one `FundFlowController` for both the community and operator pools, having one for each. That would give the flexibility to track the different periods separately.

Stake.link Acknowledged.

Cyfrin Acknowledged.

7.4.4 User can frontrun operator slashing/update rewards by withdrawing from the pool

Description: When an operator is slashed a part of their staked principal is removed and transferred to the slasher. This will happen to all the slashable operators configured in the Chainlink `PriceFeedAlertsController`. This slashing would lower the amount of LINK that is staked and thus lower the value of the `stLINK` token.

Note, that it is not technically when the slashing happens the value (exchange rate for LINK) is changed but when the `totalStaked` is updated which is done on the call to `StakingPool::updateStrategyRewards` for the corresponding strategy.

A large holder of `stLINK` could thus have seen the slashing event and then front run the call to `updateStrategyRewards` with withdrawing, thus getting the higher rate of `LINK` than the users still holding.

Impact: Using the numbers from `StakingVault` and `OperatorStakingPool`:

```
staked=2438937.970114142763711705
shares=2193487.121097554919462043

staked/shares = 1.1118998359533434
```

Given that there are 9 vaults that are currently slashable in the `PriceFeedAlertsController`, that would give a new fx rate of:

```
slashAmount=700

(staked-9*slashedAmout)/shares=1.109027696910718
```

Which is a difference of `0.002` link per share. If you take a position of `~15000 stLINK`, which is roughly one complete vault gives us this will save you `~40 LINK`, which is ~$400, more than the gas cost but not any enormous amount.

Proof of Concept:

```
it('user can withdrawal before updateStrategyRewards instant', async () => {
    const { adrs, vaults, signers, token, pp, stakingController, stakingPool, updateVaultGroup}
      = await loadFixture(deployFixture)

    const alice = signers[2]

    await pp.deposit(toEther(475), false, [encodeVaults([])])

    await token.transfer(alice, toEther(25))
    await pp.connect(alice).deposit(toEther(100), false, [encodeVaults([])])

    assert.equal(vaults.length, 5)
    for(let i = 0; i < 5; i++) {
      assert.equal(fromEther(await stakingController.getStakerPrincipal(vaults[i])), 100)
    }

    // unbond all groups
    for(let i = 0; i < 5; i++) {
      await updateVaultGroup(i)

      // don't increase after last iteration to keep the current group in claim period
      if(i < 4) {
        await time.increase(claimPeriod + 1)
      }
    }

    // slashing happens
    await stakingController.slashStaker(vaults[0], toEther(50))
    assert.equal(fromEther(await stakingController.getStakerPrincipal(vaults[0])),50)


    const balanceBefore = await token.balanceOf(alice.address)
    await stakingPool.connect(alice).approve(adrs.pp, toEther(25))
    await pp.connect(alice).withdraw(
      toEther(25),
      toEther(25),
      toEther(25),
      [ethers.ZeroHash],
      false,
      false,
      [encodeVaults([0])]
    )
    const balanceAfter = await token.balanceOf(alice.address)
    assert.equal(balanceAfter - balanceBefore, toEther(25))

    // -1000 because of initial burnt amount
    assert.equal(Number(await stakingPool.balanceOf(signers[0].address)), Number(toEther(475))-1000)

    // when strategy reward are updated it will reduce the values of the shares held by the rest of the
    ↪   stakers
    await stakingPool.updateStrategyRewards([0],'0x')

    assert.equal(Number(await stakingPool.balanceOf(signers[0].address)), Number(toEther(425))-1000)
  })
```

Recommended Mitigation: Consider implementing some sort of delay on withdrawals. It doesn't need to be as long as Chainlink just so that it cant be done in one tx.

Stake.link Acknowledged. This issue should be largely mitigated when `RebaseController` is deployed.

Cyfrin Acknowledged.

7.4.5 Withdrawals when Chainlink pool is paused will take longer than necessary

Description: The chainlink docs explain that withdrawals can happen under these circumstances:

```
/// @dev precondition The caller must be staked in the pool.
/// @dev precondition The caller must be in the claim period or the pool must be closed or paused.
```

Here they always allow withdrawals to happen when the pool is closed or paused.

This is not reflected in `FundFlowController` as for a withdrawal to happen the pool must be in the correct vault group. Hence "emptying" out the vaults after a pause will take at least 4*claim period.

Impact: If the chainlink pool closes clearing out the positions held by LinkPool will take up to 4 times longer than necessary.

Recommended Mitigation: Consider allowing withdrawals from vaults outside the current vault group and claim period if the staking controller is paused.

Stake.link Acknowledged.

Cyfrin Acknowledged.

7.5 Gas Optimization

7.5.1 Gas optimisation of `FundFlowController::_getVaultDepositOrder`

Description: In `FundFlowController:: _getVaultDepositOrder`, the first vault is always the vault corresponding to the `groupDepositIndex`, as seen below

```
      if (groupDepositIndex < maxVaultIndex) {
            vaultDepositOrder[0] = groupDepositIndex;
            ++totalVaultsAdded;
            (uint256 withdrawalIndex, ) = _vcs.vaultGroups(groupDepositIndex % numVaultGroups);
            uint256 deposits = IVault(vaults[groupDepositIndex]).getPrincipalDeposits();
            if (
                deposits != maxDeposits && (groupDepositIndex != withdrawalIndex || deposits == 0)
            ) {
                totalDepositsAdded += maxDeposits - deposits;//@audit missing check to see if
                ↪  _toDeposits is already accomodated
            }
        }
```

It is likely that `toDeposit` can be filled with the deposit room of just this vault. In this case, it is prudent to check if `totalDepositsAdded >= _toDeposit` and exit with just a single vault ID in the vault order.

Additionally, in this case, the costly `_sortIndexesDescending` is not necessary as it is only applicable from second vault onwards.

Recommended Mitigation: Consider the following:

1. Add a check of `totalDepositsAdded >= _toDeposit` right after vault 0 and return with just a single vault if condition is satisfied.
2. Move the `_sortIndexesDescending` after including vault 0

Stake.link Acknowledged.

Cyfrin Acknowledged.

7.5.2 Cache storage reads in `FundFlowController::performUpkeep`

Description: `FundFlowController::performUpkeep` contains a lot of repetitive reads of storage variables like: `curUnbondedVaultGroup`, `unbondingPeriod`, `claimPeriod`, `timeOfLastUpdateByGroup[curUnbondedVaultGroup]` and `timeOfLastUpdateByGroup[nextUnbondedVaultGroup]`

Recommended Mitigation: Consider caching these to save unnecessary storage reads.

Stake.link Acknowledged.

Cyfrin Acknowledged.

7.5.3 Optimize VaultGroup and Fee Structs Using Bitwise Operations

Description: The current `VaultGroup` and `Fee` structs in the `VaultControllerStrategy` contract are defined as:

```
struct VaultGroup {
    uint64 withdrawalIndex;
    uint128 totalDepositRoom;
}

struct Fee {
    address receiver;
    uint256 basisPoints;
}
```

These structs can be further optimized by packing them into a single uint192, using bitwise operations for access and modification. This is especially beneficial as these structs are used in arrays, potentially leading to significant gas savings. Note that the maximum basisPoints is 10000 (100%), so it can fit into a 32 bit slot.

Recommended Mitigation: Consider packing the `VaultGroup` struct into a single uint192 (64 bits for withdrawalIndex, 128 bits for totalDepositRoom) as follows:

```
uint192[] public vaultGroups;

function getVaultGroup(uint256 index) public view returns (uint64 withdrawalIndex, uint128
↪   totalDepositRoom) {
    uint192 group = vaultGroups[index];
    withdrawalIndex = uint64(group);
    totalDepositRoom = uint128(group >> 64);
}

function setVaultGroup(uint256 index, uint64 withdrawalIndex, uint128 totalDepositRoom) internal {
    vaultGroups[index] = uint192(withdrawalIndex) | (uint192(totalDepositRoom) << 64);
}
```

Consider packing the `Fee` struct into a single uint192 (160 bits for address, 32 bits for basisPoints) as follows:

```
uint192[] public fees;

function getFee(uint256 index) public view returns (address receiver, uint32 basisPoints) {
    uint256 fee = fees[index];
    receiver = address(uint160(fee));
    basisPoints = uint32(fee >> 160);
}

function setFee(uint256 index, address receiver, uint32 basisPoints) internal {
    require(basisPoints <= 10000, "Basis points cannot exceed 10000");
    fees[index] = uint192(uint160(receiver)) | (uint192(basisPoints) << 160);
}
```

Stake.link Acknowledged.

Cyfrin Acknowledged.