POLYGON STAKING Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

holydevoti0n

May 19, 2025

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **2**

6 Executive Summary **2**

7 Findings **5**

7.1 Medium Risk . . . 5

7.1.1 Unfair distribution of unbonding due to improper validator withdrawal index advancement . . . 5

7.1.2 Validator and protocol fees can exceed 100 percent of rewards in `PolygonStrategy.sol` . . . 6

7.2 Low Risk . . . 10

7.2.1 `PolygonStrategy::upgradeVaults` does not verify if vaults to upgrade are the same as the ones stored in strategy . . . 10

7.2.2 Reward restaking creates unnecessary unbonding in edge cases . . . 10

7.2.3 DoS in unbonding when validator rewards fall below min amount threshold . . . 12

7.2.4 `disableInitializers` not used to prevent uninitialized contracts . . . 14

7.3 Informational . . . 16

7.3.1 Missing event emission for PolygonStrategy::setFundFlowController . . . 16

7.3.2 Missing check for `validatorMEVRewardsPercentage` in `PolygonStrategy::initialize` . . . 16

7.3.3 Missing zero address checks . . . 16

7.3.4 Incorrect event emission when removing fees via the `PolygonStrategy::updateFee` . . . 17

7.3.5 Missing check for `_feeBasisPoints > 0` in `PolygonStrategy::addFee` . . . 17

7.3.6 No slippage protection when interacting with `ValidatorShares` . . . 18

7.3.7 Missing input validation in `PolygonFundFlowController.setMinTimeBetweenUnbonding` . . . 18

7.3.8 Potential infinite loop in `PolygonStrategy::unbond` due to insufficient balance check . . . 19

7.4 Gas Optimization . . . 20

7.4.1 Validate total fees in `PolygonStrategy::updateFee` only if the `_feeBasisPoints > 0` . . . 20

7.4.2 Inefficient rewards collection pattern in the `PolygonStrategy::unbond()` function leads to unnecessary gas consumption . . . 20

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

5 Audit Scope

Following contracts were included in the aduit scoope:

- PolygonFundFlowController.sol
- PolygonVault.sol
- PolygonStrategy.sol

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the POLYGON STAKING smart contracts provided by Stake.Link. In this period, a total of 16 issues were found.

This audit covers the Stake.Link Polygon staking protocol, which consists of a fund flow controller, strategy, and vault system for staking assets on the Polygon network. The code enables efficient staking, management of rewards, and seamless withdrawals for the protocol's users.

The audit identified two medium-risk issues, one relating to the distribution of unbonding requests across validators and second, relating to the fees exceeding 100% of rewards due to insufficient validation. The audit also found several low-risk and informational issues. All major issues were successfully mitigated by the Stake.Link team.

Summary

Project Name	POLYGON STAKING
Repository	contracts
Commit	f41aadb35f72...
Audit Timeline	May 5th - May 9th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	4
Informational	8
Gas Optimizations	2
Total Issues	16

Summary of Findings

[M-1] Unfair distribution of unbonding due to improper validator withdrawal index advancement	Resolved
[M-2] Validator and protocol fees can exceed 100 percent of rewards in `PolygonStrategy.sol`	Resolved
[L-1] `PolygonStrategy::upgradeVaults` does not verify if vaults to upgrade are the same as the ones stored in strategy	Resolved
[L-2] Reward restaking creates unnecessary unbonding in edge cases	Acknowledged
[L-3] DoS in unbonding when validator rewards fall below min amount threshold	Resolved
[L-4] `disableInitializers` not used to prevent uninitialized contracts	Resolved
[I-1] Missing event emission for PolygonStrategy::setFundFlowController	Acknowledged
[I-2] Missing check for `validatorMEVRewardsPercentage` in `PolygonStrategy::initialize`	Resolved
[I-3] Missing zero address checks	Resolved
[I-4] Incorrect event emission when removing fees via the `PolygonStrategy::updateFee`	Resolved
[I-5] Missing check for `_feeBasisPoints > 0` in `PolygonStrategy::addFee`	Acknowledged
[I-6] No slippage protection when interacting with `ValidatorShares`	Acknowledged
[I-7] Missing input validation in `PolygonFundFlowController.setMinTimeBetweenUnbonding`	Acknowledged

[I-8] Potential infinite loop in `PolygonStrategy::unbond` due to insufficient balance check	Resolved
[G-1] Validate total fees in `PolygonStrategy::updateFee` only if the `_feeBasisPoints > 0`	Resolved
[G-2] Inefficient rewards collection pattern in the `PolygonStrategy::unbond()` function leads to unnecessary gas consumption	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Unfair distribution of unbonding due to improper validator withdrawal index advancement

Description: `PolygonStrategy::unbond` tracks the `validatorWithdrawalIndex` to evenly distribute unbonding requests across vaults. The index is advanced based on the loop progression rather than actual unbonding actions, which can lead to uneven distribution of unbonding across validators.

Currently, when processing vaults during unbonding:

- The function starts at the validatorWithdrawalIndex and iterates through vaults
- If a vault has sufficient rewards to cover the unbonding amount, only rewards are withdrawn with no actual unbonding of principal
- The validatorWithdrawalIndex is still updated to the next vault in sequence, even if no principal was unbonded
- If the index reaches the end of the vaults array, it wraps around to 0

This logic causes validators with high rewards to effectively "shield" their principal deposits from being unbonded, unfairly pushing the unbonding burden to other validators that might have already unbonded.

Specifically, in the unbond function:

```
while (toUnbondRemaining != 0) {
    // Process vault[i]...
    ++i;
    if (i >= vaults.length) i = 0;
}
validatorWithdrawalIndex = i; // @audit This happens regardless of whether unbonding occurred on the
↪  last processed vault or not
```

Even if the rewards were sufficient to honor `unbond` request, `vaultWithdrawalIndex` still advances.

Impact: This issue creates a potential fairness problem in the distribution of unbonding actions across validators. Validators that generate more rewards are less likely to have their principal deposits unbonded, while validators with fewer rewards will bear a disproportionate share of the unbonding burden.

Effectively, some vaults get repeatedly unbonded while others skip their turn.

Proof of Concept: In the `unbond should work correctly` test, at line 304 in `polygon-strategy.test.ts`, note that the `validatorWithdrawalIndex` resets to 0 even though the there was no unbonding on vault[2]. vault[0] gets unbonded twice while vault[2] skips unbonding altogether in the current round.

Recommended Mitigation: Consider modifying the `unbond` function to only advance the `validatorWithdrawalIndex` when actual unbonding of principal deposits occurs, not just when rewards are withdrawn:

```
function unbond(uint256 _toUnbond) external onlyFundFlowController {
    if (numVaultsUnbonding != 0) revert UnbondingInProgress();
    if (_toUnbond == 0) revert InvalidAmount();

    uint256 toUnbondRemaining = _toUnbond;
    uint256 i = validatorWithdrawalIndex;
    uint256 skipIndex = validatorRemoval.isActive
        ? validatorRemoval.validatorId
        : type(uint256).max;
    uint256 numVaultsUnbonded;
    uint256 preBalance = token.balanceOf(address(this));
++ uint256 nextIndex = i; // Track the next index separately

    while (toUnbondRemaining != 0) {
        if (i != skipIndex) {
```

```
            IPolygonVault vault = vaults[i];
            uint256 deposits = vault.getTotalDeposits();

            if (deposits != 0) {
                uint256 principalDeposits = vault.getPrincipalDeposits();
                uint256 rewards = deposits - principalDeposits;

                if (rewards >= toUnbondRemaining) {
                    vault.withdrawRewards();
                    toUnbondRemaining = 0;
                } else {
                    toUnbondRemaining -= rewards;
                    uint256 vaultToUnbond = principalDeposits >= toUnbondRemaining
                        ? toUnbondRemaining
                        : principalDeposits;

                    vault.unbond(vaultToUnbond);
++                   nextIndex = (i + 1) % vaults.length; // Update next index only when unbonding principal
                    toUnbondRemaining -= vaultToUnbond;
                    ++numVaultsUnbonded;
                }
            }
        }

        ++i;
        if (i >= vaults.length) i = 0;
    }
--    validatorWithdrawalIndex = i;
++    // Only update the index if we actually unbonded principal from at least one vault
++    if (numVaultsUnbonded > 0) {
++       validatorWithdrawalIndex = nextIndex;
++   }

    numVaultsUnbonding = numVaultsUnbonded;

    uint256 rewardsClaimed = token.balanceOf(address(this)) - preBalance;
    if (rewardsClaimed != 0) totalQueued += rewardsClaimed;

    emit Unbond(_toUnbond);
}
```

Stake.link: Resolved in PR 151.

Cyfrin: Resolved.

7.1.2 Validator and protocol fees can exceed 100 percent of rewards in `PolygonStrategy.sol`

Description: The `setValidatorMEVRewardsPercentage` function in `PolygonStrategy.sol` allows setting the validator MEV rewards percentage up to 100% (10,000 basis points), but does not account for the total protocol fees (`_totalFeesBasisPoints`). This means the combined sum of all fees can exceed 100% of the rewards, leading to over-distribution of rewards. While `setValidatorMEVRewardsPercentage` only checks its own limit:

```
function setValidatorMEVRewardsPercentage(
    uint256 _validatorMEVRewardsPercentage
) external onlyOwner {
    // @audit should consider the totalFeesBasisPoints.
    // updateDeposits will report more than it can pay in fees.
    if (_validatorMEVRewardsPercentage > 10000) revert FeesTooLarge();

    validatorMEVRewardsPercentage = _validatorMEVRewardsPercentage;
```

```
    emit SetValidatorMEVRewardsPercentage(_validatorMEVRewardsPercentage);
}
```

Notice the other fees has a limit of 30% in basis points:

```
// updateFee
if (_totalFeesBasisPoints() > 3000) revert FeesTooLarge();
```

This separation allows the sum of `validatorMEVRewardsPercentage` and protocol fees to exceed 100% of the rewards, which can lead to an over-minting scenario in `StakingPool.sol`:

```
function updateDeposits(
        bytes calldata
    )
        external
        onlyStakingPool
        returns (int256 depositChange, address[] memory receivers, uint256[] memory amounts)
    {
    ...
@>            uint256 validatorMEVRewards = ((balance - totalQueued) *
                validatorMEVRewardsPercentage) / 10000;

            receivers = new address[](fees.length + (validatorMEVRewards != 0 ? 1 : 0));
            amounts = new uint256[](receivers.length);

            for (uint256 i = 0; i < fees.length; ++i) {
                receivers[i] = fees[i].receiver;
@>                amounts[i] = (uint256(depositChange) * fees[i].basisPoints) / 10000;
            }

            if (validatorMEVRewards != 0) {
@>                receivers[receivers.length - 1] = address(validatorMEVRewardsPool);
@>                amounts[amounts.length - 1] = validatorMEVRewards;
            }
           ...
}

// StakingPool._updateStrategyRewards
uint256 sharesToMint = (totalFeeAmounts * totalShares) /
    (totalStaked - totalFeeAmounts);
_mintShares(address(this), sharesToMint);
```

Example

1. The owner sets `validatorMEVRewardsPercentage` to 100% (10,000 basis points).
2. The protocol fee is 20% (2,000 basis points) via `_totalFeesBasisPoints`.
3. The contract now has a combined fee rate of 120% (10,000 + 2,000 basis points).
4. When 10 tokens are available as rewards, the `updateDeposits` reports to `StakingPool` logic 12 tokens to be minted.
5. This results in the system issuing more tokens than it actually has, leading to over-minting of shares in the pool.

Impact: protocol fees can exceed 100% of rewards

Proof of Concept: In `polygon-strategy.test.ts` modify the strategy initialization to include 20% of fees:

```
const strategy = (await deployUpgradeable('PolygonStrategy', [
      token.target,
      stakingPool.target,
      stakeManager.target,
```

```
      vaultImp,
      2500,
-     []
+      [
+        {
+          receiver: ethers.Wallet.createRandom().address,
+          basisPoints: 1000 // 10%
+        },
+        {
+          receiver: ethers.Wallet.createRandom().address,
+          basisPoints: 1000 // 10%
+        }
+      ],
    ])) as PolygonStrategy
```

Then add the following test and run `npx hardhat test test/polygonStaking/polygon-strategy.test.ts`:

```
describe.only('updateDeposits mint above 100%', () => {
   it('should cause stakingPool to overmint rewards', async () => {
     const { strategy, token, vaults, accounts, stakingPool, validatorShare, validatorShare2 } =
       await loadFixture(deployFixture)

   await strategy.setValidatorMEVRewardsPercentage(10000) // 100%
   // current fees is 20% for receivers + 100% for validator rewards

   await stakingPool.deposit(accounts[1], toEther(1000), ['0x'])
   await strategy.depositQueuedTokens([0, 1, 2], [toEther(10), toEther(20), toEther(30)])

   assert.equal(fromEther(await strategy.getTotalDeposits()), 1000)
   assert.equal(fromEther(await strategy.totalQueued()), 940)
   assert.equal(fromEther(await strategy.getDepositChange()), 0)

   // send 10 MATIC to strategy
   await token.transfer(strategy.target, toEther(10))
   assert.equal(fromEther(await strategy.getDepositChange()), 10)

  // @audit we will mint 120% of 10 instead of 100%
  // notice totalRewards is 10(depositChange),
  await expect(stakingPool.updateStrategyRewards([0], '0x'))
  .to.emit(stakingPool, "UpdateStrategyRewards")
  // totalRewards = strategy.depositChange
  //       msg.sender,  totalStaked,   totalRewards, totalFeeAmounts
  .withArgs(accounts[0], toEther(1010), toEther(10), toEther(12));
    })
  });
```

Output:

```
PolygonStrategy
    updateDeposits mint above 100%
       should cause stakingPool to overmint rewards (608ms)

  1 passing (611ms)
```

Recommended Mitigation: When updating fees check if total fees(mevRewards + fees) do not surpass 100%:

```
    function setValidatorMEVRewardsPercentage(
        uint256 _validatorMEVRewardsPercentage
    ) external onlyOwner {
-         if (_validatorMEVRewardsPercentage > 10000) revert FeesTooLarge();
+        if (_validatorMEVRewardsPercentage + _totalFeesBasisPoints() > 10000) revert FeesTooLarge();
```

```
        validatorMEVRewardsPercentage = _validatorMEVRewardsPercentage;
        emit SetValidatorMEVRewardsPercentage(_validatorMEVRewardsPercentage);
    }
    function addFee(address _receiver, uint256 _feeBasisPoints) external onlyOwner {
        _updateStrategyRewards();
        fees.push(Fee(_receiver, _feeBasisPoints));
-        if (_totalFeesBasisPoints() > 3000) revert FeesTooLarge();
+       uint256 totalFees = _totalFeesBasisPoints();
+       if (totalFees > 3000 || totalFees + validatorMEVRewardsPercentage > 10000) revert
↪   FeesTooLarge();
        emit AddFee(_receiver, _feeBasisPoints);
    }

       function updateFee(
        uint256 _index,
        address _receiver,
        uint256 _feeBasisPoints
    ) external onlyOwner {
        _updateStrategyRewards();

        if (_feeBasisPoints == 0) {
            fees[_index] = fees[fees.length - 1];
            fees.pop();
        } else {
            fees[_index].receiver = _receiver;
            fees[_index].basisPoints = _feeBasisPoints;
        }

-        if (_totalFeesBasisPoints() > 3000) revert FeesTooLarge();
+       uint256 totalFees = _totalFeesBasisPoints();
+       if (totalFees > 3000 || totalFees + validatorMEVRewardsPercentage > 10000) revert
↪   FeesTooLarge();
        emit UpdateFee(_index, _receiver, _feeBasisPoints);
    }
```

Stake.link: Resolved in PR 151

Cyfrin: Resolved.

7.2 Low Risk

7.2.1 `PolygonStrategy::upgradeVaults` does not verify if vaults to upgrade are the same as the ones stored in strategy

Description: `PolygonStrategy::upgradeVaults` takes an input of `_vaults` array and upgrades them to the latest `vaultImplementation` as follows:

```
function upgradeVaults(address[] calldata _vaults, bytes[] memory _data) external onlyOwner {

        for (uint256 i = 0; i < _vaults.length; ++i) { //@audit no check if the _vault address is
        ↪  actually one locally stored
            if (_data.length == 0 || _data[i].length == 0) {
                IPolygonVault(_vaults[i]).upgradeTo(vaultImplementation);
            } else {
                IPolygonVault(_vaults[i]).upgradeToAndCall(vaultImplementation, _data[i]);
            }
        }
        emit UpgradedVaults(); //@audit does not emit the list of vaults actually updated
    }
```

Current implementation has several issues:

1. There is no check to verify if the input addresses actually match the ones locally stored in the contract. Owner can send totally different addresses and still can trigger the `UpgradedVaults` event
2. `UpgradedVaults` event does not contain the list of vault addresses actually updated. This prevents off-chain listeners from detecting which vaults were actually upgraded.
3. Passing specific vaults as input can lead to a scenario where a section of vaults are running on the latest implementation while another batch correspond to an older vault implementation. Not only can this create confusion among protocol admins but it can also make the upgrade process risky, specially when a bug is discovered in the vault implementation contract.

Impact: Lack of transparency around vault upgrades can lead to human errors associated with contract upgrades.

Recommended Mitigation: Consider making following changes:

1. Upgrade all vaults in the strategy at once to new implementation
2. In the event there are too many vaults and 1. is not feasible, then emit the vault indices that are upgraded. In addition, instead of passing the vault addresses as input, pass the vault indices as input to the `upgradeVaults` function.

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.2.2 Reward restaking creates unnecessary unbonding in edge cases

Description: The `PolygonStrategy::restakeRewards()` function is publicly callable by anyone, allowing rewards to be restaked immediately before the `PolygonStrategy::unbond()` function is called.

In specific scenarios, this can cause the protocol to unnecessarily unbond principal funds when rewards would have been sufficient to satisfy withdrawal requests.

The issue stems from how rewards are calculated in the `unbond()` function:

```
// In PolygonStrategy::unbond()
uint256 deposits = vault.getTotalDeposits();
uint256 principalDeposits = vault.getPrincipalDeposits();
uint256 rewards = deposits - principalDeposits; //@audit this would be 0 if restakeRewards is called
↪  just before

if (rewards >= toUnbondRemaining) {
```

```
    vault.withdrawRewards();
    toUnbondRemaining = 0;
} else {
    toUnbondRemaining -= rewards;
    uint256 vaultToUnbond = principalDeposits >= toUnbondRemaining
        ? toUnbondRemaining
        : principalDeposits;

    vault.unbond(vaultToUnbond);

    toUnbondRemaining -= vaultToUnbond;
    ++numVaultsUnbonded; //@audit this will be 1 more than necessary
}
```

Meanwhile, the `PolygonStrategy::restakeRewards()` function is publicly accessible:

```
// In PolygonStrategy::restakeRewards()
function restakeRewards(uint256[] calldata _vaultIds) external {
    for (uint256 i = 0; i < _vaultIds.length; ++i) {
        vaults[_vaultIds[i]].restakeRewards();
    }

    emit RestakeRewards();
}
```

Impact: Consider a specific scenario:

- First eligible vault has sufficient rewards to cover an unbonding request completely
- Just before unbond() is called, Alice calls restakeRewards()
- When unbond() executes, it finds no rewards and must unbond principal instead
- This unnecessarily increments numVaultsUnbonding

In this scenario, queued token deposits will face denial of service

Proof of Concept: Add the following:

```
describe.only('restakeRewards forces protocol to wait for withdrawal delay', async () => {
    it('should force protocol to wait for withdrawal delay', async () => {
      const { token, strategy, vaults, fundFlowController, withdrawalPool, validatorShare } =
        await loadFixture(deployFixture)

      await withdrawalPool.setTotalQueuedWithdrawals(toEther(960))
      assert.equal(await fundFlowController.shouldUnbondVaults(), true);

      // 1. Pre-condition: one validator can cover the amount to unbond with his rewards.
      await validatorShare.addReward(vaults[0].target, toEther(1000));

      // 2. User front-runs the unbondVaults transaction and restakes rewards for validators.
      await strategy.restakeRewards([0]);

      const totalQueuedBefore = await strategy.totalQueued();
      // 3. Protocol is forced to unbond and wait for withdrawal delay even though
      // it could obtain the funds after calling vault.withdrawRewards.
      await fundFlowController.unbondVaults()
      const totalQueuedAfter = await strategy.totalQueued();

      assert.equal(await fundFlowController.shouldUnbondVaults(), false)

      // 4. TotalQueued did not increase, meaning strategy could not withdraw any funds
      // and vault is unbending(it wouldn't be if it cover the funds with the rewards).
```

```
      assert.equal(totalQueuedBefore, totalQueuedAfter)
      assert.equal(await vaults[0].isUnbonding(), true)
    })
  })
```

Here is the output:

```
  PolygonFundFlowController
    restakeRewards forces protocol to wait for withdrawal delay
       should force protocol to wait for withdrawal delay (626ms)


  1 passing (628ms)
```

Recommended Mitigation: Consider restricting access to `restakeRewards` in both the strategy and vault contracts.

Stake.Link: Acknowledged. There should never be a significant amount of unclaimed rewards under normal circumstances as they will be claimed every time there is a deposit/unbond.

Cyfrin: Acknowledged.

7.2.3 DoS in unbonding when validator rewards fall below min amount threshold

Description: The unbond function in `PolygonStrategy` is vulnerable to DoS when small reward amounts are used to cover the remaining amount to withdraw.

When PolygonFundFlowController.unbondVaults is called it passes the amount to withdraw/unbond to `PolygonStrategy.unbond`:

```
    function unbondVaults() external {
       ...
        uint256 toWithdraw = queuedWithdrawals - (queuedDeposits + validatorRemovalDeposits);
@>        strategy.unbond(toWithdraw);
        timeOfLastUnbond = uint64(block.timestamp);
    }
```

The `unbond` function loops through validators, using their rewards to cover the withdrawal amount. If rewards are insufficient, it unbonds from the validator's principal deposits.

```
function unbond(uint256 _toUnbond) external onlyFundFlowController {
   ...
    if (rewards >= toUnbondRemaining) {
        // @audit withdrawRewards could be below the threshold from ValidatorShares
@>        vault.withdrawRewards();
        toUnbondRemaining = 0;
    } else {
        toUnbondRemaining -= rewards;
        uint256 vaultToUnbond = principalDeposits >= toUnbondRemaining
            ? toUnbondRemaining
            : principalDeposits;
@>        vault.unbond(vaultToUnbond);
        toUnbondRemaining -= vaultToUnbond;
        ++numVaultsUnbonded;
    }
   ...
}
```

A common, though infrequent, scenario occurs when iteration leaves `toUnbondRemaining` with a small amount for the next validator. This amount may be coverable by the validator's rewards, but if those rewards are below the threshold in the ValidatorShares contract, the transaction will revert, causing a DoS in the `unbond` process.

Example:

1. Strategy has 3 validators.
2. Validator 3 has accumulated 0.9 tokens in rewards
3. The system needs to unbond 30.5 tokens in total
4. Validators 1 and 2 cover 30 tokens with their unbonds
5. Validator 3 is expected to cover the remaining 0.5 tokens with its 0.9 rewards(`withdrawRewards` is triggered)
6. The transaction reverts because 0.9 tokens is less than minAmount from ValidatorShares

The current logic also allows a malicious user to DoS the `unbond` transaction due to the way rewards are calculated in the `unbound` function:

```
uint256 deposits = vault.getTotalDeposits();
uint256 principalDeposits = vault.getPrincipalDeposits();
uint256 rewards = deposits - principalDeposits;
```

The `getTotalDeposits` accounts for the current token balance in the `PolygonVault` contract. If the `toUnbondRemaining` and validator's rewards are less than vault's minimum rewards, an attacker can front-run the transaction and cause `unbond` to revert given the current scenario:

1. Attacker sent dust amount to the `PolygonVault`. Enough to cover `toUnbondRemaining` but < 1e18 (minimum rewards).
2. `rewards = deposits - principalDeposits >= toUnbondRemaining`.
3. Vault will call `withdrawRewards` but current claim amount to be withdrawn < `1e18`(trigger for the revert in ValidatorShares)
4. Transaction reverts.

Impact: DoS in the unbonding process when small rewards (< 1e18) are needed to complete withdrawals, blocking the protocol's withdrawal flow.

Proof of Concept:

1. First adjust the `PolygonValidatorShareMock` to reflect the same behavior as `ValidatorShares` by adding a minimum amount requirement when withdrawing rewards.

```
// PolygonValidatorShareMock
   function withdrawRewardsPOL() external {
        uint256 rewards = liquidRewards[msg.sender];
        if (rewards == 0) revert NoRewards();
+        require(rewards >= 1e18, "Too small rewards amount");

        delete liquidRewards[msg.sender];
        stakeManager.withdraw(msg.sender, rewards);
    }
```

2. Paste the following test in `polygon-fund-flow-controller.test.ts` and `run npx hardhat test test/polygonStaking/polygon-fund-flow-controller.test.ts`:

```
describe.only('DoS', async () => {
    it('will revert when withdrawRewards < 1e18', async () => {
      const { token, strategy, vaults, fundFlowController, withdrawalPool, validatorShare,
      ↪  validatorShare2, validatorShare3 } =
      await loadFixture(deployFixture)
      console.log("will print some stuff")

      // validator funds: [10, 20, 30]
      await withdrawalPool.setTotalQueuedWithdrawals(toEther(970.5));
      assert.equal(await fundFlowController.shouldUnbondVaults(), true);
```

```
    // 1. Pre-condition: validator acumulated dust rewards since last unbonding.
    await validatorShare3.addReward(vaults[2].target, toEther(0.9));

    // expect that vault 2 has rewards
    assert.equal(await vaults[2].getRewards(), toEther(0.9));

    // 2. Unbond:
    // Validator A will cover 10 with unbond
    // Validator B will cover 20 with unbond
    // Validator C will cover the remaining 0.5 with his 0.9 rewards.
    await expect(fundFlowController.unbondVaults()).to.be.revertedWith("Too small rewards amount");
  })
})
```

Output:

```
PolygonFundFlowController
    DoS
       will revert when withdrawRewards < 1e18 (800ms)

  1 passing (800ms)
```

Recommended Mitigation: In the `PolygonStrategy.unbond` check if the **actual** rewards that can be withdrawn from the `ValidatorShares` is greater than the min amount for claim(1e18) before calling `withdrawRewards`.

```
-if (rewards >= toUnbondRemaining) {
-    vault.withdrawRewards();
-    toUnbondRemaining = 0;
+if (rewards >= toUnbondRemaining && vault.getRewards() >= vault.minAmount()) {
+    vault.withdrawRewards();
+    toUnbondRemaining = 0;
} else {
+    if (toUnbondRemaining > rewards) {
+        toUnbondRemaining -= rewards;
+    }
-    toUnbondRemaining -= rewards;
     uint256 vaultToUnbond = principalDeposits >= toUnbondRemaining
         ? toUnbondRemaining
         : principalDeposits;
```

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.2.4 `disableInitializers` not used to prevent uninitialized contracts

Description: `PolygonVault` and `PolygonStrategy` contracts are designed to be upgradeable.

They are both inheriting from `Initializable` but do not have a constructor that calls `_disableInitializers()`.

An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation contract, which may impact the proxy. To prevent the implementation contract from being used, `_disableInitializers()` should be called in the constructor to automatically lock it when it is deployed.

Impact: Missing constructor with `_disableInitializers()` in `PolygonVault` and `PolygonStrategy` risks unauthorized takeover of uninitialized upgradeable contracts.

Recommended Mitigation: Consider adding a constructor to `PolygonVault` and `PolygonStrategy` that explicitly calls `_disableInitializers()`.

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.3 Informational

7.3.1 Missing event emission for PolygonStrategy::setFundFlowController

Description: A crucial strategy parameter ie. the `fundFlowController` can be updated by the owner. However no event emission exists for such an update.

Recommended Mitigation: Consider emitting an event for the `setFundFlowController` function.

Stake.Link: Acknowledged. `fundFlowController` is only ever set once at time of contract deployment

Cyfrin: Acknowledged.

7.3.2 Missing check for `validatorMEVRewardsPercentage` in `PolygonStrategy::initialize`

Description: `validatorMEVRewardsPercentage`is a reward percentage with valid values ranging from 0 to 10000 (100%). `PolygonStrategy::setValidatorMEVRewardsPercentage` rightly checks that the reward percentage does not exceed 10000. However this validation is missing during initialization.

```
    function initialize(
        address _token,
        address _stakingPool,
        address _stakeManager,
        address _vaultImplementation,
        uint256 _validatorMEVRewardsPercentage,
        Fee[] memory _fees
    ) public initializer {
        __Strategy_init(_token, _stakingPool);

        stakeManager = _stakeManager;
        vaultImplementation = _vaultImplementation;
        validatorMEVRewardsPercentage = _validatorMEVRewardsPercentage; //@audit missing check on
        ↪  validatorMEVRewardsPercentage
```

Recommended Mitigation: Consider making the validation checks consistent at every place where the `validatorMEVRewardsPercentage` is updated.

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.3.3 Missing zero address checks

Description: Critical address in the controller, strategy and vault contracts have missing zero address checks, both at the time of initialization and later, at the time of updates.

`PolygonStrategy.sol`

```
    function setValidatorMEVRewardsPool(address _validatorMEVRewardsPool) external onlyOwner {
        validatorMEVRewardsPool = IRewardsPool(_validatorMEVRewardsPool); //@audit missing zero address
        ↪  check
    }

    function setVaultImplementation(address _vaultImplementation) external onlyOwner { //@audit missing
    ↪  zero address check
        vaultImplementation = _vaultImplementation;
        emit SetVaultImplementation(_vaultImplementation);
    }
```

`PolygonFundFlowController.sol`

```
function setDepositController(address _depositController) external onlyOwner {
     depositController = _depositController; // @audit missing zero address check
 }
```

Recommended Mitigation: Consider introducing zero address checks at all places highlighted above.

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.3.4 Incorrect event emission when removing fees via the `PolygonStrategy::updateFee`

Description: Fee for a given index can be removed by passing `_feeBasisPoints = 0` for a given fee index.

Current logic is swapping the fee at the index with that of the last index in the `fees` array, and then popping the last element of the `fees` array.

However, the event emitted for this action suggests that the fee at the index is updated to 0. This is incorrect because the fee at the index is the fee corresponding to the last index of the original `fees` array. Additionally, in this scenario, the `UpdateFee` event is emitting the `receiver` passed as an input to the function without verifying if this receiver indeed matches the receiver for the removed fee index.

```
function updateFee(
        uint256 _index,
        address _receiver,
        uint256 _feeBasisPoints
    ) external onlyOwner {

        if (_feeBasisPoints == 0) {
            fees[_index] = fees[fees.length - 1];
            fees.pop();

        } else {
              // ... code
        }
        emit UpdateFee(_index, _receiver, _feeBasisPoints); //@audit if _feeBasisPoints == 0, the fee
        ↪  for the index is now the fee in the last index (for old array)
    }
```

Recommended Mitigation: Consider adding an event `RemoveFee` similar to `AddFee` for the special case when `_feeBasisPoints == 0`. This will clearly separate the scenario where fee at an index is just updated v/s removed altogether.

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.3.5 Missing check for `_feeBasisPoints > 0` in `PolygonStrategy::addFee`

Description: `PolygonStrategy::addFee` does not check if `_feeBasisPoints > 0` when adding a new fee element to the `fees` array. This is inconsistent with the `PolygonStrategy::updateFee` logic where `_feeBasisPoints = 0` triggers a removal of the fee element from the `fees` array.

Recommended Mitigation: Consider introducing a non-zero check to `_feeBasisPoints`.

Stake.Link: Acknowledged. Owner will ensure the correct value is set.

Cyfrin: Acknowledged.

7.3.6 No slippage protection when interacting with `ValidatorShares`

Description: In the `PolygonVault.sol` contract, the `buyVoucherPOL` and `sellVoucherPOL` functions do not set a slippage limit.

```
// PolygonVault
    function deposit(uint256 _amount) external onlyVaultController {
        token.safeTransferFrom(msg.sender, address(this), _amount);

        // @audit-issue - no slippage limit.
@>        validatorPool.buyVoucherPOL(_amount, 0);

        uint256 balance = token.balanceOf(address(this));
        if (balance != 0) token.safeTransfer(msg.sender, balance);
    }

   function unbond(uint256 _amount) external onlyVaultController {
         // @audit-issue no slippage limit
@>         validatorPool.sellVoucherPOL(_amount, type(uint256).max);

        uint256 balance = token.balanceOf(address(this));
        if (balance != 0) token.safeTransfer(msg.sender, balance);
    }
```

Currently, there is no active slashing in `ValidatorShares`, therefore loss of funds cannot occur.

Recommended Mitigation: Keep monitoring Polygon PoS governance for slashing updates; if implemented, upgrade the contract to support slashing accordingly.

[Project]: Acknowledged. Will implement slippage protection if slashing is introduced.

Cyfrin: Acknowledged.

7.3.7 Missing input validation in `PolygonFundFlowController.setMinTimeBetweenUnbonding`

Description: The `setMinTimeBetweenUnbonding` function from `PolygonFundFlowController` lacks input validation for the `_minTimeBetweenUnbonding` parameter, allowing it to be set to any value including zero or extremely high values.

```
    function setMinTimeBetweenUnbonding(uint64 _minTimeBetweenUnbonding) external onlyOwner {
        minTimeBetweenUnbonding = _minTimeBetweenUnbonding;  // @audit missing input validation
        emit SetMinTimeBetweenUnbonding(_minTimeBetweenUnbonding);
    }
```

Recommended Mitigation: Add a min/max value check before setting the `minTimeBetweenUnbonding`. I.e:

```
// declare MIN_VALUE and MAX_VALUE with the proper values.
function setMinTimeBetweenUnbonding(uint64 _minTimeBetweenUnbonding) external onlyOwner {
+    require(_minTimeBetweenUnbonding >= MIN_VALUE, "Time between unbonding too low");
+    require(_minTimeBetweenUnbonding <= MAX_VALUE, "Time between unbonding too high");

    minTimeBetweenUnbonding = _minTimeBetweenUnbonding;
    emit SetMinTimeBetweenUnbonding(_minTimeBetweenUnbonding);
}
```

Stake.Link: Acknowledged. Owner will ensure correct value is set.

Cyfrin: Acknowledged.

7.3.8 Potential infinite loop in `PolygonStrategy::unbond` due to insufficient balance check

Description: `PolygonStrategy::unbond` function contains a while loop that continues until the entire requested unbonding amount (`toUnbondRemaining`) is processed. However, there is no safeguard to handle a scenario where the total available balance across all vaults is insufficient to satisfy the unbonding request.

```
function unbond(uint256 _toUnbond) external onlyFundFlowController {
    // ...
    uint256 toUnbondRemaining = _toUnbond;

    // ...
    while (toUnbondRemaining != 0) {
        // Process vaults...
        ++i;
        if (i >= vaults.length) i = 0;
        // No check for complete loop iteration without progress
    }
    // ...
}
```

The issue arises because the function assumes that the unbonding amount will always be covered by the total staked amount across all vaults. While this assumption might hold in the current single-strategy setup, it is not guaranteed, especially in a multi-strategy environment.

It is noteworthy that the withdrawal pool, which keeps track of `queuedWithdrawals` that determine the magnitude of `unbonding`, operates at a global level across all strategies.

Impact: Although an unlikely scenario for the current single strategy setup, an infinite while loop consumes all available gas.

Recommended Mitigation: Consider adding a safety mechanism to detect when the loop has iterated through all vaults without making progress, indicating insufficient funds to satisfy the unbonding request.

```
function unbond(uint256 _toUnbond) external onlyFundFlowController {

        while (toUnbondRemaining != 0) {
             // ... code
             if (i >= vaults.length) i = 0;

++       // Add safety check to prevent infinite loop
++        if (i == startingIndex) {
++            // We've gone through all vaults and still have amount to unbond
++            // Process partial unbonding with what we've got so far OR revert
++           break; // or revert if that's more appropriate
++        }

       }
}
```

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.4 Gas Optimization

7.4.1 Validate total fees in `PolygonStrategy::updateFee` only if the `_feeBasisPoints > 0`

Description: While updating the fee in `PolygonStrategy::updateFee`, if `_feeBasisPoints == 0`, the fee at the index is replaced with the fee at the last index of the array, and the last element is popped from the array.

Since fees is a non-negative value, there is no need to perform the total fee check in this scenario. It is worth highlighting that the check is gas heavy as it involves looping over a fee array.

```
if (_totalFeesBasisPoints() > 3000) revert FeesTooLarge();
```

Recommended Mitigation: Consider moving the check into the `else` block.

Stake.Link: Resolved in PR 151

Cyfrin: Resolved.

7.4.2 Inefficient rewards collection pattern in the `PolygonStrategy::unbond()` function leads to unnecessary gas consumption

Description: The current implementation of the `PolygonStrategy::unbond()` function processes vaults sequentially and makes immediate unbonding decisions on a per-vault basis.

This approach can lead to unnecessary unbonding operations when the total rewards across all vaults would be sufficient to cover the withdrawal amount.

In the current implementation:

1. The function iterates through each vault
2. For each vault, it extracts rewards if available
3. If rewards from a single vault are insufficient, it immediately unbonds principal from that vault
4. It continues this process until the full `_toUnbond` amount is satisfied

This implementation doesn't consider the total available rewards across all vaults before making unbonding decisions, resulting in potential unnecessary unbonding operations that consume significant gas.

Impact:

- Higher gas costs: Each unnecessary unbonding operation consumes additional gas, especially since it later requires a separate unstakeClaim call to finalize the withdrawal
- Capital inefficiency: Unbonding principal when rewards would have been sufficient reduces the amount of capital earning yields in the protocol
- Longer waiting periods: Unlike rewards, unbonded principal is subject to waiting periods before it can be re-used or withdrawn

Recommended Mitigation: Refactor the unbond() function to use a two-phase approach:

1. Collect all rewards until the cumulative rewards are enough to honor unbonding request
2. Start vault unbonding only after all rewards are collected.

Here is a sample implementation:

```
function unbond(uint256 _toUnbond) external onlyFundFlowController {
    if (numVaultsUnbonding != 0) revert UnbondingInProgress();
    if (_toUnbond == 0) revert InvalidAmount();

    uint256 toUnbondRemaining = _toUnbond;
    uint256 preBalance = token.balanceOf(address(this));
    uint256 skipIndex = validatorRemoval.isActive ? validatorRemoval.validatorId : type(uint256).max;
    uint256 numVaultsUnbonded;
```

```
    // @audit Phase 1 -> Withdraw all rewards first
    for (uint256 i = 0; i < vaults.length; i++) {
        if (i != skipIndex) {
            IPolygonVault vault = vaults[i];
            uint256 rewards = vault.getRewards();

            if (rewards > 0) {
                vault.withdrawRewards();

                // @audit Check if we've collected enough rewards
                uint256 currentBalance = token.balanceOf(address(this));
                uint256 collectedRewards = currentBalance - preBalance;
                if (collectedRewards >= toUnbondRemaining) {
                    // @audit We've collected enough rewards, no need to unbond principal
                    totalQueued += collectedRewards;
                    emit Unbond(_toUnbond);
                    return;
                }
            }
        }
    }

    // @audit Phase 2: If rewards weren't enough, unbond principal
    uint256 rewardsCollected = token.balanceOf(address(this)) - preBalance;
    toUnbondRemaining -= rewardsCollected;

    uint256 i = validatorWithdrawalIndex;
    while (toUnbondRemaining != 0) {
        if (i != skipIndex) {
            IPolygonVault vault = vaults[i];
            uint256 principalDeposits = vault.getPrincipalDeposits();

            if (principalDeposits != 0) {
                uint256 vaultToUnbond = principalDeposits >= toUnbondRemaining
                    ? toUnbondRemaining
                    : principalDeposits;

                vault.unbond(vaultToUnbond);
                toUnbondRemaining -= vaultToUnbond;
                ++numVaultsUnbonded;
            }
        }

        ++i;
        if (i >= vaults.length) i = 0;
    }

    validatorWithdrawalIndex = i;
    numVaultsUnbonding = numVaultsUnbonded;
    totalQueued += token.balanceOf(address(this)) - preBalance;

    emit Unbond(_toUnbond);
}
```

Stake.Link: Acknowledged.

Cyfrin: Acknowledged.