Zellic

StaFi

Smart Contract Security Assessment

July 12, 2023

Prepared for:

Liam Young

StaFi Protocol

Prepared by:

Ayaz Mammadov and Vlad Toie

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment . . . 4

1.2 Non-goals and Limitations . . . 4

1.3 Results . . . 5

2 Introduction 6

2.1 About StaFi . . . 6

2.2 Methodology . . . 6

2.3 Scope . . . 7

2.4 Project Overview . . . 7

2.5 Project Timeline . . . 8

3 Detailed Findings 9

3.1 The `migrate` function can be recalled . . . 9

3.2 Checks to limit parameters missing . . . 11

3.3 Lack of documentation . . . 13

4 Discussion 15

4.1 Pragma causes math operations to be unchecked by default . . . 15

4.2 Test suite . . . 15

4.3 Reentrancy checks . . . 16

5 Threat Model 17

5.1 Module: StakeManager.sol . . . 17

5.2 Module: StakePool.sol 30

6 Audit Results **39**

6.1 Disclaimer 39

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for StaFi Protocol from July 3th to July 10th, 2023. During this engagement, Zellic reviewed StaFi's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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:

- Can a malicious user steal funds through the staking pools?
- Can a malicious staker cause a frozen state of the staking manager, such that a pool is not stakeable or unstakeable?
- Can a lockup of staked funds occur due to user error?
- Can a malicious user manipulate the rate by burning tokens?
- Can the unstake limit be bypassed?

1.2 Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Other contracts that are not part of the scope of this assessment

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

During this assessment, we focused on a limited scope of files, which restricted our ability to thoroughly analyze the mechanisms of the protocol that impact core invariants of the staking manager and staking pool. This includes external calls that determine the total stake of a staker and other external functions that determine the protocol's functionality.

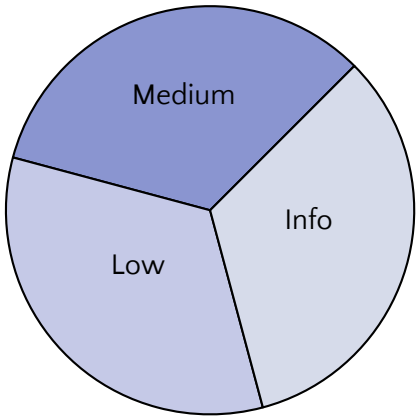
1.3 Results

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

Additionally, Zellic recorded its notes and observations from the assessment for StaFi Protocol's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About StaFi

rMATIC is a decentralized DeFi product issued by StaFi, an LSD solution for staking MATIC on the Ethereum chain. rMATIC was initially issued on the StaFi chain, and the upgrade being audited involves an overall technical architecture enhancement, with the migration of relevant logic and data to Ethereum. Users can perform operations such as stake/unstake/withdraw completely through the StakeManager contract on Ethereum.

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 code base 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, upgradeability 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.

2.3 Scope

The engagement involved a review of the following targets:

StaFi Contracts

Repository	https://github.com/stafiprotocol/rtoken-contracts
Version	rtoken-contracts: `f97c59c7caeb6df49e31bce5a99cfe7f6050a858`
Programs	• rmatic/StakeManager • rmatic/StakePool
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six engineer days. The assessment was conducted over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 3, 2023	Start of primary review period
July 10, 2023	End of primary review period

3 Detailed Findings

3.1 The `migrate` function can be recalled

- **Target**: StakeManager
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The `migrate` function is responsible for migrating the state of the StakeManager contract when it is bridged to the Ethereum Mainnet. However, the current implementation lacks proper checks, allowing for the `_rate` to be set to zero, which would allow the function to be called again.

```
function migrate(
    address _poolAddress,
    uint256 _validatorId,
    uint256 _govDelegated,
    uint256 _bond,
    uint256 _unbond,
    uint256 _rate,
    uint256 _totalRTokenSupply,
    uint256 _totalProtocolFee,
    uint256 _era
) external onlyAdmin {
    require(rate == 0, "already migrate");
    require(bondedPools.add(_poolAddress), "already exist");

    validatorIdsOf[_poolAddress].add(_validatorId);
    poolInfoOf[_poolAddress] = PoolInfo(
        {
        bond: _bond,
        unbond: _unbond,
        active: _govDelegated
        });

    rate = _rate;
    totalRTokenSupply = _totalRTokenSupply;
```

```
    totalProtocolFee = _totalProtocolFee;
    latestEra = _era;
    eraRate[_era] = _rate;
}
```

Impact

In addition to the obvious impact of the contract being migrated with incorrect values, if the `_rate` in the `migrate` function is set to zero, it opens the possibility of the function being called again, potentially causing unintended consequences for the contract. The limited severity in this case is due to the fact that the function can only be called by the contract's admin, and the admin is a trusted entity.

Recommendations

We recommend ensuring that all parameters are comprehensively checked before the migration is allowed to proceed. One way to do this is to implement input validation checks in the `migrate` function to ensure that only valid and expected values are accepted for migration. Furthermore, we highly recommend to explicitly check the `_rate` parameter to ensure that it is not set to zero.

Remediation

This issue has been acknowledged by StaFi Protocol, and a fix was implemented in commit 1f980d34.

3.2 Checks to limit parameters missing

- **Target**: StakeManager
- **Category**: Code Maturity
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

Both the `init` and `setParams`, `migrate` functions are used for modifying the contract's most important state variables, such as the `_eraSeconds`, the `_minStakeAmount`, and more. However, both functions lack proper checks to ensure that the parameters are within acceptable ranges or that they are not set to zero.

For example, despite the `eraSeconds` being checked against zero in the `setParams` function, there is no upper bound check to ensure that the `_eraSeconds` is not set to a value that is too large to be handled by the contract.

```
function setParams(
    uint256 _unstakeFeeCommission,
    uint256 _protocolFeeCommission,
    uint256 _minStakeAmount,
    uint256 _unbondingDuration,
    uint256 _rateChangeLimit,
    uint256 _eraSeconds,
    uint256 _eraOffset
) external onlyAdmin {
    unstakeFeeCommission = _unstakeFeeCommission
    == 1 ? unstakeFeeCommission : _unstakeFeeCommission;
    protocolFeeCommission = _protocolFeeCommission
    == 1 ? protocolFeeCommission : _protocolFeeCommission;
    minStakeAmount = _minStakeAmount == 0 ? minStakeAmount
    : _minStakeAmount;
    rateChangeLimit = _rateChangeLimit == 0 ? rateChangeLimit
    : _rateChangeLimit;
    eraSeconds = _eraSeconds == 0 ? eraSeconds : _eraSeconds;
    eraOffset = _eraOffset == 0 ? eraOffset : _eraOffset;
    if (_unbondingDuration > 0) {
        unbondingDuration = _unbondingDuration;
        emit SetUnbondingDuration(_unbondingDuration);
    }
}
```

Impact

The lack of checks on the parameters may result in the contract being set to an invalid state or a state that is not expected by the contract's users. For example, setting the `_eraSeconds` to a very large value may result in the contract being unable to handle `eras` properly, since it would take too long for the contract to progress to the next `era`.

Recommendations

We recommend ensuring that all parameters are comprehensively checked, in a transparent way. One way to do this is to implement input validation checks in the `setParams`, `migrate` and `init` functions to ensure that only valid and expected values are accepted for modification.

Remediation

This issue has been acknowledged by StaFi Protocol, and fixes were implemented in the following commits:

- 1f980d34
- c2053dc3

3.3 Lack of documentation

- **Target**: Project Wide
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

While the white paper accurately describes the project and its mechanisms, there are certain areas in the code where a reaffirmation of the mechanisms would help with comprehensibility.

```
function newEra() external {
    uint256 _era = latestEra.add(1);
    require(currentEra() ≥ _era, "calEra not match");

    // update era
    latestEra = _era;

    uint256 totalNewReward;
    uint256 newTotalActive;
    address[] memory poolList = getBondedPools();
    for (uint256 i = 0; i < poolList.length; ++i) {
        address poolAddress = poolList[i];

        uint256[] memory validators = getValidatorIdsOf(poolAddress);

        // newReward
        uint256 poolNewReward
    = IStakePool(poolAddress).checkAndWithdrawRewards(validators);
        emit NewReward(poolAddress, poolNewReward);
        totalNewReward = totalNewReward.add(poolNewReward);

        // unstakeClaimTokens
        for (uint256 j = 0; j < validators.length; ++j) {
            uint256 oldClaimedNonce
    = maxClaimedNonceOf[poolAddress][validators[j]];
            uint256 newClaimedNonce
    = IStakePool(poolAddress).unstakeClaimTokens(validators[j],
    oldClaimedNonce);
            if (newClaimedNonce > oldClaimedNonce) {
```

```
                maxClaimedNonceOf[poolAddress][validators[j]]
    = newClaimedNonce;

                emit NewClaimedNonce(poolAddress, validators[j],
    newClaimedNonce);
            }
        }
    // ... 70 more lines
    }
}
```

The `newEra` function is a good example of this. The function is quite long (around 100 lines of code), and lacks clear documentation on its purpose. Moreover, it is shallowly commented, and the comments do not provide much insight into the function's purpose or what invariant it is trying to maintain.

Impact

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified in the future.

In general, a lack of documentation impedes the auditors' and external developers' abilities to read, understand, and extend the code. The problem is also carried over if the code is ever forked or reused.

Recommendations

We recommend adding more comments to the code — especially comments that tie operations in code to locations in the white paper and brief comments to reaffirm developers' understanding.

Remediation

This issue has been acknowledged by StaFi Protocol.

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 Pragma causes math operations to be unchecked by default

Many of the files lock pragma to Solidity versions that do not include built-in checks for math operations. It is crucial for future developers to be aware of this and take appropriate measures when performing math operations. They should either utilize the SafeMath library, ensure their code is resilient to overflows and underflows or undergo a thorough security audit.

It is important to note that, at the time of this assessment, all math operations were found to be safe from overflows and underflows.

In the case of the current codebase, the pragma version is pragma Solidity 0.7.6, which is an almost three-year-old version of Solidity. We recommend bumping the version at least to 0.8, which is the earliest version of Solidity that includes built-in checks for math operations.

4.2 Test suite

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, not just surface-level functions. It is important to test the invariants required for ensuring security and also verify mathematical properties as specified in the white paper. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

For example, ensuring that all the intended and negative test cases developed in the signoffs are covered in the test suite would be a good start.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

4.3 Reentrancy checks

No reentrancy checks or modifiers were noted; however, in the case of reentrancy, no security issue would be present as the checks-effects-interactions pattern was followed, though there are external calls out of the scope that could not account for reentrant cases. Reentrancy would not be directly possible in the majority of the external calls present in the protocol.

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: StakeManager.sol

Function: `addStakePool(address _poolAddress)`

Allows adding a new stake pool.

Inputs

- `_poolAddress`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked that it does not already exist within `bondedPools`.
  - **Impact**: The address of the pool to add.

Branches and code coverage (including function calls)

Intended branches

- Should add the `_poolAddress` to `bondedPools`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the admin to call this function. Ensured through the `onlyAdmin` modifier.
  ☑ Negative test
- Should not allow re-adding an existing pool. Ensured through the `require` statement.
  ☑ Negative test

Function: `approve(address _poolAddress, uint256 _amount)`

Should allow the approval of the ERC-20 token to a pool.

Inputs

- `_poolAddress`
  - **Control**: Fully controlled by the admin.
  - **Constraints**: None.
  - **Impact**: The address of the pool to approve.
- `_amount`
  - **Control**: Fully controlled by the admin.
  - **Constraints**: None.
  - **Impact**: The amount to approve.

Branches and code coverage (including function calls)

Intended branches

- Should approve the `_amount` on the `_poolAddress`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the admin to call this function. Ensured through the `onlyAdmin` modifier.
  ☑ Negative test

Function call analysis

- `IStakePool(_poolAddress).approveForStakeManager(erc20TokenAddress, _amount)`
  - **What is controllable?** `poolAddress` and `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Approval fails.

Function: `init(address _rTokenAddress, address _erc20TokenAddress, uint256 _unbondingDuration)`

Should initialize the contract.

Inputs

- `_rTokenAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.

  - **Impact**: The address of the rToken contract.
- `_erc20TokenAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the ERC-20 token contract.
- `_unbondingDuration`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The duration of the unbonding period, in seconds.

Branches and code coverage (including function calls)

Intended branches

- Should set all initial parameters.
  ☑ Test coverage
- Assure that the `erc20TokenAddress` is not the zero address.
  ☐ Test coverage
- Assure that the `_unbondingDuration` is not zero or that it is within a reasonable range.
  ☐ Test coverage
- Assure that the `_rTokenAddress` is not the zero address.
  ☐ Test coverage

Negative behavior

- Should not allow calling it more than once.
  ☑ Negative test

Function: `migrate(address _poolAddress, uint256 _validatorId, uint256 _govDelegated, uint256 _bond, uint256 _unbond, uint256 _rate, uint256 _totalRTokenSupply, uint256 _totalProtocolFee, uint256 _era)`

Allows the migration of the state of the contract to a bridged version of the contract.

Inputs

- `_poolAddress`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked that it does not already exist within bondedpools.
  - **Impact**: The address of the pool to migrate.
- `_validatorId`

  - **Control**: Fully controller by the owner.
  - **Constraints**: Should not exist for the specified pool.
  - **Impact**: The validator ID to migrate.
- `_govDelegated`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The governance-delegated amount to migrate.
- `_bond`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The bonded amount to migrate.
- `_unbond`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The unbonded amount to migrate.
- `_rate`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None; it is important to add a check that it is not zero, so that `migrate` cannot be called more than once.
  - **Impact**: The rate to migrate.
- `_totalRTokenSupply`
  - **Control**: Fully controlled by owner.
  - **Constraints**: None.
  - **Impact**: The total rToken supply to migrate.
- `_totalProtocolFee`
  - **Control**: Fully controlled by owner.
  - **Constraints**: None.
  - **Impact**: The total protocol fee to migrate.
- `_era`
  - **Control**: Fully controlled by owner.
  - **Constraints**: None.
  - **Impact**: The `era` to migrate.

Branches and code coverage (including function calls)

Intended branches

- Should migrate all the states of the contract to the new contract.
  - ☐ Test coverage

- Check that `_rate` is not zero, so that `migrate` cannot be called more than once.
  - ☐ Test coverage
- Check that the `_poolAddress` does not already exist in `bondedPools`.
  - ☑ Test coverage
- Assumed that the admin calls this function with **legitimate** values.
  - ☑ Test coverage

Negative behavior

- Should not allow calling it more than once. Ensured through the `rate` variable partly, since when set it is not checked if it is zero.
  - ☐ Negative test
- Should not be callable by anyone other than the owner. Ensured through the `onlyAdmin` modifier.
  - ☑ Negative test

Function: `newEra()`

Starts a new cycle.

Branches and code coverage (including function calls)

Intended branches

- If pool bond is above the pool, unbond call is delegated.
  - ☐ Test coverage
- If pool unbond is above the pool, bond call is undelegated.
  - ☐ Test coverage

Negative behavior

- Check that the appropriate amount of time has passed since the `newEra`.
  - ☐ Negative test
- Rate cannot change over limit.
  - ☐ Negative test

Function call analysis

- `IStakePool(poolAddress).checkAndWithdrawRewards(validators)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** New pool reward.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- `IStakePool(poolAddress).unstakeClaimTokens(validators[j], oldClaimedNonce)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** New pool reward.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IStakePool(poolAddress).delegate(validators[0], needDelegate)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IStakePool(poolAddress).getTotalStakeOnValidator(validators[j])`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Total stake of validators[j].
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IStakePool(poolAddress).undelegate(validators[j], unbondAmount)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IERC20MintBurn(rTokenAddress).mint(address(this), rTokenProtocolFee)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `redelegate(address _poolAddress, uint256 _srcValidatorId, uint256 _dstValidatorId, uint256 _amount)`

Redelegate a certain token amount to a certain validator.

Inputs

- `_poolAddress`
  - **Control**: Full.

  - **Constraints**: Must contain `srcValidatorId`.
  - **Impact**: The pool being staked into.
- `_srcValidatorId`
  - **Control**: Full.
  - **Constraints**: Cannot be `_dstValidatorId`.
  - **Impact**: The validator whose stake will be removed.
- `_dstValidatorId`
  - **Control**: Full.
  - **Constraints**: Cannot be `_srcValidatorId`.
  - **Impact**: The validator who will receive that stake.
- `_amount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The amount of stake to delegate to the validator.

Branches and code coverage (including function calls)

Intended branches

- DST validator stake amount increases.
  ☐ Test coverage
- SRC validator stake amount decreases.
  ☐ Test coverage
- Add DST validator if not currently in the list.
  ☐ Test coverage
- Remove validator if stake == 0.
  ☐ Test coverage

Negative behavior

- Cannot redelegate 0 amount.
  ☐ Negative test

Function call analysis

- `IStakePool(_poolAddress).getTotalStakeOnValidator(_srcValidatorId)`
  - **What is controllable?** `_srcValidatorId`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `rmStakePool(address _poolAddress)`

Allows removing a stake pool.

Inputs

- `_poolAddress`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked that it exists within `bondedPools`.
  - **Impact**: The address of the pool to remove.

Branches and code coverage (including function calls)

Intended branches

- Should delete the `poolInfoOf[_poolAddress]` entry.
  ☐ Test coverage
- Should remove the `_poolAddress` from `bondedPools`.
  ☑ Test coverage
- Should remove all additional states related to the pool, like the validator IDs.
  ☑ Test coverage
- Assure that each entry of the validators does not have any stake on the pool.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the admin to call this function. Ensured through the `onlyAdmin` modifier.
  ☑ Negative test
- Should not allow removing a nonexisting pool. Ensured through the `require` statement.
  ☑ Negative test
- Should not allow removing a pool that has active or bonded amounts. Ensured through the `require` statement.
  ☑ Negative test

Function call analysis

- `poolAddress.getTotalStakeOnValidator(validators[j])`
  - **What is controllable?** Validators, to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** Returns whether the validator has any stake on the pool.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

N/A.

Function: `setParams(uint256 _unstakeFeeCommission, uint256 _protocolFeeCommission, uint256 _minStakeAmount, uint256 _unbondingDuration, uint256 _rateChangeLimit, uint256 _eraSeconds, uint256 _eraOffset)`

Allows setting the params of the stake manager.

Inputs

- `_unstakeFeeCommission`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The unstake fee commission.
- `_protocolFeeCommission`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The protocol fee commission.
- `_minStakeAmount`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The minimum stake amount.
- `_unbondingDuration`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The unbonding duration.
- `_rateChangeLimit`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The rate change limit.
- `_eraSeconds`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The era seconds.
- `_eraOffset`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Limited. Should be improved.
  - **Impact**: The era offset.

Branches and code coverage (including function calls)

Intended branches

- All the parameters should be checked against either particular values or ranges. Currently not performed.
  - ☐ Test coverage
- Set all necessary parameters. Update all states with the corresponding values.
  - ☑ Test coverage
- Assumed that it can be called multiple times.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin. Ensured through the `onlyAdmin` modifier.
  - ☑ Negative test

Function: `stakeWithPool(address _poolAddress, uint256 _stakeAmount)`

Stake into certain bonded pool.

Inputs

- `_poolAddress`
  - **Control**: Full.
  - **Constraints**: Must be in the `bondedPools` list.
  - **Impact**: The pool to stake into.
- `_stakeAmount`
  - **Control**: Full.
  - **Constraints**: `>= minStakeAmount`.
  - **Impact**: The amount to stake.

Branches and code coverage (including function calls)

Intended branches

- Staked amount increases.
  - ☐ Test coverage

Negative behavior

- Cannot stake into nonexistent pool (not in `bondedPools`).
  - ☐ Negative test
- Cannot stake `< minStakeAmount`.

☐ Negative test

Function: `transferAdmin(address _newAdmin)`

Allows the transfer of the admin role to a new address.

Inputs

- `_newAdmin`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked that it is not the zero address.
  - **Impact**: The address of the new admin.

Branches and code coverage (including function calls)

Intended branches

- Should be a two-step process, such that the new admin has to accept the role.
  ☐ Test coverage
- Should set the new admin address.
  ☑ Test coverage

Negative behavior

- No one other than the current admin should be able to call this function. Ensured through the `onlyAdmin` modifier.
  ☑ Negative test

Function: `transferDelegationBalancer(address _newDelegationBalancer)`

Allows the transfer of the delegation balancer role to a new address.

Inputs

- `_newDelegationBalancer`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked that it is not the zero address.
  - **Impact**: The address of the new delegation balancer.

Branches and code coverage (including function calls)

Intended branches

- Sets the `delegationBalancer` variable to the new address.

☑ Test coverage

Negative behavior

- No one other than the current admin should be able to call this function. Ensured through the `onlyAdmin` modifier.
  ☑ Negative test

Function: `unstakeWithPool(address _poolAddress, uint256 _rTokenAmount)`

Unstaked from a certain bonded pool.

Inputs

- `_poolAddress`
  - **Control**: Full.
  - **Constraints**: Must be in `bondedPools`.
  - **Impact**: The pool.
- `_rTokenAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to unstake.

Branches and code coverage (including function calls)

Intended branches

- Staked tokens are burned.
  ☐ Test coverage
- Returns original rToken.
  ☐ Test coverage

Negative behavior

- Cannot unstake 0 tokens.
  ☐ Negative test
- Cannot unstake more than `UNBOND_TIMES_LIMIT`.
  ☐ Negative test
- Cannot unstake from a pool not in `bondedPools`.

Function call analysis

- `IERC20MintBurn(rTokenAddress).burnFrom(msg.sender, leftRTokenAmount)`
  - **What is controllable?** `leftRTokenAmount` (after unstake fee).

  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IERC20(rTokenAddress).safeTransferFrom(msg.sender, address(this), unstakeFee)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `withdrawProtocolFee(address _to)`

Allows withdrawing the protocol fee.

Inputs

- `_to`
  - **Control**: Fully controlled by admin.
  - **Constraints**: None.
  - **Impact**: The address to withdraw the protocol fee to.

Branches and code coverage (including function calls)

Intended branches

- Assumed it would not be abused by the admin.
  ☑ Test coverage
- Decrease the `rTokenAddress` balance of the contract by transferring it to `_to`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the admin to call this function. Ensured through the `onlyAdmin` modifier.
  ☑ Negative test

Function call analysis

- `IERC20(rTokenAddress).safeTransfer(_to, IERC20(rTokenAddress).balanceOf(address(this)))`
  - **What is controllable?** `_to`.

- **If return value controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters, or does other unusual control flow?** Transfer fails.

Function: `withdrawWithPool(address _poolAddress)`

Withdraw unstaked tokens from the pool.

Inputs

- `_poolAddress`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The bonded pool to withdraw unstaked tokens from.

Branches and code coverage (including function calls)

Intended branches

- Calls `withdrawForStaker`.
  - ☐ Test coverage

Negative behavior

- Cannot withdraw twice.
  - ☐ Negative test

Function call analysis

- `IStakePool(_poolAddress).withdrawForStaker(erc20TokenAddress, msg.sender, totalWithdrawAmount)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.2 Module: StakePool.sol

Function: `approveForStakeManager(address _erc20TokenAddress, uint256 amount)`

Increase allowance of the specified token for the stake manager.

Inputs

- `_erc20TokenAddress`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The token to increase allowance for.
- `amount`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None,
  - **Impact**: The amount to increase allowance by.

Branches and code coverage (including function calls)

Intended branches

- Increase allowance of the specified token for the stake manager.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the stake manager to call it. Ensured through `onlyStakeManager`.
  ☑ Negative test

Function: `checkAndWithdrawRewards(uint256 _validators)`

Allows the stake manager to withdraw rewards.

Inputs

- `_validators`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The validators to withdraw rewards from.

Branches and code coverage (including function calls)

Intended branches

- Allow withdrawing the rewards from the given validators.
  ☑ Test coverage
- Verify that the validators are legitimate.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the stake manager. Ensured through `onlyStakeManager`.
  - ☑ Negative test
- Should not allow calling the same validator twice — in other words, assumed that the validator's contract will return a different liquid reward if called multiple times.
  - ☑ Negative test
- Should not allow calling it with illegitimate parameters. Not specifically checked — it also depends on the `ValidatorShare`'s implementation.
  - ☐ Negative test

Function call analysis

- `govStakeManager.getValidatorContract(_validators[j])`
  - **What is controllable?** `_validators[j]`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; returns the validator contract address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the validator is not valid.
- `IValidatorShare(valAddress).getLiquidRewards(address(this))`
  - **What is controllable?** `valAddress` partly.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; returns the liquid rewards.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the validator is not valid.
- `IValidatorShare(valAddress).buyVoucher(0, 0)`
  - **What is controllable?** `valAddress` partly.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the `amountToDeposit`, which is currently not checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** None.

Function: `delegate(uint256 _validator, uint256 _amount)`

Allows delegating.

Inputs

- `_validator`
  - **Control**: Fully controlled by the stake manager.

  - **Constraints**: Will call function, so it must be a valid validator ID.
  - **Impact**: The validator to delegate to.
- `_amount`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The amount to delegate.

Branches and code coverage (including function calls)

Intended branches

- Should call `buyVoucher` on the validator share contract with the given parameters.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the stake manager to call it. Ensured through `onlyStakeManager`.
  ☑ Negative test
- Should not allow calling it with illegitimate parameters. Not specifically checked — it also depends on the `ValidatorShare`'s implementation.
  ☐ Negative test

Function call analysis

- `valAddress.buyVoucher(_amount, 0)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the amount is not valid.

Function: `init(address _stakeMangerAddress, address _govStakeMangerAddress)`

Initializes the contract.

Inputs

- `_stakeMangerAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The stake manager address.

- `_govStakeMangerAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The gov stake manager address.

Branches and code coverage (including function calls)

Intended branches

- Set the stake manager address.
  ☑ Test coverage
- Set the gov stake manager address.
  ☑ Test coverage
- Rename to `_stakeManagerAddress` and `_govStakeManagerAddress`.
  ☐ Test coverage
- Assure `_stakeMangerAddress` and `_govStakeMangerAddress` are not zero.
  ☐ Test coverage
- Assure `_stakeMangerAddress` and `_govStakeMangerAddress` are not the same.
  ☐ Test coverage

Negative behavior

- Should not allow being called twice. Partly ensured through `require(stakeMana gerAddress() == address(0), “already init”);`.
  ☐ Negative test

Function: `redelegate(uint256 _fromValidatorId, uint256 _toValidatorId, uint256 _amount)`

Allows the stake manager to redelegate.

Inputs

- `_fromValidatorId`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The validator to redelegate from.
- `_toValidatorId`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The validator to redelegate to.
- `_amount`

  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The amount to redelegate.

Branches and code coverage (including function calls)

Intended branches

- Should call `migrateDelegation` on the gov stake manager with the given parameters.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the stake manager. Ensured through `onlyStakeManager`.
  ☑ Negative test

Function call analysis

- `govStakeManager.migrateDelegation(_fromValidatorId, _toValidatorId, _amount)`
  - **What is controllable?** `fromValidatorId`, `toValidatorId`, and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Then the redelegate will fail.

Function: `undelegate(uint256 _validator, uint256 _claimAmount)`

Allows undelegating.

Inputs

- `_validator`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: Will call function, so it must be a valid validator ID.
  - **Impact**: The validator to undelegate from.
- `_claimAmount`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The amount to undelegate.

Branches and code coverage (including function calls)

Intended branches

- Should call `sellVoucher_new` on the validator share contract with the given parameters.
  ☑ Test coverage
- The amount should be considered undelegated after the call.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the stake manager to call it. Ensured through `onlyStakeManager`.
  ☑ Negative test
- Should not allow calling it with illegitimate parameters. Not specifically checked — it also depends on the `ValidatorShare`'s implementation.
  ☐ Negative test

Function call analysis

- `valAddress.sellVoucher_new(_claimAmount, _claimAmount)`
  - **What is controllable?** `_claimAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the amount is not valid.

Function: `unstakeClaimTokens(uint256 _validator, uint256 _claimedNonce)`

Allows unstaking the claim tokens.

Inputs

- `_validator`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: Will call function, so it must be a valid validator ID.
  - **Impact**: The validator to unstake from.
- `_claimedNonce`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The claimed nonce.

Branches and code coverage (including function calls)

Intended branches

- Assure that the `_claimNonce` is valid. Basically ensured through `unbonds_new()` function call.
  ☑ Test coverage
- Should call `unstakeClaimTokens_new` on the validator share contract with the given parameters.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the stake manager to call it. Ensured through `onlyStakeManager`.
  ☑ Negative test

Function call analysis

- `govStakeManager.getValidatorContract(_validator)`
  - **What is controllable?** `_validator`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; returns the validator contract address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the validator is not valid.
- `IValidatorShare(valAddress).unbonds_new(address(this), willClaimedNonce)`
  - **What is controllable?** `willClaimedNonce`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; returns the unbond struct.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the nonce is not valid.
- `IValidatorShare(valAddress).unstakeClaimTokens_new(willClaimedNonce)`
  - **What is controllable?** `willClaimedNonce`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Assumed it will revert if the nonce is not valid.

Function: `withdrawForStaker(address _erc20TokenAddress, address _staker, uint256 _amount)`

Allows withdrawing tokens for a staker.

Inputs

- `_erc20TokenAddress`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The token to withdraw.
- `_staker`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: None.
  - **Impact**: The address to withdraw to.
- `_amount`
  - **Control**: Fully controlled by the stake manager.
  - **Constraints**: Checked that it is not zero.
  - **Impact**: The amount to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Transfer the `_amount` of `_erc20TokenAddress` to `_staker`.
  ☑ Test coverage
- Decrease the balance of `_erc20TokenAddress` in `address(this)` by `_amount`.
  ☑ Test coverage
- Increase the balance of `_erc20TokenAddress` in `_staker` by `_amount`.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the stake manager. Ensured through `onlyStakeManager`.
  ☑ Negative test
- Should not be abused since it literally transfers any token to any address. Theoretically this should be ensured in the StakeManager contract.
  ☐ Negative test

Function call analysis

- `IERC20(_erc20TokenAddress).safeTransfer(_staker, _amount)`
  - **What is controllable?** `_erc20TokenAddress`, `_staker`, and `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Then the transfer will fail; it means there is not enough balance.

6 Audit Results

At the time of our audit, the audited code was deployed to the Ethereum Mainnet.

During our assessment on the scoped StaFi contracts, we discovered three findings. No critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature. StaFi Protocol acknowledged all findings and implemented fixes.

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 audit version of 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.

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.