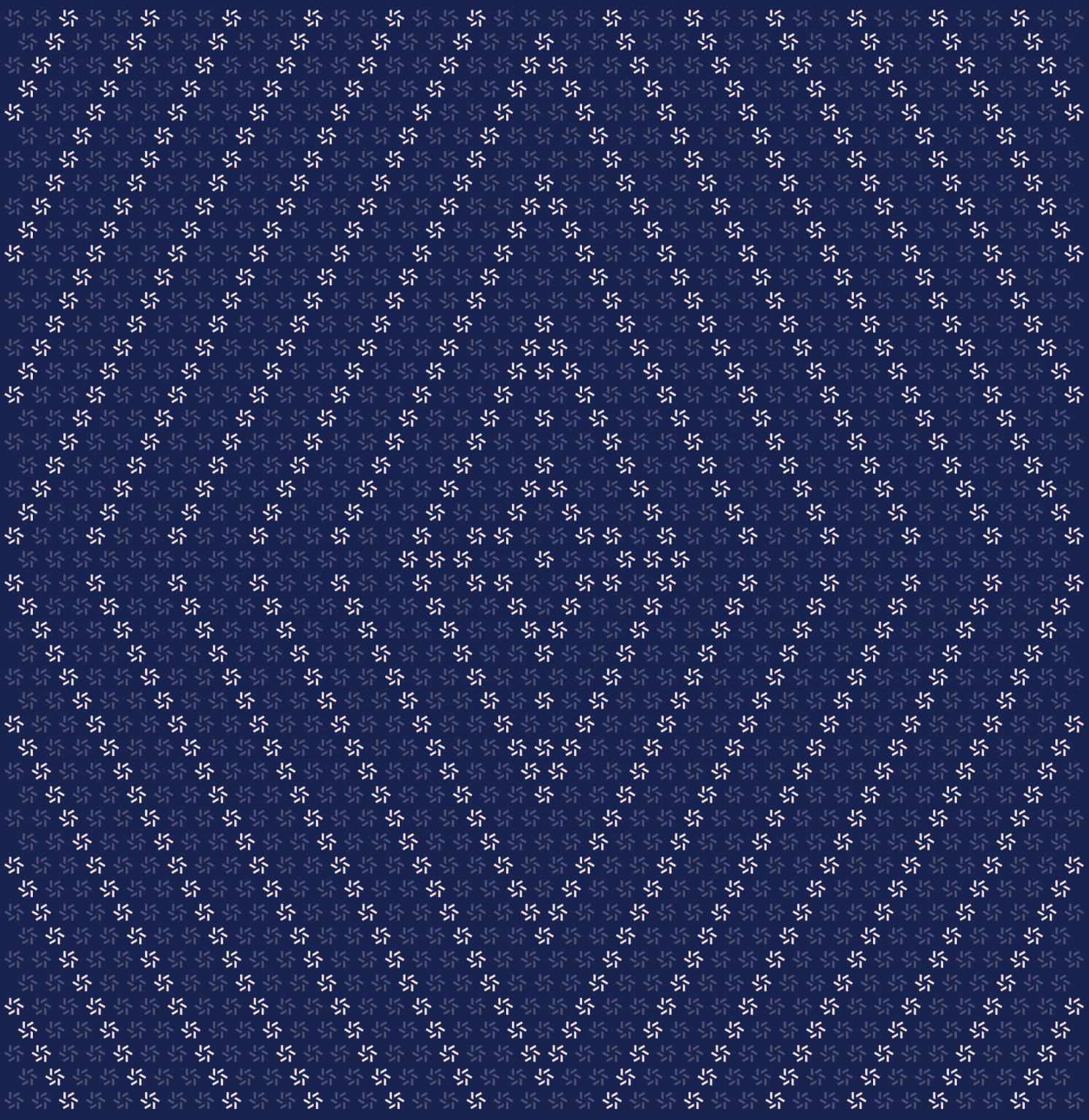
Zellic

Prepared for
Jonny Huxtable
Stake.link

Prepared by
Juchang Lee
Kuilin Li
Zellic

July 2, 2025

Polygon Staking
Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Stake.link from June 20th to June 26th, 2025. During this engagement, Zellic reviewed Polygon Staking's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the process of splitting rewards working correctly?
- Are rewards and fees calculated correctly?
- Could an on-chain attacker drain the staking vaults?
- How does the strategy generate the rewards it provides to its users?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Contracts not listed in the scope
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

This audit assumes that the upstream ValidatorShare implementation that the PolygonVault assigns to `validatorPool` is commit 5b7e40e5 ↗ on the 0xPolygon repository.

1.4. Results

During our assessment on the scoped Polygon Staking contracts, we discovered 13 findings. No critical issues were found. One finding was of high impact, one was of medium impact, four were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

Stake.link in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	4
Informational	7

2. Introduction

2.1. About Polygon Staking

Stake.link contributed the following description of Polygon Staking:

> stake.link is the first of its kind delegated liquid staking protocol for Chainlink Staking. Powered and governed by the protocol token SDL, with DeFi interoperability enabled by the liquid staking receipt token stLINK, the stake.link protocol enables anyone to provide LINK collateral to and receive a share of rewards from the most reliable and performant Chainlink node operators.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Polygon Staking Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts
Repository	https://github.com/stakedotlink/contracts ↗
Version	`00b568399854dba200a54cb16b0c337ddc6078f0`

Programs	`PolygonFundFlowController.sol`
	`PolygonVault.sol`
	`PolygonStrategy.sol`

Target	contracts
Repository	https://github.com/stakedotlink/contracts ↗
Version	`4c35b081f887c9a42c54408ab08b506ea3dc633e`
Programs	`LSTRewardsSplitter.sol` `LSTRewardsSplitterController.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Juchang Lee
Engineer
lee@zellic.io

Kuilin Li
Engineer
kuilin@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 20, 2025	Start of primary review period
June 24, 2025	Kick-off call
June 26, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect minimum unbond quantity calculation

Target	PolygonStrategy		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In the main loop inside the `unbond` function, the PolygonStrategy contract checks to ensure that the quantity it will claim as rewards from the external staking contract is above the minimum claim amount:

```
function unbond(uint256 _toUnbond) external onlyFundFlowController {
    // [...]
    while (toUnbondRemaining != 0) {
        // [...]
        IPolygonVault vault = vaults[i];
        uint256 deposits = vault.getTotalDeposits();

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

            if (rewards >= toUnbondRemaining && rewards
    >= vault.minRewardClaimAmount()) {
                vault.withdrawRewards();
                toUnbondRemaining = 0;
                break;
            } // [...]
```

The above code assumes that if the comparison `rewards >= vault.minRewardClaimAmount()` passes, then calling `vault.withdrawRewards` will not revert.

However, the `rewards` variable is set to `deposits - principalDeposits`. The `getTotalDeposits` function in the PolygonVault contract has the following implementation:

```
function getTotalDeposits() public view returns (uint256) {
    return
        getPrincipalDeposits() +
        getRewards() +
        getQueuedWithdrawals() +
```

```
        token.balanceOf(address(this));
}
// [...]
```

There are four components to this calculation:

1. The second call to `getPrincipalDeposits()` likely returns the same quantity as the first one, so that subtraction cancels out. Unless there is an upgrade to the upstream vault, there is no reentrancy site for it to have changed.
2. The call to `getQueuedWithdrawals()` will return zero, because the vault will not be unbonding at this time, unless upgrades made to the upstream vault break the invariant implemented, `numVaultsUnbonding`.
3. The call to `getRewards()` will return the rewards quantity that is the same one used by the upstream's minimum check.
4. The call to `token.balanceOf`, however, is a free parameter and can be arbitrarily inflated by the attacker.

In the upstream vault implementation ↗, the implementation of this minimum is this:

```
function _withdrawRewards(bool pol) internal {
    uint256 rewards = _withdrawAndTransferReward(msg.sender, pol);
    require(rewards >= minAmount, "Too small rewards amount");
}

// [...]

function _withdrawAndTransferReward(address user, bool pol)
    private returns (uint256) {
    uint256 liquidRewards = _withdrawReward(user);
    // [...]
    return liquidRewards;
}

// [...]

function _withdrawReward(address user) private returns (uint256) {
    // [...]
    uint256 liquidRewards = _calculateReward(user, _rewardPerShare);
    // [...]
    return liquidRewards;
}
```

So the call to `token.balanceOf` should be excluded from this check, because the token balance is not considered in the upstream's minimum check.

Impact

An attacker can cause some calls to `unbond` to revert.

More specifically, if a call to `unbond` considers skipping the withdrawal of rewards from a vault whose rewards are too little to claim on the upstream, then an attacker who is positioned to front-run `unbond` can donate tokens to the vault in order to meet that minimum. This will cause the reward claim to be attempted and therefore revert the original call to `unbond`.

This revert is inconvenient because the same `unbond` cannot be attempted again, since it will revert again. The only fixes to this state would be 1) waiting for enough rewards to accrue naturally so that the withdrawal of rewards succeeds or 2) manual intervention by governance.

Recommendations

Correct this calculation by calling `getRewards`, which calls `getLiquidRewards`, directly to check this threshold.

In the upstream's implementation, the `getLiquidRewards` function directly returns the result of `_calculateReward`, which is the same quantity used by the minimum check, as seen above.

```
function getLiquidRewards(address user) public view returns (uint256) {
    return _calculateReward(user, getRewardPerShare());
}
```

Additionally, in line with Finding 3.2. ↗, we recommend separately handling the three steps of a vault unbond — withdrawing the standing token balance, claiming the rewards, and unstaking the claim.

Remediation

This issue has been acknowledged by Stake.link, and a fix was implemented in commit `ce454ac1` ↗.

3.2. The rewards quantity is not deducted if the unbond quantity is lesser

Target	PolygonStrategy		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

In the `unbond` function, this logic handles deciding what to do with the vault that is next in line for unbonding — specifically, whether to skip it, withdraw rewards from it, or unbond it:

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

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

    if (rewards >= toUnbondRemaining && rewards
    >= vault.minRewardClaimAmount()) {
        vault.withdrawRewards();
        toUnbondRemaining = 0;
        break;
    } else if (principalDeposits != 0) {
        if (toUnbondRemaining > rewards) {
            toUnbondRemaining -= rewards;
        }

        uint256 vaultToUnbond = principalDeposits >= toUnbondRemaining
            ? toUnbondRemaining
            : principalDeposits;

        vault.unbond(vaultToUnbond);

        toUnbondRemaining -= vaultToUnbond;
        ++numVaultsUnbonded;
    }
}
```

The `if (toUnbondRemaining > rewards) {` statement runs in two different cases, because of the previous conditional. In the case where the previous conditional fails due to `rewards >=`

`toUnbondRemaining` being false, then the `toUnbondRemaining -= rewards;` line always runs, which is appropriate. But in the case where the previous conditional fails due to the `minRewardClaimAmount` check, this logic does not deduct `rewards` from the quantity to be unbonded.

This is a misaccounting, because according to the upstream vault implementation, when the vault is unbonded, the rewards are claimed. So the `unbond` function will unbond more funds than necessary.

Impact

The `unbond` function will unbond more funds than necessary when satisfying an unbond request. This will slightly decrease the rewards rate and also cause the vaults that are next in line to be unduly unbonded earlier than needed.

Recommendations

We recommend separating this logic into three stages when calculating whether to withdraw or unbond a vault.

First, the vault's standing balance should be checked. These are tokens that are not staked to the underlying vault and therefore are freely transferrable back to the PolygonStrategy to be queued. We recommend unconditionally withdrawing and requeueing these funds, since these funds will typically only be the result of attackers directly donating vault shares into the vault (a donation of any amount will claim rewards into the vault), and they should be accounted and requeued immediately. Even if the `toUnbondRemaining` is less, it should still claim the full amount, and then exit.

Secondly, the vault's withdrawable rewards should be checked. These are the unclaimed rewards that can only be directly withdrawn if the amount is greater than the minimum. We did not analyze the benefits or drawbacks of immediately unconditionally withdrawing the rewards here, but a decision should be made whether to withdraw them.

Finally, the vault's principal deposits should be checked. If the previous step did not withdraw the rewards for any reason (e.g., the quantity is below the minimum, or it is not economical to withdraw it), then the amount of those rewards should be added to the amount we will get if we unbond any quantity of principal, since the logic in the upstream contract implements it like that.

In between each of these three steps, the `toUnbondRemaining` should be reduced either to zero or by the amount that is released.

Remediation

This issue has been acknowledged by Stake.link, and a fix was implemented in commit `ce454ac1` ↗.

3.3. Lack of maximum-fee check in the constructor

Target	LSTRewardsSplitter		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Currently, there is no maximum-value check for the fee in the constructor:

```
constructor(address _lst, Fee[] memory _fees, address _owner) {
    controller = ILSTRewardsSplitterController(msg.sender);
    lst = IERC677(_lst);
    for (uint256 i = 0; i < _fees.length; ++i) {
        fees.push(_fees[i]);
    }
    _transferOwnership(_owner);
}

function addFee(address _receiver, uint256 _feeBasisPoints)
    external onlyOwner {
    // [...]
    if (_totalFeesBasisPoints() > 10000) revert FeesExceedLimit();
}

function updateFee(
    uint256 _index,
    address _receiver,
    uint256 _feeBasisPoints
) external onlyOwner {
    // [...]
    if (_totalFeesBasisPoints() > 10000) revert FeesExceedLimit();
}
```

Since the 10000 threshold is not checked in the constructor, arbitrarily large amounts can be assigned to the fee at that time. If the assigned fees are so large that removing one fee does not fix the problem, then this permanently bricks fee updates, because no single call to `addFee` or `updateFee` will be able to satisfy the threshold to not revert with `FeesExceedLimit`.

Impact

Incorrect contract construction can lead to unintended and excessive fees charged. In extreme cases, these cannot be fixed except by upgrading the vault.

Recommendations

Add a `_totalFeesBasisPoints() > 10000` check to the constructor.

Remediation

This issue has been acknowledged by Stake.link, and a fix was implemented in commit `429503fd` ↗.

3.4. Adding too many splitters can cause a denial of service

Target	LSTRewardsSplitterController		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `addSplitter` function is an `onlyOwner` function that adds a splitter to the contract's list:

```
function addSplitter(
    address _account,
    LSTRewardsSplitter.Fee[] memory _fees
) external onlyOwner {
    if (address(splitters[_account]) != address(0))
    revert SplitterAlreadyExists();

    address splitter = address(new LSTRewardsSplitter(lst, _fees, owner()));
    splitters[_account] = ILSTRewardsSplitter(splitter);
    accounts.push(_account);
    IERC677(lst).safeApprove(splitter, type(uint256).max);
}
```

This function runs in constant time, so it can be called an unlimited number of times by the owner, no matter how many items are in the `accounts` array.

But if it is called too many times, then it may be impossible to perform upkeep or use this contract because the gas fees required to iterate through the `accounts` array is too high.

Impact

Unexpected out-of-gas reverts can occur if too many splitters are added.

Note that although `removeSplitter` does have a loop that loops through `accounts`, this loop exits early if the target account is found:

```
uint256 numAccounts = accounts.length;
for (uint256 i = 0; i < numAccounts; ++i) {
    if (accounts[i] == _account) {
        accounts[i] = accounts[numAccounts - 1];
        accounts.pop();
```

```
            break;
        }
    }
```

Thus, removing the first account will always take a constant amount of gas. So, if this brick occurs, then it can be resolved by removing and re-adding splitters that are earlier in the array.

Recommendations

We recommend hardcoding a reasonable upper limit on the number of splitters, in order to fail noisily and early if it is reached. This is better because it is safer to fail on the addition of a new splitter than it is at any other part of the business logic.

If the above recommendation is taken, then we additionally recommend adding tests that reach that limit so that the gas costs at the limit can be monitored through any future development.

Remediation

This issue has been acknowledged by Stake.link. In addition Stake.link has provided the following response:

> We will ensure that the number of splitters will always be small.

3.5. Donations can arbitrarily inflate fees to skip paying any fees via an upstream safety check

Target	PolygonStrategy		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `updateDeposits` function is called by the staking pool to update the amount of accrued fees based on the profit and loss sustained by the vaults. It calls `getDepositChange` to calculate the difference between the current total balance across all the vaults and the `totalDeposits`, which is the baseline amount of assets that the vault is expected to have — any assets above that amount are profits that are taxed by the fees, and any assets below that amount are losses that do not incur fees.

The `getDepositChange` function takes into account the current balance of the contract as well as the balances of all of the vaults:

```
function getDepositChange() public view returns (int) {
    uint256 totalBalance = token.balanceOf(address(this));

    for (uint256 i = 0; i < vaults.length; ++i) {
        totalBalance += vaults[i].getTotalDeposits();
    }
    return int(totalBalance) - int(totalDeposits);
}
```

So, if any third party donates assets, either in the form of the underlying token sent to the PolygonStrategy (or any vault) or in the form of staked tokens sent to any vault, those assets will be accounted as profits and increase the amount of fees returned by `updateDeposits`.

The caller of `updateDeposits` is the `_updateStrategyRewards` function in the StakingPool contract in the core directory. In this contract, there is a safety check that skips sending any of the fees to anyone if the amount is unexpectedly high:

```
function _updateStrategyRewards(uint256[] memory _strategyIdxs,
    bytes memory _data) private {
    // [...]

    // sum up rewards and fees across strategies
```

```
    // [...]

    // update totalStaked if there was a net change in deposits
    // [...]

    // calulate fees if net positive rewards were earned
    // [...]

    // safety check
    if (totalFeeAmounts >= totalStaked) {
        totalFeeAmounts = 0;
    }

    // distribute fees to receivers if there are any
    if (totalFeeAmounts > 0) {
        uint256 sharesToMint = (totalFeeAmounts * totalShares) /
            (totalStaked - totalFeeAmounts);
        _mintShares(address(this), sharesToMint);

        // [...]
    }

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

It is important to note that the recipients of the fees are the designated `receiver` addresses of the `fees` array, which is only modifiable by the owner. These recipients are not necessarily the same parties as the depositors to the strategy or the validator MEV recipients.

Impact

This safety check may incentivize abnormal profit-seeking behavior, depending on the parameters and the ability for various parties to collude. For example, if the addresses in the `fees` array are completely separate from the depositors to the strategy, and the validator MEV recipient's share is zero or insignificant, then the depositors may be able to collude in order to artificially greatly inflate the profits seen by the strategy, to trigger this edge case. The extra value they donate will be entirely returned to them as profits, since the fees are reduced to zero.

Recommendations

Without analyzing the rest of the protocol and broader market, we have no specific recommendation on how this edge case may be prevented. We recommend designing the fee structure in a way that is safe against the profit-seeking collusion of any set of actors in this system, and that may require a safety check in this contract that sets aside excess profits or caps the fees

to ensure that this core safety-check condition is never triggered.

Remediation

This issue was present and documented in a previous audit of this project, and it was acknowledged by Stake.link. It is written here for completeness only.

3.6. Centralization risk of upgradability.

Target	PolygonFundFlowController, PolygonVault		
Category	Protocol Risks	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Note that PolygonVault and PolygonFundFlowController are upgradable.

Impact

While upgradability is a useful, common feature, it also introduces centralization risks. A malicious owner could change the implementation of the contracts to steal funds, disable the contract, or otherwise change the behavior of the contract in a way that is not in the interest of the users.

Recommendations

We recommend the owner of the contracts should be a governance contract or multi-sig. Otherwise, users must be aware of and accept the centralization risk.

Remediation

This issue has been acknowledged by Stake.link. In addition Stake.link has provided the following response:

> We will ensure that the owner will be a multi-sig.

3.7. Hardcoded rate precision

Target	PolygonVault		
Category	Protocol Risks	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `getPrincipalDeposits` function in the PolygonVault contract uses the exchange rate to get the current rebasing amount that the principal deposit shares are worth:

```
function getPrincipalDeposits() public view returns (uint256) {
    return
        (validatorPool.balanceOf(address(this))
    * validatorPool.exchangeRate()) /
        _getRatePrecision();
}

// [...]

function _getRatePrecision() private view returns (uint256) {
    uint256 validatorId = validatorPool.validatorId();

    // if foundation validator, use old precision
    if (validatorId < 8) {
        return EXCHANGE_RATE_PRECISION;
    }

    return EXCHANGE_RATE_HIGH_PRECISION;
}
```

The reimplementation of the `_getRatePrecision` function is copied from the upstream ValidatorShare contract's implementation. This means that this PolygonVault implementation assumes that the rate position will never change, which is not a feature that the upstream contract reasonably guarantees.

Impact

There is no current impact at the time of writing. However, if the implementation of Valuableshare changes in the future, the assumptions about the ratio position may be broken.

Recommendations

We recommend calling `getTotalStake` on the ValidatorShare contract in order to rely on it to calculate the exchange rate with the correct precision.

The current implemented version of `getTotalStake` matches the above calculation:

```
function getTotalStake(address user) public view returns (uint256, uint256) {
    uint256 shares = balanceOf(user);
    uint256 rate = exchangeRate();
    if (shares == 0) {
        return (0, rate);
    }

    return (rate.mul(shares).div(_getRatePrecision()), rate);
}
```

So, as of now, there would be no difference between the two implementations. But in the future, if the ValidatorShare implementation changes, then calling the public view function to get the total stake is safer than making internal assumptions about the rate precision.

Remediation

This issue has been acknowledged by Stake.link, and a fix was implemented in commit `1aee7ed0` ↗.

3.8. Rewards remain unclaimed when `principalDeposits` is zero

Target	PolygonStrategy		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

When `queueValidatorRemoval` runs on a vault with `principalDeposits` equal to zero but containing rewards, these rewards remain unclaimed and are lost in accounting because they are not added to the second entry of `ValidatorRemoval`:

```
function queueValidatorRemoval(uint256 _validatorId) external onlyOwner {
    // [...]
    if (principalDeposits != 0) {
        uint256 preBalance = token.balanceOf(address(this));
        vault.unbond(principalDeposits);
        uint256 rewardsClaimed = token.balanceOf(address(this)) - preBalance;
        if (rewardsClaimed != 0) totalQueued += rewardsClaimed;
    }
    // [... remaining logic does not transfer vault token balance]
}
```

The finalization of the validator removal also does not withdraw the tokens held by the vault. So those tokens are lost during this process.

Impact

The impact is minimal. We identified two ways a token balance can unexpectedly exist that is owned by the PolygonVault contract.

The first case is if a user directly donates tokens to the contract. Although it would maximize yield to transfer such tokens, it is reasonable to not bother collecting them.

The second case is if a user donates a small quantity of shares to the contract. The implementation of the upstream ValidatorShare contract has a transfer hook that claims the rewards on behalf of both the sender and the recipient:

```
function _transfer(address to, uint256 value, bool pol) internal {
    address from = msg.sender;
    // get rewards for recipient
```

```
    _withdrawAndTransferReward(to, pol);
    // convert rewards to shares
    _withdrawAndTransferReward(from, pol);
    // move shares to recipient
    super._transfer(from, to, value);
    _getOrCacheEventsHub().logSharesTransfer(validatorId, from, to, value);
}
```

This means if an attacker sends an insignificant amount of shares to the PolygonVault, it will claim the rewards and send them to the vault. However, all code paths that lead to a `principalDeposits` of zero include claiming the existing rewards and transferring the standing balance back to the PolygonStrategy. So the only rewards that may accrue are from donated shares during the validator-removal process, which it is also reasonable to not collect.

Recommendations

We recommend adding a check in the `finalizeValidatorRemoval` function that withdraws and queues any standing token balance in the PolygonVault contract, even if `getQueuedWithdrawals` is zero.

Remediation

This issue has been acknowledged by Stake.link, and a fix was implemented in commit a675197e ↗.

3.9. No length check is present on the `amounts` parameter

Target	PolygonStrategy		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `depositQueuedTokens` and `forceUnbond` functions both take in an `_amounts` array that is expected to match the length of the `_vaultIds` array:

```
function depositQueuedTokens(
    uint256[] calldata _vaultIds,
    uint256[] calldata _amounts
) external onlyFundFlowController {
    // [...]
    for (uint256 i = 0; i < _vaultIds.length; ++i) {
        if (_vaultIds[i] == skipIndex) revert InvalidVaultIds();
        uint256 amount = _amounts[i];
        if (amount == 0) revert InvalidAmount();
        vaults[_vaultIds[i]].deposit(amount);
    }
    // [...]
}

function forceUnbond(
    uint256[] calldata _vaultIds,
    uint256[] calldata _amounts
) external onlyFundFlowController {
    // [...]

    for (uint256 i = 0; i < _vaultIds.length; ++i) {
        if (_vaultIds[i] == skipIndex) revert InvalidVaultIds();
        if (i > 0 && _vaultIds[i] <= _vaultIds[i - 1])
    revert InvalidVaultIds();
        if (_amounts[i] == 0) revert InvalidAmount();

        vaults[_vaultIds[i]].unbond(_amounts[i]);
        totalUnbonded += _amounts[i];
    }
    // [...]
}
```

However, there is no length check in either function.

Impact

If the `_amounts` parameter is longer than the `_vaultIds` parameter, then the extra entries will be disregarded. If it is shorter, then the above code will revert due to the array's bounds check. If the lengths do not match, then the calldata is malformed.

Recommendations

We recommend adding a length check to both these functions so that the erroneous case where the `_amounts` parameter is longer than the `_vaultIds` parameter fails noisily instead of quietly succeeding.

Remediation

This issue has been acknowledged by Stake.link, and a fix was implemented in commit `337298aa` ↗.

3.10. The `totalStaked` function name could be confusing

Target	PolygonStrategy		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function name `totalStaked` is used in many other places in the upstream contracts, and all of those other functions define it to mean the total staked quantity — for example, in the SecurityPool contract:

```
/**
 * @notice returns the total staked amount for use by the rewards pool
 * controlled by this contract
 * @dev shares are used so this contract can rebase without affecting rewards
 * @return total staked amount
 */
function totalStaked() external view override returns (uint256) {
    return totalShares;
}
```

However, the `totalStaked` defined in PolygonStrategy returns the total number of active validators:

```
/**
    * @notice Returns the total number of active validators
    * @dev used by the validator MEV rewards pool
    * @return total number of active validators
*/
function totalStaked() public view returns (uint256) {
    uint256 totalValidators = validators.length;
    if (validatorRemoval.isActive) --totalValidators;
    return totalValidators;
}
```

Although this is also a `uint256`, it is a number with fundamentally different meaning and units. So, if it is mistakenly called under the assumption that it returns the quantity that all the other `totalStaked` view functions return, it can lead to developer confusion and errors.

Impact

These inconsistencies can confuse subsequent code reviews or revisions.

Recommendations

We recommend changing this function name to `totalValidators`.

Remediation

This issue has been acknowledged by Stake.link. In addition Stake.link has provided the following response:

> Naming is used to conform to the interface expected by the rewards pool.

3.11. Gas optimization in the `removeSplitter` function

Target	LSTRewardsSplitterController		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `removeSplitter` function removes a splitter. Since the splitters are recorded in an array, it loops through the array in order to locate the splitter and then replaces it with the last splitter:

```
function removeSplitter(address _account) external onlyOwner {
    // [...]
    uint256 numAccounts = accounts.length;
    for (uint256 i = 0; i < numAccounts; ++i) {
        if (accounts[i] == _account) {
            accounts[i] = accounts[numAccounts - 1];
            accounts.pop();
            break;
        }
    }
    // [...]
}
```

As noted in Finding 3.4. ↗, this costs a variable amount of gas and could be prohibitively expensive if too many splitters are added.

Since the index of the splitter is known to the caller and more easily computable off chain, this function could take the index as a hint or alternatively just take the index. That will make it run in constant time.

Impact

This requires more gas than necessary.

Recommendations

We recommend either changing `removeSplitter` to identify the account by index instead of address or adding an additional parameter that serves as a hint for the index. If the latter option is taken, then the function should either revert or do the current loop if the hint is incorrect. This will

save gas if there is more than a threshold amount of accounts — the exact threshold depends on the compilation parameters.

Remediation

This issue has been acknowledged by Stake.link. In addition Stake.link has provided the following response:

> We will ensure that the number of splitters will always be small.

3.12. Renumbered vault/validator IDs

Target	PolygonStrategy		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `finalizeValidatorRemoval` function removes a validator by moving every validator after it one slot backwards in the `validators` array:

```
function finalizeValidatorRemoval() external onlyOwner {
    // [...]

    for (uint256 i = validatorId; i < validators.length - 1; ++i) {
        validators[i] = validators[i + 1];
        vaults[i] = vaults[i + 1];
    }

    validators.pop();
    vaults.pop();

    // [...]
}
```

This means that whenever `finalizeValidatorRemoval` is called by the owner at the same time as another call that references a validator by ID, and the referenced validator is after the validator that is being removed, then the block builder will have control over which validator is actually referenced by the other call.

Here is a list of functions that reference a vault or validator by ID:

- `depositQueuedTokens(_vaultIds, _amounts)`
- `forceUnbond(_vaultIds, _amounts)`
- `unstakeClaim(_vaultIds)`
- `restakeRewards(_vaultIds)`
- `queueValidatorRemoval(_validatorId)`

The `depositQueuedTokens` and `forceUnbond` functions are only callable by the deposit controller through the fund-flow controller, so that likely calculates the correct vault IDs using on-chain information.

The `unstakeClaim` and `restakeRewards` functions are permissionlessly callable, so messing with the vault IDs of someone else's call does not add to the threat surface aside from causing calls to unexpectedly revert.

The `queueValidatorRemoval` function is only callable by the owner after the `finalizeValidatorRemoval` is successfully called, so it cannot be reordered.

Impact

The only impact is that the block builder of a block that includes the `finalizeValidatorRemoval` call can cause some calls to `unstakeClaim`, `restakeRewards`, and `queueValidatorRemoval` to unexpectedly revert. This is a minor inconvenience.

Recommendations

We recommend refactoring the storage to not have to reassign IDs when a validator is removed. Iteration through the validators can still be done by maintaining a next-index pointer for each validator.

This will increase the friendliness of the external interface, because users and external code can store the ID as a permanent identifier of the particular vault.

Remediation

This issue has been acknowledged by Stake.link.

3.13. The addFee functions can repeat receivers

Target	PolygonStrategy, LSTRewardsSplitter		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the PolygonStrategy contract, the addFee function adds a new receiver with an assigned split of the fee:

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

Similarly, in the LSTRewardsSplitter contract, there is another addFee function that does the same thing:

```
function addFee(address _receiver, uint256 _feeBasisPoints)
    external onlyOwner {
    fees.push(Fee(_receiver, _feeBasisPoints));
    if (_totalFeesBasisPoints() > 10000) revert FeesExceedLimit();
}
```

Both of these addFee functions can add the same receiver multiple times.

Impact

It may be unintuitive to users or external code that a receiver can be present multiple times in the fees array.

Recommendations

We recommend adding a mapping by the receiver address in order to ensure that a receiver is not added twice. If the mapping was public, this would also allow users to easily check what the fee is

for a particular receiver without having to iterate across the entire `fees` array.

Remediation

This issue has been acknowledged by Stake.link.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The logic in `_updateStrategyRewards` may not need to update all the strategies

The `_updateStrategyRewards` function is called whenever the fees in the PolygonStrategy are changed in order to call into the staking pool to distribute rewards at the old fee configuration first:

```
function _updateStrategyRewards() internal {
    address[] memory strategies = stakingPool.getStrategies();
    uint256[] memory strategyIdxs = new uint256[](strategies.length);
    for (uint256 i = 0; i < strategies.length; ++i) {
        strategyIdxs[i] = i;
    }
    stakingPool.updateStrategyRewards(strategyIdxs, "");
}
```

The `updateStrategyRewards` function in the StakingPool contract then calls `updateDeposits` on the PolygonStrategy contract:

```
function updateStrategyRewards(uint256[] memory _strategyIdxs,
    bytes memory _data) external {
    if (msg.sender != rebaseController && !_strategyExists(msg.sender))
        revert SenderNotAuthorized();
    _updateStrategyRewards(_strategyIdxs, _data);
}
// [...]
function _updateStrategyRewards(uint256[] memory _strategyIdxs,
    bytes memory _data) private {
    // [...]

    // sum up rewards and fees across strategies
    for (uint256 i = 0; i < _strategyIdxs.length; ++i) {
        IStrategy strategy = IStrategy(strategies[_strategyIdxs[i]]);

        (
            int256 depositChange,
            address[] memory strategyReceivers,
            uint256[] memory strategyFeeAmounts
        ) = strategy.updateDeposits(_data);
        totalRewards += depositChange;
```

```
        // [...]
    }

    // [...]
}
```

It is unclear whether it is valid to call `updateStrategyRewards` with a subset of the strategies, but since the fees configured in the PolygonStrategy only apply to itself, it is likely that only the caller PolygonStrategy needs to actually be updated, because no other contract will behave differently from the newly set fees.

If so, we recommend determining the index of the current strategy and only calling `updateStrategyRewards` on itself, to save gas and reduce the attack surface for timing or sandwiching the `updateDeposits` calls on other strategies.

4.2. The LSTRewardsSplitter's splitting is not path-independent, which may have cross-protocol implications

The `splitRewards` function in the LSTRewardsSplitter does the actual splitting referenced in the contract name:

```
function splitRewards() external {
    int256 newRewards = int256(lst.balanceOf(address(this)))
    - int256(principalDeposits);
    if (newRewards < 0) {
        principalDeposits -= uint256(-1 * newRewards);
    } else if (newRewards == 0) {
        revert InsufficientRewards();
    } else {
        _splitRewards(uint256(newRewards));
    }
}
// [...]
function _splitRewards(uint256 _rewardsAmount) private {
    for (uint256 i = 0; i < fees.length; ++i) {
        Fee memory fee = fees[i];
        uint256 amount = (_rewardsAmount * fee.basisPoints) / 10000;

        if (fee.receiver == address(lst)) {
            IStakingPool(address(lst)).burn(amount);
        } else {
            lst.safeTransfer(fee.receiver, amount);
```

```
        }
    }

    principalDeposits = lst.balanceOf(address(this));
    emit RewardsSplit(_rewardsAmount);
}
```

Each time the function is called, at the end, the `principalDeposits` state variable becomes the balance of the contract. So, at the start, it is the previous split's ending balance. If the balance has dropped since the previous split, then the principal is lowered without any splitting logic. If the balance has increased, then the increase in balance causes a fee to be sent out to all the receivers.

This is notably not a path-independent splitting process. If the balance of the address increases and then decreases by the same quantity or a matching percentage, then the net behavior of the contract will be nothing if no call to `splitRewards` was done in the middle, but fees will be sent out if it was called in the middle.

Depending on the specific implementation of the underlying token and the associated other protocols, it is possible that profit-seeking attackers can have the ability to reentrantly call `splitRewards` while the balance of the contract is ephemerally high or low. Since this would be read-only reentrancy, reentrancy locks on the external contract are not likely to cause calls to `balanceOf` to revert.

For instance, if the underlying token is a rebasing token whose current value depends on its own valuation of several assets held by a vault, and the vault has open orders that attackers can permissionlessly fill trades on, then, during the order fill, the balance of the LSTRewardsSplitter would be either temporarily higher or lower than the correct value after the fill is complete. In this case, using the LSTRewardsSplitter to split this asset would not be safe, because the path dependency of the `splitRewards` logic will let anyone cause more or less principal to be distributed than originally intended.

4.3. Calling `checkUpkeep` on chain is against the automation documentation and may revert or waste gas

In the Chainlink automation documentation ↗, the `checkUpkeep` function is defined to contain gas-intensive logic that is explicitly indicated to be run off chain, in order to generate data that is then passed into the `performUpkeep` function. It recommends having this function use the `cannotExecute` modifier from their AutomationCompatible contract in order to ensure that this function is not executed on chain.

However, the PolygonFundFlowController contract does execute this on chain in the `withdrawVaults` function:

```
function withdrawVaults(uint256[] calldata _vaultIds) external {
```

```
        strategy.unstakeClaim(_vaultIds);

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

        if (upkeepNeeded) {
            withdrawalPool.performUpkeep("");
        }
    }
```

If it is required that upkeep needs to be done after the `unstakeClaim`, then the current logic is the only way to guarantee that upkeep was attempted. However, if that is not a hard requirement, then we recommend omitting this check and call. If the `withdrawalPool` contract contains gas-intensive operations in its `checkUpkeep` function, or it takes the above linked recommendation and disallows `checkUpkeep` from running on chain at all, then `withdrawVaults` will start failing.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: LSTRewardsSplitterController.sol

Function: `performUpkeep(bytes calldata _performData)`

This function splits rewards between fee receivers.

Inputs

- `_performData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: ABI-encoded list of splitters to call.

Branches and code coverage

Intended branches

- Iterate through the list of splitters to call and call `performUpkeep` on them.
  - ☑ Test coverage
- Check if any splitter was called.
  - ☑ Test coverage

Negative behavior

- Revert transactions if any splitter was not called.
  - ☑ Negative test

Function: `withdraw(uint256 _amount)`

This function is called to withdraw tokens from the splitter contract.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to deposit.

Branches and code coverage

Intended branches

- Check if the caller is a splitter.
  - ☑ Test coverage
- Call the withdraw function of the splitter.
  - ☑ Test coverage

Negative behavior

- Revert transaction if caller is not a splitter.
  - ☑ Negative test

5.2. Module: LSTRewardsSplitter.sol

Function: `deposit(uint256 _amount)`

This function deposits an amount of LST tokens.

Inputs

- `_amount`
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to deposit.

Branches and code coverage

Intended branches

- Receive the amount of tokens to deposit.
  - ☑ Test coverage

- Add the amount to the `principalDeposits`.
  - ☑ Test coverage

Function: `performUpkeep(bytes calldata)`

This function splits rewards between fee receivers.

Branches and code coverage

Intended branches

- Subtract LST balance from principal deposits to get new rewards.
  - ☑ Test coverage
- Add negative rewards, if there are any, to principal deposits.
  - ☑ Test coverage
- Check if there are enough new rewards.
  - ☑ Test coverage
- Split rewards otherwise.
  - ☑ Test coverage

Negative behavior

- Revert transactions if there are not enough new rewards.
  - ☑ Negative test

Function: `splitRewards()`

This function splits rewards between fee receivers. Unlike `performUpkeep`, it bypasses the reward threshold.

Branches and code coverage

Intended branches

- Subtract LST balance from principal deposits to get new rewards.
  - ☑ Test coverage
- Add negative rewards, if there are any, to principal deposits.
  - ☑ Test coverage
- Check if there are enough new rewards.

- ☑ Test coverage
- Split rewards otherwise.
  - ☑ Test coverage

Negative behavior

- Revert transactions if there are not enough new rewards.
  - ☑ Negative test

Function: `withdraw(uint256 _amount, address _receiver)`

This function withdraws an amount of LST tokens.

Inputs

- `_amount`
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to deposit.
- `_receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address to receive tokens.

Branches and code coverage

Intended branches

- Subtract the amount to `principalDeposits`.
  - ☑ Test coverage
- Transfer the amount of tokens to the receiver.
  - ☑ Test coverage

5.3. Module: PolygonFundFlowController.sol

Function: `restakeRewards(uint256[] _vaultIds)`

This function restakes rewards for vaults. It is a wrapper of `restakeRewards` in strategy.

Inputs

- `_vaultIds`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Array of vaults to restake rewards for.

Branches and code coverage

Intended branches

- Call `restakeRewards` in strategy.
  - ☑ Test coverage

Function: `unbondVaults()`

This function initiates the unbonding of staked tokens from validator vaults when user withdrawals exceed available liquid funds.

Branches and code coverage

Intended branches

- Check if there is a vault that is being unbonded.
  - ☑ Test coverage
- Check if enough time has passed since the last unbonding.
  - ☑ Test coverage
- Check if there are enough queued withdrawals to cover validator-removal deposits.
  - ☑ Test coverage

Negative behavior

- Revert transaction if there is not a vault that is being unbonded.
  - ☑ Negative test
- Revert transaction if not enough time has passed since the last unbonding.
  - ☑ Negative test
- Revert transaction if there are not enough queued withdrawals to cover validator-removal deposits.
  - ☑ Negative test

Function: `withdrawVaults(uint256[] _vaultIds)`

This function withdraws from vaults and then performs upkeep if needed.

Inputs

- `_vaultIds`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Array of vaults to withdraw from.

Branches and code coverage

Intended branches

- Call `unstakeClaim` for each vault.
  - ☑ Test coverage
- Call `performUpkeep` to perform upkeep if upkeep is needed.
  - ☑ Test coverage

5.4. Module: PolygonStrategy.sol

Function: `deposit(uint256 _amount, bytes)`

This function deposits tokens from the caller into this strategy. It can only be called by the staking pool.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to deposit.

Branches and code coverage

Intended branches

- Receive the amount of tokens to deposit.
  - ☑ Test coverage

- Add the amount to the total deposits and total queued.
  - ☑ Test coverage

5.5. Module: PolygonVault.sol

Function: `deposit(uint256 _amount)`

This function deposits tokens from the vault controller into the validator pool.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to deposit.

Branches and code coverage

Intended branches

- Receive the amount of tokens to deposit.
  - ☑ Test coverage
- Call the `buyVoucherPOL` function on the validator pool.
  - ☑ Test coverage
- Send any remaining tokens back to the vault controller.
  - ☑ Test coverage

Function: `restakeRewards()`

This function restakes rewards in the validator pool.

Branches and code coverage

Intended branches

- Call the `restakePOL` function on the validator pool if there are rewards to restake.
  - ☑ Test coverage

Function: `unbond(uint256 _amount)`

This function queues tokens for withdrawal in the validator pool.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to unbond.

Branches and code coverage

Intended branches

- Call the `sellVoucherPOL` function on the validator pool.
  - ☑ Test coverage
- Send any remaining tokens back to the vault controller.
  - ☑ Test coverage

Function: `withdrawRewards()`

This function withdraws rewards from the validator pool.

Branches and code coverage

Intended branches

- Call the `withdrawRewardsPOL` function on the validator pool if there are rewards to withdraw.
  - ☑ Test coverage
- Send tokens to the vault controller.
  - ☑ Test coverage

Function: `withdraw()`

This function withdraws tokens from the validator pool and sends tokens to the vault controller.

Branches and code coverage

Intended branches

- Call the `unstakeClaimTokensPOL` function on the validator pool.
  - ☑ Test coverage
- Send tokens to the vault controller.
  - ☑ Test coverage

6. Assessment Results

During our assessment on the scoped Polygon Staking contracts, we discovered 13 findings. No critical issues were found. One finding was of high impact, one was of medium impact, four were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.