Zellic

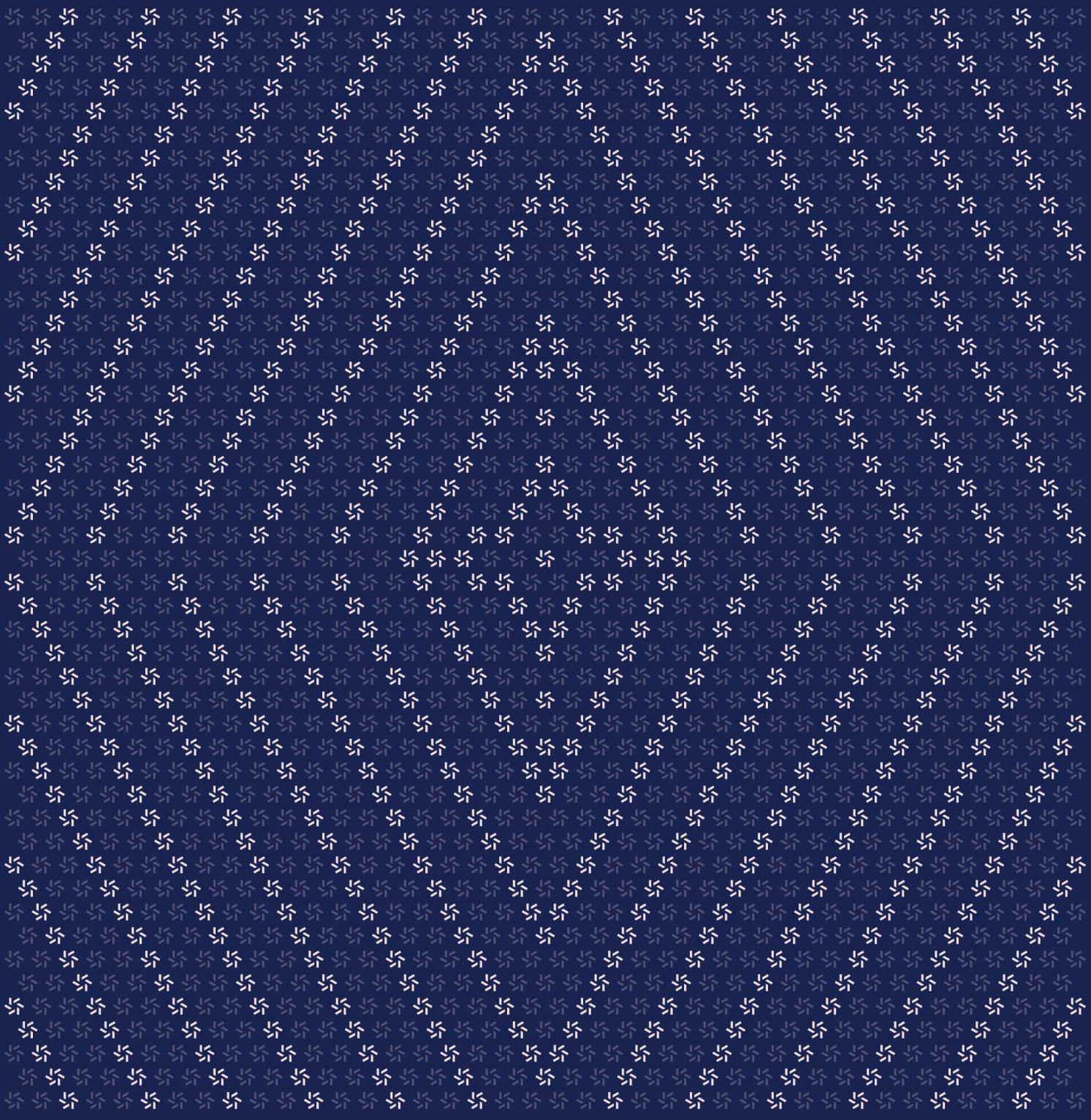
Prepared for
Symbiotic Team
Symbiotic

Prepared by
Junghoon Cho
Mohit Sharma
Zellic

July 30, 2024

Symbiotic

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 Symbiotic from July 15th to July 22nd, 2024. During this engagement, Zellic reviewed Symbiotic'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:

- Are the system and mechanisms of Symbiotic securely designed?
- Are there issues in the protocol math or logic that lead to loss of funds?
- Does Symbiotic have any centralization risks?
- Is Symbiotic secure against common smart contract vulnerabilities?

1.3. 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

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

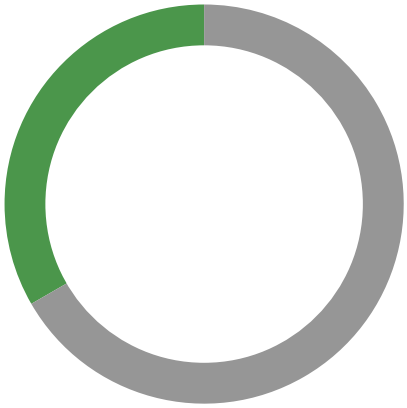
1.4. Results

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

Zellic recorded its notes and observations from the assessment for Symbiotic's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	2



2. Introduction

2.1. About Symbiotic

Symbiotic contributed the following description of Symbiotic:

> Symbiotic is a shared security protocol that serves as a thin coordination layer, empowering network builders to control and adapt their own (re)staking implementation in a permissionless manner.

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:

Symbiotic Contracts

Type	Solidity
Platform	EVM-compatible
Target	core
Repository	https://github.com/symbioticfi/core ↗
Version	`06d18b123e2a83e90d2f311136785ab2411e1d54`

Programs	`DelegatorFactory.sol`
	`NetworkRegistry.sol`
	`OperatorRegistry.sol`
	`SlasherFactory.sol`
	`VaultConfigurator.sol`
	`VaultFactory.sol`
	`common/*.sol`
	`delegator/*.sol`
	`libraries/*.sol`
	`service/*.sol`
	`slasher/*.sol`
	`vault/*.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.7 person-weeks. The assessment was conducted by two consultants 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:

Junghoon Cho
Engineer
junghoon@zellic.io

Mohit Sharma
Engineer
mohit@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 15, 2024	Kick-off call
July 15, 2024	Start of primary review period
July 22, 2024	End of primary review period

3. Detailed Findings

3.1. Veto slash cannot be canceled

Target	VetoSlasher.sol		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

There are two types of slashing methods: instant slashing, which is applied immediately, and veto slashing, which is applied after a certain period. The middleware of each network can request a veto slash for a specific operator through the `requestSlash` function.

However, there is no function implemented to cancel a slash request.

Impact

Once a slash is requested, it can be executed by anyone through the `executeSlash` function after a specified period. Therefore, if the slash amount or operator is incorrectly set, it cannot be reversed.

Recommendations

Add a function that can cancel a specific slash request before the execution window.

Remediation

This issue has been acknowledged by Symbiotic.

3.2. Missing event emissions

Target	Factory.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `whitelist` and `create` functions play a crucial role in the protocol by adding new implementations to the whitelist and cloning them to create entities within the protocol. However, the current version of the protocol does not emit events after these operations are performed.

Impact

Without events, tracking the execution flow and diagnosing issues within these functions become more challenging. Also, off-chain systems may not be aware of the actions taken within the contract, reducing the overall transparency.

Recommendations

Modify `whitelist` and `create` to emit `Whitelist` and `Create` events at the end of each function, respectively.

Remediation

This issue has been acknowledged by Symbiotic, and a fix was implemented in commit `55d95690` ↗.

3.3. Implementation whitelisting is irreversible

Target	Factory.sol		
Category	Protocol Risks	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The current version of the protocol allows adding a new implementation address to `_whitelistedImplementations` using the `whitelist` function. However, there is no function implemented to remove an implementation from this set, making the whitelisting of implementations irreversible.

```
/**
 * @inheritdoc IFactory
 */
function whitelist(address implementation_) external onlyOwner {
    if (IEntity(implementation_).FACTORY() != address(this) ||
    IEntity(implementation_).TYPE() != totalTypes()) {
        revert InvalidImplementation();
    }
    if (!_whitelistedImplementations.add(implementation_)) {
        revert AlreadyWhitelisted();
    }
}
```

Impact

When an implementation that was previously whitelisted is later found to be flawed, it cannot be removed from the whitelist to prevent the creation of new instances of the implementation. This necessitates additional concern from the admin.

Recommendations

Add a function that can remove a specific implementation from `_whitelistedImplementations`.

Remediation

This issue has been acknowledged by Symbiotic, and a fix was implemented in commit `916b3ede` ↗.

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. Incompatible with fee-on-transfer tokens

Users can deposit collateral tokens into the vault and receive shares. The implementation updates the active stake and calculates shares with the user's specified amount.

```
function deposit(address onBehalfOf, uint256 amount)
    external returns (uint256 shares) {
    // [...]
    IERC20(collateral).safeTransferFrom(msg.sender, address(this), amount);

    uint256 activeStake_ = activeStake();
    uint256 activeShares_ = activeShares();

    shares = ERC4626Math.previewDeposit(amount, activeShares_, activeStake_);

    _activeStake.push(Time.timestamp(), activeStake_ + amount);
    // [...]
}
```

If the collateral is a fee-on-transfer token, the vault will receive tokens less than the `amount`. This will cause a mismatch between the accounting and the actual balance of the contract.

To support fee-on-transfer tokens, we recommend checking the balance before and after the transfer. This way, the vault gets the actual amount of tokens received.

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: BaseDelegator.sol

Function: `onSlash(address network, address operator, uint256 slashedAmount, uint48 captureTimestamp, byte[] hints)`

This function is called when a slash occurs.

Inputs

- `network`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the network.
- `operator`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the operator.
- `slashedAmount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of the collateral slashed.
- `captureTimestamp`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Time point when the stake was captured.
- `hints`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Hints for the checkpoints' indexes.

Branches and code coverage (including function calls)

Intended branches

- Hook is called if it is set.

☑ Checked

Negative behavior

- Reverts if not called by the slasher.
  ☑ Checked
- Reverts if `slashedAmount` is more than the slashable amount for the given operator.
  ☐ Unchecked

5.2. Module: FullRestakeDelegator.sol

Function: `setNetworkLimit(address network, uint256 amount)`

This function sets a network's limit (how much stake the vault curator is ready to give to the network).

Inputs

- `network`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the network.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: New limit of the network.

Branches and code coverage (including function calls)

Intended branches

- Network limit after the current timestamp is set to the new amount.
  ☑ Checked
- Network limit before the current timestamp returns the old limit.
  ☑ Checked

Negative behavior

- Reverts if the new limit amount exceeds the network limit.
  ☑ Checked
- Reverts if not called by an address with `NETWORK_LIMIT_SET_ROLE`.
  ☐ Unchecked

5.3. Module: NetworkRestakeDelegator.sol

Function: `setOperatorNetworkShares(address network, address operator, uint256 shares)`

This function sets an operator's shares for a network.

Inputs

- `network`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the network.
- `operator`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the operator.
- `shares`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: New shares of the operator for the network.

Branches and code coverage (including function calls)

Intended branches

- Operator shares after the current timestamp is set to the new amount.
  - ☑ Checked
- Operator shares before the current timestamp returns the old limit.
  - ☑ Checked

Negative behavior

- Reverts if not called by `OPERATOR_NETWORK_SHARES_SET_ROLE`.
  - ☐ Unchecked

5.4. Module: Slasher.sol

Function: `slash(address network, address operator, uint256 amount, uint48 captureTimestamp, byte[] hints)`

This function performs a slash using a network for a particular operator by a given amount using hints.

Inputs

- `network`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the network.
- `operator`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the operator.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Maximum amount of the collateral to be slashed.
- `captureTimestamp`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Time point when the stake was captured.
- `hints`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Hints for checkpoint indexes.

Branches and code coverage (including function calls)

Intended branches

- Stake in correct vault is slashed.
  - ☑ Checked
- Cumulative slash is calculated correctly.
  - ☑ Checked

Negative behavior

- Reverts if not called by middleware.
  - ☑ Checked
- Reverts if network has not opted in the vault.
  - ☑ Checked
- Reverts if operator has not opted in the vault.
  - ☐ Unchecked
- Reverts if operator has not opted in the network.
  - ☐ Unchecked
- Reverts if capture timestamp is invalid.
  - ☑ Checked

5.5. Module: Vault.sol

Function: `claim(uint256 epoch)`

This function claims collateral from the vault.

Inputs

- `epoch`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be less than the current epoch.
  - **Impact**: Epoch to claim the collateral for.

Branches and code coverage (including function calls)

Intended branches

- Amount withdrawn in the previous epoch is transferred to the caller.
  ☑ Checked

Negative behavior

- Reverts if called with the current or future epoch.
  ☑ Checked
- Reverts if called twice in the same epoch.
  ☑ Checked
- Reverts if withdrawable amount is zero.
  ☑ Checked

Function: `deposit(address onBehalfOf, uint256 amount)`

Deposit collateral into the vault.

Inputs

- `onBehalfOf`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Should not be `address(0)`.
  - **Impact**: Account the deposit is made on behalf of.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Should not be zero.
  - **Impact**: Amount of the collateral to deposit.

Branches and code coverage (including function calls)

Intended branches

- Appropriate amount of shares is allocated to `onBehalfOf`.
  - ☑ Checked
- Specified amount of collateral is transferred from the caller to the vault.
  - ☑ Checked

Negative behavior

- Reverts if amount is zero.
  - ☑ Checked
- Reverts if `onBehalOf` is `address(0)`.
  - ☑ Checked
- Reverts if whitelist is active and depositor is not whitelisted.
  - ☑ Checked

Function: `onSlash(uint256 slashedAmount, uint48 captureTimestamp)`

This function slashes callback for burning collateral.

Inputs

- `slashedAmount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount to slash.
- `captureTimestamp`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Time when the stake was captured.

Branches and code coverage (including function calls)

Intended branches

- Slashed amount of collateral is transferred to burner.
  - ☑ Checked
- Active stake and the next epoch's withdrawals are slashed in proportion if `captureTimestamp` is in the current epoch.
  - ☑ Checked
- Active stake, current epoch's withdrawal, and next epoch's withdrawals are slashed in proportion if `captureTimestamp` is in the previous epoch.

☑ Checked

Negative behavior

- Reverts if `captureTimestamp` is not in the current or previous epoch.
  ☑ Checked
- Reverts if the caller is not slasher.
  ☑ Checked

Function: `withdraw(address recipient, uint256 amount)`

This function withdraws collateral from the vault.

Inputs

- `recipient`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Cannot be `address(0)`.
  - **Impact**: Account that needs to claim the withdrawal.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Cannot be zero.
  - **Impact**: Amount of the collateral to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Withdrawal updates the active stake and shares of the sender.
  ☑ Checked

Negative behavior

- Reverts if called with zero amount.
  ☑ Checked
- Reverts if amount exceeds caller's active shares.
  ☑ Checked
- Reverts if recipient is `address(0)`.
  ☑ Checked

5.6. Module: VetoSlasher.sol

Function: `executeSlash(uint256 slashIndex)`

This function executes a queued slash request at `slashIndex`.

Inputs

- `slashIndex`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be a valid index into the `slashRequests` array.
  - **Impact**: Index of the slash request.

Branches and code coverage (including function calls)

Intended branches

- Vault is slashed.
  - ☑ Checked
- Hook on delegator is called if it is set.
  - ☐ Unchecked

Negative behavior

- Reverts if `slashedAmount` is an invalid index.
  - ☑ Checked
- Reverts if vote window is still active.
  - ☑ Checked
- Reverts if `captureTimestamp` of request is not in the current epoch.
  - ☑ Checked
- Reverts if request is already completed.
  - ☑ Checked

Function: `requestSlash(address network, address operator, uint256 amount, uint48 captureTimestamp, byte[] hints)`

This function requests a slash using a network and a resolver for a particular operator by a given amount using hints.

Inputs

- `network`
  - **Control**: Completely controlled by the caller.

- **Constraints**: N/A.
- **Impact**: Address of the network.

- `operator`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the operator.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Cannot be zero.
  - **Impact**: Maximum amount of the collateral to be slashed.
- `captureTimestamp`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be within the veto window.
  - **Impact**: Time point when the stake was captured.
- `hints`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Hints for checkpoints' indexes.

Branches and code coverage (including function calls)

Intended branches

- Slash request is queued.
  - ☑ Checked

Negative behavior

- Reverts if slash amount is zero.
  - ☑ Checked
- Reverts if `captureTimestamp` is not within the veto period.
  - ☑ Checked
- Reverts if not called by network middleware.
  - ☑ Checked
- Reverts if network has not opted in the vault.
  - ☐ Unchecked
- Reverts if operator has not opted in the vault.
  - ☐ Unchecked
- Reverts if operator has not opted in the network.
  - ☐ Unchecked

Function: `vetoSlash(uint256 slashIndex)`

This function vetoes a slash with a given slash index.

Inputs

- `slashIndex`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be a valid index in the `slashRequests` array.
  - **Impact**: Index of the slash request.

Branches and code coverage (including function calls)

Intended branches

- Vetoed shares of a request are incremented by the caller's resolver shares.
  - ☑ Checked
- Request is marked completed if vetoed shares reach shares base.
  - ☑ Checked

Negative behavior

- Reverts if the slash index is invalid.
  - ☑ Checked
- Reverts if the request is already completed.
  - ☑ Checked
- Reverts if the user has already vetoed the request.
  - ☑ Checked
- Reverts if the veto period has ended.
  - ☑ Checked

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Symbiotic contracts, we discovered three findings. No critical issues were found. One finding was of low impact and the other 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.