Security Assessment & Formal Verification Final Report

SYMBIOTIC

Symbiotic Shared Security Protocol

August 2024

Prepared for Symbiotic

certora

SYMBIOTIC

Table of content

certora

SYMBIOTIC

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Symbiotic Shared Security Protocol	https://github.com/symbioticfi/core	9e81d85b94d13141b7046316d400d36c27922caa	EVM

Project Overview

This document describes the specification and verification of **Symbiotic Shared Security Protocol** using the Certora Prover and manual code review findings. The work was undertaken from **August 12, 2024,** to **September 2, 2024.**

The following contract list is included in our scope:

```
/symbiotic/core/src/contracts/common/*
/symbiotic/core/src/contracts/delegator/*
/symbiotic/core/src/contracts/libraries/*
/symbiotic/core/src/contracts/service/*
/symbiotic/core/src/contracts/slasher/*
/symbiotic/core/src/contracts/vault/*
/symbiotic/core/src/contracts/interface/*
```

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct concerning the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Please note that a few more formal rules are not included in this report, as they were proven with an unreleased version of the Certora Prover. Once those rules are proven on a released version of the Certora Prover, we will add them to the next version of this document.

Protocol Overview

Within this document, we audited the symbiotic re-staking protocol. A protocol that provides various networks the ability through delegators, and slasher services to manage a stake held in a user's vault, that stake is represented by an ECR20 token, defined when the vault is created.

The protocol allows for an abstraction layer for networks to slash and reward users while making sure that the stake held in the vault is not over-risked.

Findings Summary

The table below summarizes the review's findings, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	1	-	-
Medium	6	-	-
Low	6	-	-
Informational	2	-	-
Total	15	-	-

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
H-01	H-01 Slashing can be prevented by executing a later slashing first.	High	Fixed
M-01	Vault can be DoS-ed for deposits after a few slashing rounds.	Medium	Acknowledged
M-02	Network middleware can grief the hook by supplying less gas.	Medium	Fixed
M-03	Factory approved implementation cannot be blocked.	Medium	Partially Fixed
M-04	claimBatch can revert mid batch, blocking the entire batch from completing.	Medium	Acknowledged
M-05	Uninitialized entities have exposed APIs that don't revert when called before initialization.	Medium	Fixed
M-06	Withdrawal can be reverted by slashing.	Medium	Fixed
L-01	VetoSlasher execution period isn't in accordance with the docs.	Low	Fixed
L-02	Lack of reentrancy guard in onSlash can lead to unfair behavior when depositing.	Low	Fixed

L-03	Allowing the vault to reduce the stake immediately can be used to avoid predictable slashing	Low	Acknowledged
L-04	Hook might be used to reenter the contract and revert slashing.	Low	Fixed
L-05	Initialization can be front-run for configuration Griffing.	Low	Partially fixed
L-06	Slashing can be prevented if the middleware allows non-ordered execution	Low	Acknowledged

High Severity Issues

H-01 Slashing can be prevented by executing a later slashing first

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VetoSlasher.sol	Status: Fixed	

Description: The veto slasher makes slashing a 2 step process, the network has to request the slash first, and only after the veto-delay passes it can be executed.
Anybody can execute a request after the delay passes.
The execution has to be in a non-descending order, meaning that if we attempt to execute a slashing with a capture timestamp that's lower than the capture timestamp of the last execution it'd revert.
When we have two requests with slightly different capture-timestamps an attacker can execute the latter first, preventing the execution of the second one.

Exploit Scenario:

- An operator has misbehaved on network X
- Network middleware requests 2 slashes, slash A with `captureTime=t` and slash B with `captureTime=t+1`
- After veto duration passes, an attacker (e.g. somebody that holds shares in the vault and wants to reduce the slashing) executes slash B first
- Now slash A can't be executed since its capture timestamp is less than the capture timestamp of the last slashing that was executed.

Recommendations: add access control to `executeSlash()`, allowing only the network to do it

Customer's response: Fixed

Fix commit: 7df0544f0cebd458c83b12fc1b1dea8419f1f07c

Fix Review: Fixed

Medium Severity Issues

M-01 Vault can be DoS-ed for deposits after a few slashing rounds.

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: src/contracts/vault/Vault.sol	Status: Acknowledged	

Description: The vault uses a classic ERC4626 design to keep account of the shares of the users in the vault.
When a user deposits assets new shares are minted according to the current total shares to total assets ratio.
If the shares-to-assets ratio becomes big enough, the number of total shares might exceed the maximum value of `uint256`, causing a revert due to overflow.
Under current implementation, this ratio can increase significantly by large slashings. After a few rounds of slashing we might reach a state where even a small deposit would revert, effectively DoS-ing deposits.

Exploit Scenario:

- Alice deposits 1 WETH (1e18 wei) to the vault
  - Total shares = ~1e18
  - Total assets = ~1e18
  - Shares to assets ratio = ~1
- A 100% slashing happens
  - Total shares = ~1e18
  - Total assets = 0

    - Shares to ratio assets = ~1e18 (`ERC4626Math` adds 1 wei to assets to prevent division by zero)
- Alice deposits 1 WETH (1e18 wei) to the vault
    - Total shares = ~1e36
    - Total assets = ~1e18
    - Shares to assets ratio = ~1e18
- A 100% slashing happens
    - Total shares = ~1e36
    - Total assets = 0
    - Shares to ratio assets = ~1e36
- 2 more similar rounds happen
    - Shares to assets ratio is now ~1e72
- Alice now tries to deposit another 1 WETH into the vault, but it causes an overflow and reverts
    - The new shares to be minted are 1e18*1e72 = 1e90 > 2^256. Therefore the overflow.

Recommendations: One possible way to resolve this is by adding a mechanism to reset the shares on a 100% slashing or when the value of the remaining assets goes below some threshold

Customer's response: Acknowledged

M-02 Network middleware can grief the hook by supplying less gas.

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: BaseDelegator.sol	Status: Fixed	

Description: The protocol allows setting a hook for slashing. That hook is set by a role that's set by the vault during deployment.
The hook is supplied with 250K gas units, and if it runs out of gas the execution of slashing continues.
There's currently no verification that there are 250K gas units left when the function calls, which means it's theoretically possible for the middleware to cause the tx to have less than 250K gas units at this point, causing the hook to run out of gas and revert.
The function would still have 1/64 gas to complete the tx, which might be sufficient, depending on the implementation of the Slasher and the Vault.

```
        if (hook_ != address(0)) {
            bytes memory calldata_ = abi.encodeWithSelector(
                IDelegatorHook.onSlash.selector, subnetwork, operator,
slashedAmount, captureTimestamp, data
            );
            assembly ("memory-safe") {
                pop(call(250000, hook_, 0, add(calldata_, 0x20),
mload(calldata_), 0, 0))
            }
        }
```

Exploit Scenario:

- The slasher and vault are upgraded to a version where less events are emitted,
    - Alternatively, this can be a new version of the slasher where `delegator.onSlash()` is called after `vault.onSlash()`
- Alice deploys a set of vault, delegator and slasher contracts
- Alice sets a hook for slashing, so that whenever slashing happens the hook immediately reduces the stake for that operator to zero
    - That hook consumes 220K gas units, under the 250K limit
- Alice delegates 1K USDC to operator A and 10K USDC to operator B on network X
- Network X operates via a Gnosis Safe multisig wallet, so anybody can execute the wallet's txs
- Due to a misbehavior by both operators, network X signs 2 wallet-transactions to slash 10% of each operator's stake in the vault
- Eve executes in the same tx the 2 wallet transactions
    - Eve sets the gas limit for the tx so that when the hook is called there's 200K units left
    - After the hook runs out of gas, the execution of slashing resumes with 3.1K gas units. The previous slashing reduces the cost of `vault.onSlash()` (since keys are already hot and modified), so that's sufficient to complete the tx
    - As a consequence of that, the slashing is executed but the limit for operator B isn't reduced
- Operator B misbehaves again, and the network issues slashing. That slashing is executed, causing a loss of funds to the vault.

Recommendations: Ensure there's enough gas left before calling the hook (while accounting for the 1/64 rule and the additional gas use before the call), if not - revert the tx.

Customer's response: Fixed

Fix commit: 71f6e32dc2142882fbe53d525b6f9fdfcaefb9e7

Fix Review: Fixed

M-03 Factory approved implementation cannot be blocked.

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: src/contracts/common/Factory.sol	Status: Partially Fixed	

Description: Currently Once an implementation has been approved to a factory, it cannot be revoked. If at any point during the development, a vulnerable implementation is released, it would forever plague the protocol, with no way to upgrade out of it.

Exploit Scenario: Vulnerable Implementation has been added to the whitelist of valid implementations, It is later confirmed that a new version needs to be released in order to avoid exploitation. Bad actors, can still create an entity through the factory of the bad version, or even upgrade an earlier version to the vulnerable one.

Recommendations: Add a blacklist option to ban some implementations. And check when creating or migrating that we are not using the bad versions.

Customer's response: Partially Fixed

Fix commit: 916b3ede63d841a0cd92007b98153d98c74e43d7

Fix Review: The fix review doesn't enforce the blacklisting on deployment on migration. The customer has acknowledged that.

M-04 claimBatch can revert mid batch, blocking the entire batch from completing.

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: src/contracts/vault/Vault.sol	Status: Acknowledged	

Description: claimBatch can revert due to one claim of the batch being bad, such as when it was already claimed,

If at any point a third party would want to use the claim job to earn some rewards, this would block them from completing the batch, disincentivising use of this feature.

Exploit Scenario: One claim within the batch was previously claimed causing a full revert.

Recommendations: Allow for some API either through a boolean argument or a different method call such as forcedClaimBatch that would allow users to finish the batch even if there are "reverts" of some claims within the batch.

Customer's response: Acknowledged

M-05 Uninitialized entities have exposed APIs that don't revert when called before initialization.

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: src/interfaces/common/IEntity.sol	Status: Fixed	

Description: Entities are meant to be used only after initialization has been completed. Currently, most APIs of those entities don't check that it was properly initiated. Even when an entity has been properly created through a factory, it may not have been initialized during that creation.

The exact vulnerabilities are far-reaching, from reinitializing the reentrancy lock to working with bad configuration. Some of those bugs can be mitigated through network checking, but not all. Especially in cases where for example we use that intermediate illegal state to achieve some bad end goal, then finish initialization.

Exploit Scenario: An example of such a theoretical case that is not applicable in the current version is within the vault. using the reentrancy double initialization to cause a double deposit. Then later if initialization is done properly it would seem like a valid vault but with double the intended money.

Recommendations: Add a modifier to make sure all other APIs are locked during initialization.

Customer's response: Fixed

Fix commit: 5f4136494f1f81a983f227662c2c5469237fb742
Fix Review: Fixed

M-06 Withdrawal can be reverted by slashing.		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: Vault.sol	Status: Fixed	

Description: Front-running a withdrawal with slashing can cause the withdrawal to revert. This is because the amount is specified by assets rather than shares, and the amount of assets that a user can withdraw can be reduced by slashing.

Exploit Scenario:

- Alice deploys a vault and delegates 1K USDC to Eve's network
- Bob deposits 10K USDC to the vault
  - This mints 10K*1e6 shares
- After a while Bob tried to withdraw their funds, so they call withdraw() with amount=10K
- Eve front runs this with slashing 10 wei
- This would cause the withdrawal to revert, since Bob doesn't have enough shares to withdraw 10K
  - The required `burnedShares` would be about 10K*1e6+10wei while Bob has only 10K*1e6
- Eve can keep doing that to Bob as long as they try to withdraw their full amount

Recommendations: Allow the user to specify shares rather than amount during withdrawal

Customer's response: Fixed

Fix commit: bdbe883be158cf0806f55988690aac81e52eff1c

Fix Review: Fixed

Low Severity Issues

L-01 VetoSlasher execution period isn't in accordance with the docs.

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: VetoSlasher.sol	Status: Fixed	

Description: According to the docs:
"If the slashing is not resolved after this phase, there is a period of E=EPOCH-V time to execute the slashing"
(where EPOCH is the epoch duration and V is the veto duration)

However, in reality the phase is less than that:

- A request can't be executed till timestamp_at_request+veto_duration
- A request can't be executed after capture_timestamp+epoch_duration
- Therefore the length of the execution period is epoch_duration-veto_duration - (timestamp_at_request-capture_timestamp)

Exploit Scenario:

- Network X requests slashing at `t+500` with `captureTimestamp=t`
- `veto_duration=1000` and `epoch_duration=2000`
- After the voting period has passed (t+1500), the network assumes they have `epoch_duration-veto_duration=1000` to execute it
- At `t+2300` they attempt to execute it, this would revert because more than epoch duration has passed since capture timestamp

Recommendations: Change either the code or the docs to be in accordance with the other.

Customer's response: Fixed

Fix Review: Fixed by updating the docs to match the code

L-02 Lack of reentrancy guard in onSlash can lead to unfair behavior when depositing.

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: src/contracts/vault/Vault.sol	Status: Fixed	

Description: When depositing, the pattern of checking the balance of the vault before and after the deposit might lead to unfair behavior when a coin such as ERC777 is used for that vault. Such a coin will trigger an OnTransfer call when transferring. If during that intermediate time, some collateral is pulled from the vault, such as through a slash. The vault would think that it had a smaller deposit than anticipated.

Exploit Scenario:

- Alice deposits 10 collateral to the vault.
- A slash is triggered after the transfer of the 10 collateral, of 9 assets.
- Deposit continues to run checking and noticing the vault only has 1 asset, gifting shares to Alice equal to 1 collateral deposited. As opposed to 10.

Recommendations: Add a reentrancy guard on onSlash.

In addition check that the depositedAmount and the amount specified by the method are equal and revert if not.

Customer's response: Fixed

Fix commit: f6c67a7412e5f42775a6dafb9926749cb6e7826b

Fix Review: Fixed

L-03 Allowing the vault to reduce the stake immediately can be used to avoid predictable slashing.

Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: NetworkRestakeDelegator.sol FullRestakeDelegator.sol	Status: Acknowledged	

Description: The protocol allows the vault (role holders which are set during the vault deployment, to be more accurate) to reduce the network and network-operator stake, and this reduction takes effect immediately.
In case the network operates in a way that slashing is predictable on or before the capture-timestamp, the vault can detect the upcoming slashing and reduce the stake right before the capture timestamp.

Exploit Scenario:

- Eve deploys a vault-slasher-delegator (NetworkRestakeDelegator) set and sets herself as the `NETWORK_LIMIT_SET_ROLE` holder
- Eve stakes 10K USDC to operator A on network X
- Operator A misbehaves on network X at timestamp `t`
- Eve detects that and immediately reduces the stake to 1 wei
- Network X goes through a process of detecting the misbehavior and issues a slashing with capture timestamp set to `t+30`
- The slashing passes with 1 wei due to the reduced stake
- Eve was able to avoid most of the slashing by predicting

Recommendations: Ensure stake reduction takes place only after some delay that would prevent such a scenario.
Alternatively, warn networks against this scenario, so that they can ensure that capture timestamp is set to a timestamp where the slashing can't be predicted beforehand.

Customer's response: ç. It shouldn't happen if a network uses stakes from the past.

L-04 Hook might be used to reenter the contract and revert slashing.

Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: BaseSlasher.sol BaseDelegator.sol Vault.sol	Status: Fixed	

Description: During slashing a hook is called, that hook can reenter the contract, execute another slashing etc. which might cause the slashing to revert, depending on the implementation of the vault.

Exploit Scenario:

- Vault is upgraded to a version where `onSlash()` reverts if there's no sufficient stake to slash
- Eve deploys a set of vault-delegator-slasher
- Eve also creates network X and delegates 100% of the stake to it
- Eve delegates 100% of the stake to operator A on network Y as well
- Eve adds a hook that would slash 100% of the stake for network X, and sets the hook as network X's middleware
- Operator A misbehaves on network Y, and Y's middleware calls the `slasher.slash()` function
    - The hook slashes 100% of the stake, and as a result when the hook is returned there's no stake left to slash
    - The original slashing execution continues, the slasher calls `vault.onSlash()` with `slashedAmount` set to 100% of the stake
    - `vault.onSlash()` reverts and as a result the whole tx reverts
- This would effectively DoS slashing

Recommendations: Add a reentrancy guard to the slashing function (`slasher.slash()`). Also consider having a 'global' reentrancy guard across the vault, delegator and slasher to prevent other possible reentrancies.

Customer's response: Fixed

Fix commit: f6c67a7412e5f42775a6dafb9926749cb6e7826b

Fix Review: Fixed

L-05 Initialization can be front-run for configuration Griffing.		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: src/interfaces/common/IEntity.sol	Status: Partially Fixed	

Description: Currently there is a possibility to decouple factory creation and initialization of the entities. This allows bad actors to force bad configurations on vaults that have not been initialized, even though they have not created them.

Exploit Scenario: Alice creates a Vault through the Vault factory.

In a different transaction, Alice initializes the Vault.

Eve front-runs the initialization requests, with her malicious configuration.

Recommendations:

1. Add a Creator modifier when creating an entity.
2. Document that using that factory API is unsafe.
3. Remove that feature.

Customer's response: Partially Fixed

Fix commit: 8deaf1c8651d8da4ecec3b0abaa0222b5566ad61 and bd0fbdad953076e0b6c34b91bc28784e585bd3e8

Fix Review: Partially Fixed (by adding code comments)

L-06 Slashing can be prevented if the middleware allows non-ordered execution

Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: VetoSlasher.sol Slasher.sol	Status: Acknowledged	

Description: The slasher requires the execution of slashing to be in a non-descending order (i.e. captureTimestamp shouldn't be older than the captureTimestamp of last execution).
In case that the network's middleware is a smart contract that allows the execution to be out of order (e.g. OpenZeppelin governance contracts) then an attacker might use this to execute the slashing with later capture timestamp first, preventing the execution of the slashing with older capture timestamp.

Exploit Scenario:

- Network X uses OpenZeppelin Governance contracts as the middleware
- 2 slashings for the same vault reach a quorum on the governance
- Eve executes the later slashing first
- Now the older slashing can't be executed, preventing a needed slashing

Recommendations: Warn networks to use as a middleware only smart contracts that enforce an ordered execution

Customer's response: Acknowledged

Informational Severity Issues

I-01. Migratable Factory relies on the implementation to return the correct owner and version

Description: Under current implementation of the MigratableFactory, the `migrate()` function retrieves the version and owner values from the entity. In case of a faulty implementation that returns the wrong version or owner this might brick the migration forever for the entities that use this implementation.

Recommendation: Register the owner and the version in the factory rather than the implementation

Customer's response: Acknowledged

I-02. Gas optimization - no need to insert a new checkpoint in `setMaxNetworkLimit()` under some scenarios

Description: `_setMaxNetworkLimit()` (at `NetworkRestakeDelegator` and `FullRestakeDelegator`) insert a new checkpoint with the lowest value of current limit and the new max limit.
In case that the new max limit is more than the current limit then there's no need to insert a new checkpoint.
This can save about 25K of gas - (22.1K `sstore` for adding a new element, and a 2.9K `sstore` for updating the length of the array). It'd also keep the size of the array shorter, saving some gas on the binary tree search (when no hint is provided).

Recommendation: Don't insert a new checkpoint if the current limit isn't above the new max limit.

Customer's response: Fixed
Fix commit: 57fb130d7e35a6744a7f96485d9a6529417aae60

Fix Review: Fixed

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

1. We work with objects inherited from the original contracts. In the inherited objects we add more view methods, flags, etc. In cases where it was not possible to collect the required information via the inherited object, we modify the original. These modifications don't affect the functionality of original contracts and the verification results hold also for the original contracts.
2. We replaced some functions with equivalent CVL implementations. Notably *mulDiv, epochAt* and the*OptInService* contract. This speeds up the verification process.
3. We unroll loops for a fixed number of iterations. The exact number of iterations is mentioned for each contract.

4.

Formal Verification Properties

Checkpoints

Module General Assumptions

- We verified the contract functions against an arbitrary storage state.
- We verify the contract as a stand-alone one, i.e. we make no assumptions about the caller. We assume all methods may be called with arbitrary arguments.
- In the `upperLookup` methods, the `hint_` argument follows the assumption that it's a lower bound on the actual position.
- Loops are assumed to iterate at most 25 times.

Module Properties

P-01. Integrity of state-changing methods	
Status: Verified	

Rule Name	Status	Description	Link to rule report
pop208_Integrity **pop256_Integrity**	Verified	`Pop` *decreases the* `length` *by* `1` *and returns the* `latest` *element.*	*Report*
push208_Integrity **push256_Integrity**	Verified	`Push` *never reverts and* `push(X); latest();` *returns* `X`.	

P-02. Data are always sorted on keys

Status: Verified

Rule Name	Status	Description	Link to rule report
allwaysSorted208 **allwaysSorted256**	Verified	*The underlying array is always sorted on key.*	*Report*

P-03. Integrity of lookup methods

Status: Verified

Rule Name	Status	Description	Link to rule report
upperLookupRecent208_Integrity **upperLookupRecent256_Integrity**	Verified	*If* `at(x)._key == y` *then* `upperLookupRecentCheckpoint(y).position == x` *and the method doesn't revert.*	*Report*
upperLookupRecentWithHint208_Integrity **upperLookupRecentWithHint256_Integrity**	Verified	*If* `at(x)._key == y && hint <= x` *then* `upperLookupRecentCheckpoint(y, hint).position == x`	

P-04. Push-Pop neutrality

Status: Verified

Rule Name	Status	Description	Link to rule report
pushPopIsNeutral208 **pushPopIsNeutral256**	Verified	*If* `key > latestKey` *then* `pop();` *returns the same value as* `push(key, value); pop(); pop();`	*Report*

Vault

Module General Assumptions

- We verified the contract functions against an arbitrary storage state.
- We use basic standard implementations for underlying ERC20 tokens.
- We verify the contract in presence of other contracts, namely *DelegatorFactory*, *SlasherFactory*, *VaultFactory*. This means that interactions between these contracts are modeled exactly, and we make no assumptions about interactions with other contracts.
- Loops are assumed to iterate at most 3 times.

Module Properties

P-05. Correct accounting of withdrawals

Status: Verified

Rule Name	Status	Description	Link to rule report
noWithdrawalsInFarFuture	Verified	`epoch > currentEpoch()+1` *then* `withdrawals[epoch]=0`	*Report*
withdrawalSharesEqualsSumOfwithdrawalSharesOf	Verified	`withdrawalShares[epoch]` *always equals to* `sum_account(withdrawalSharesOf[epoch][account])`	*Report*

P-06. Correct applications of deposit limits

Status: Verified

Rule Name	Status	Description	Link to rule report
limitFlagSetIffLimitSet	Verified	`isDepositLimit` *is true if and only if* `depositLimit != 0` *(can be violated by admin methods* `setIsDepositLimit` *and* `setDepositLimit`*)*	*Report*
stakeLimitCannotBeBreached	Verified	`isDepositLimit` *is true then* `totalStake <= depositLimit` *(can be violated by admin methods* `setIsDepositLimit` *and* `setDepositLimit`*)*	

P-07. Correct calculations of active funds

Status: Verified

Rule Name	Status	Description	Link to rule report
balanceNotLessThanActiveBalance	Verified	`activeBalanceOf(address) <= balanceOf(address)`	*Report*
totalStakeNotLessThanActiveStake	Verified	`activeStake() <= totalStake()`	

RestakeDelegators

Module General Assumptions

- We verified the contract functions against an arbitrary storage state.
- We use basic standard implementations for underlying ERC20 tokens.
- We verify the contract in presence of other contracts, namely *DelegatorFactory*, *SlasherFactory*, *VaultFactory*. This means that interactions between these contracts are modeled exactly, and we make no assumptions about interactions with other contracts.
- Loops are assumed to iterate at most 3 times.

Module Properties

P-08. Correct applications of network limits

Status: Verified

Rule Name	Status	Description	Link to rule report
noCheckpointsInTheFuture	Verified	`_networkLimit` *don't have checkpoints in the future*	NetworkRestakeDel FullRestakeDel
pastNetworkLimitsCannotBeChanged	Verified	`networkLimitAt(n, epoch)` *for* `epoch < Time.timestamp()` *cannot change*	NetworkRestakeDel FullRestakeDel
networkLimitNotBreached	Verified	`networkLimit[n] < maxNetworkLimit[n]`	NetworkRestakeDel FullRestakeDel

VetoSlasher

Module General Assumptions

- We verified the contract functions against an arbitrary storage state.
- We use basic standard implementations for underlying ERC20 tokens.
- We verify the contract in presence of other contracts, namely `DelegatorFactory`, `SlasherFactory`, `VaultFactory`. This means that interactions between these contracts are modeled exactly, and we make no assumptions about interactions with other contracts.
- Loops are assumed to iterate at most 10 times.

Module Properties

P-09. Correct usage of Resolvers

Status: Verified

Rule Name	Status	Description	Link to rule report
resolverNotSetForFarFuture	Verified	*There can be only up to 1 last checkpoint in the* `_resolver checkpoints` *that's in the future*	*Report*

P-10. Correct calculations of slashes

Status: Verified

Rule Name	Status	Description	Link to rule report
slashedAmountNoMoreThanRequested	Verified	*Slashed amount is not greater than* `slashRequest.amount`	*Report*

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.