

Produced for

SYMBIOTIC

by



CHAINSECURITY

Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	10
4	Terminology	11
5	Findings	12
6	Resolved Findings	14
7	Informational	20
8	Notes	21

1 Executive Summary

Dear all,

Thank you for trusting us to help SymbioticFi with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Core according to [Scope](#) to support you in forming an opinion on their security risks.

SymbioticFi implements a security layer for restaking. Deposited tokens can be assigned to operators running node software of networks. The assigned tokens are guaranteed to be slashable by networks in case of operator misbehaviors.

The most critical subjects covered in our audit are functional correctness, role management and front-running tolerance.

Security regarding the aforementioned subjects is high as only minor issues could be uncovered during this review.

It should be noted that the protocol design is very open, allowing various participants to create registered smart contracts with configurations that can potentially be dangerous. For this reason, it is advised to take special care when trusting any vaults, networks and operators.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	7
•	4
•	1
•	1
•	1



2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Core repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	15 July 2024	06d18b123e2a83e90d2f311136785ab2411e1d54	Initial Version
2	10 August 2024	e4ddd291029993a5392f51164cf3766ed71b8510	After Intermediate Report
3	20 August 2024	9e81d85b94d13141b7046316d400d36c27922caa	Final Version

For the solidity smart contracts, the compiler version 0.8.25 was chosen.

The following files in `src/contracts` are in scope of this review:

- NetworkRegistry.sol
- SlasherFactory.sol
- VaultConfigurator.sol
- OperatorRegistry.sol
- DelegatorFactory.sol
- VaultFactory.sol
- service/MetadataService.sol
- service/NetworkMiddlewareService.sol
- service/OptInService.sol
- libraries/Checkpoints.sol
- libraries/ERC4626Math.sol
- common/StaticDelegateCallable.sol
- common/MigratableEntityProxy.sol
- common/Registry.sol
- common/MigratablesFactory.sol
- common/Factory.sol
- common/MigratableEntity.sol
- common/Entity.sol
- vault/VaultStorage.sol
- vault/Vault.sol
- slasher/Slasher.sol

- slasher/BaseSlasher.sol
- slasher/VetoSlasher.sol
- delegator/BaseDelegator.sol
- delegator/FullRestakeDelegator.sol
- delegator/NetworkRestakeDelegator.sol

In `libraries`, the following files have been added to the scope:

- libraries/Subnetwork.sol

2.1.1 Excluded from scope

Any file not included in `Scope`_`` is out-of-scope. In particular, third-party libraries, including patched versions, and deployment scripts are not in scope.

2.2 System Overview

This system overview describes the initially received version (`1.0.0`) of the contracts as defined in the [Assessment Overview](#).

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

SymbioticFi offers a shared security protocol for restaking collateral across node operators of different services and networks. The system is composed of vaults that hold users' stakes and slashing contracts that can be used by networks to slash the stake that was allocated to them through delegation contracts. Vaults can be created by anyone and can be configured to be immutable or actively managed. Either way, the owner of a vault decides which operators on which network the stake should be delegated to. Networks and operators, in turn, can signal their willingness to cooperate with certain vaults. Once all parties have opted in, the stake is used by operators to secure the respective network.

2.2.1 Registries

The protocol is composed of multiple registries that allow the participants to enter the system and signal cooperation willingness and register:

1. Operators can register in the `OperatorRegistry`.
2. Networks can register in the `NetworkRegistry`.
3. Vaults can be deployed with the `VaultFactory`. This contract also acts as a registry.
4. Vaults make use of slasher and delegator contracts. These must be deployed and registered in `SlasherFactory` and `DelegatorFactory` respectively.

Furthermore, networks can add a middleware contract to the system using the `NetworkMiddlewareService`. Middlewares are used to integrate networks with the protocol and perform slashings.

To signal their willingness of cooperation, participants can register in three different `OptInService` contracts:

1. Operator -> Vault.
2. Operator -> Network.
3. Network -> Vault.

Only if all opt-ins are set, the operator stake in a given vault becomes available to a network.

2.2.2 Vault

The Vault manages the user's collateral and keeps track of the total active stake. The contract can be migrated to a new implementation by the owner.

The Vault divides the block time into epochs of length `epochDuration`, starting with epoch 0 at the time of contract deployment. The epoch duration is set during deployment and cannot be changed.

Epochs are used to give guarantees to networks in terms of slashable stake: At any point in time a network can start its epoch (which should be of smaller length than the vault epoch) by reading its active stake for each operator. From this point on, the protocol guarantees this stake for one full vault epoch. Stakes should be read at timestamps lower than `block.timestamp` to get fixed guarantees.

Users deposit funds into the Vault using the `deposit()` function. The token amount is then converted to its respective share of the collateral and the total active stake and total shares are updated, accordingly. The Vault has a privileged role called `DEPOSITOR_WHITELIST_ROLE` who can restrict deposits to whitelisted addresses only.

To withdraw funds from the Vault, users call the `withdraw()` function, creating a new withdrawal request for the next epoch. This pushes their pro-rata share of the collateral to a withdrawal queue.

Due to the asynchronous nature of the protocol, withdrawals have to be withheld until the end of `currentEpoch + 1`, after which they can be claimed with `claim()`. Since networks can capture their stake at any time and are given a guarantee that lasts one full epoch, this makes sure that all guaranteed funds can still be slashed in the given time period as open withdrawals are still slashable until they can be claimed.

Vault stake and withdrawals are checkpointed with their timestamp using a custom `Checkpoints` library built on top of OpenZeppelin's `Checkpoints`. This allows networks to always fetch the stake at any given point in the past, enabling the aforementioned asynchronous model.

Deposits or withdrawals create a new checkpoint of the:

1. Active stake.
2. Total shares.
3. User shares.

A checkpoint that already exists at the timestamp of the deposit or withdrawal is overwritten. This allows to query the active stake of a Vault at a given timestamp with the `activeStakeAt()` function.

2.2.3 Delegator

Delegator contracts are used to determine the stake allocation of a vault's active stake to networks and operators. Two different delegators exist. Both allow that the stake of the vault can be shared between multiple networks. This means, slashes affect other network's stakes. Therefore, the slashing guarantee given to networks is softened when there are overlaps in stake between networks.

Networks can limit the total amount of stake allocated to them by setting a maximum network limit. Vault owners can then set a limit (up to the maximum) for these networks in the delegator contract. If the total of the limits of all networks exceeds the active stake of the vault, tokens are restaked.

Operator stake, is handled differently in both contracts:

NetworkRestakeDelegator:

The `NetworkRestakeDelegator` contract allows restaking of collateral across multiple networks. Stakes are divided between operators of each network.

The contract includes the following roles:

- `NETWORK_LIMIT_SET_ROLE`: Sets the maximum stake allocated to a network.



- `OPERATOR_NETWORK_SHARES_SET_ROLE`: Sets the share of each operator within a given network.

The share of any operator in a network is calculated as:

$$\text{stake}(\text{network}, \text{operator}) = \min(\text{operatorShares} * \text{networkLimit} / \text{totalShares}, \text{maxnetworklimit})$$

FullRestakeDelegator:

The `FullRestakeDelegator` contract allows restaking of collateral across multiple services AND operators. This means that the stake assigned to a network can be used by multiple operators of that network at the same time.

It includes the following roles:

- `NETWORK_LIMIT_SET_ROLE`: Sets the maximum stake allocated to a network.
- `OPERATOR_NETWORK_LIMIT_SET_ROLE`: Sets the maximum stake allocated to an operator within a network.

The share of any operator in a network is calculated as:

$$\text{stake}(\text{network}, \text{operator}) = \min(\text{operatornetworklimit} * \text{networkLimit} / \text{totalShares}, \text{maxnetworklimit})$$

2.2.4 Slasher

Slasher contracts are (optionally) used to slash the active stake of a vault. It can be set during the deployment of the vault contract.

The slasher retrieves the delegated stake to a network and operator at a specific timestamp (`captureTimestamp`) via the `Delegator.stakeAt()` function and executes the `onSlash()` function on both the delegator and the vault. The maximum stake that is slashable is reduced by the amount of collateral already slashed after the timestamp of the slashing event.

There are two implementations of the slasher contract within the scope of this audit:

BaseSlasher:

This basic implementation contains a single `slash()` function. Slashes must be performed at most one `epochDuration` after the `captureTimestamp` to prevent collateral from being withdrawn prematurely. The `slash()` function can be called by the network middleware.

VetoSlasher:

This advanced implementation allows certain resolvers to veto slashing requests. The process involves two steps:

- The middleware calls the `requestSlash()` function to request a slashing event for a network, operator, and timestamp.
- Resolvers have a `vetoDuration` to veto the slashing request. Consequently, the slashing offense must be executed `epochDuration - vetoDuration` after the `captureTimestamp`.

Any veto vote reduces the amount of stake slashed by `vetoShares / 1e18`. When a total of `1e18` votes are reached, the slashing is fully stopped. The network decides the vote share of resolvers and can increase them using the `setResolverShares()` function. However, vote shares can only be decreased at least `resolverSetEpochsDelay` epochs in the future, which is currently set to a minimum of 3 epochs. This gives users at least 2 epochs to withdraw their stake from the Vault if the voting power of resolvers is decreased.

After the voting period, anyone can call the `executeSlash()` function to execute the slashing event. The maximum stake that is slashable is reduced by the amount of collateral already slashed after the timestamp of the slashing event.

2.2.5 Changes in Version 2

In of the protocol, the following changes were introduced:

1. The "Network -> Vault" Opt-In Service has been removed.
2. Subnetworks were introduced that give a more fine-grained control over which resolver can veto which part of the stake. Only one resolver can be assigned per subnetwork.
3. `VetoSlasher` no longer allows partial vetoes.
4. `VetoSlasher` now allows to instantly execute a slashing if no resolver is set for a given subnetwork.
5. Slashings are now enforced to be executed in-order.
6. `Vault` now supports fee-taking tokens.

2.2.6 Changes in Version 3

In of the protocol, the following changes were introduced:

1. `VetoSlasher` only allows the middleware to execute slashes.
2. `Vault` now enforces a limit on the amount of tokens that can be deposited. For this purpose it adds two more roles:
 - `DEPOSIT_LIMIT_SET_ROLE`: Sets the deposit limit.
 - `IS_DEPOSIT_LIMIT_SET_ROLE`: Enables or disables the deposit limit.
3. Veto periods in `VetoSlasher` end as soon as the corresponding resolver is reset to the 0-address.

2.3 Trust Model

Networks are fully trusted. They can instantly slash all tokens staked to them in case of `Slasher` being used. If `VetoSlasher` is used, networks are trusted to select resolvers (as the contract also allows no resolver to be set for a given subnetwork). Additionally, any rewards are paid out by the networks using their own infrastructure. Rewards are therefore not guaranteed.

Resolvers are fully trusted to veto any unjustified slashings.

Vault admins (and delegator admins which are assigned by the vault admins) are fully trusted. They have full control over the vault configuration that enables them, in the worst case, to steal all user funds.

Factory owners are trusted as they are able to add malicious implementation contracts for vaults, slashers and delegators which can affect new deployments. Existing deployments, however, cannot be affected by such implementations.

Users are advised to take care in reviewing vaults they want to invest in. See [User security considerations](#) for further information.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High			
Medium			
Low			

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the [Resolved Findings](#) section. The findings are split into these different categories:

- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	2

- [Implementation Unwhitelisting](#)
- [Missing Checks](#)

5.1 Implementation Unwhitelisting

CS-SYFC2-001

`Factory` and `MigratablesFactory` allow the owner of the contract to whitelist implementations of which users can then deploy clones. If one of these whitelisted implementations turns out to be vulnerable later on, there is no way to remove them, allowing users to deploy vulnerable contracts. Since the factories are immutable, such problems can only be mitigated by re-deploying and re-configuring them after project launch.

Acknowledged:

The client has acknowledged the issue, but has decided that:

Unwhitelisting doesn't provide any significant advantages for the Vaults' owners in case of broken implementations as each migration must be fully checked at first. However, it may cause issues for external integrations.

5.2 Missing Checks

CS-SYFC2-003

The following checks are missing that could potentially lead to problematic state:

1. `Vault._initialize()` makes sure that a provided slasher has been deployed by the `SlasherFactory`. It, however, does not check if the given slasher actually belongs to the vault (i.e., the `vault` storage variable has been set to the vault's address. If the participants do not check this by hand, chances are that the vault successfully onboards stakers and networks but then fails to execute slashes.

2. When `params.depositWhitelist` is set to `true`, `Vault._initialize()` does not check that at least one of the roles `DEFAULT_ADMIN_ROLE` and `DEPOSITOR_WHITELIST_ROLE` is set. If none are set and the vault uses a whitelist, it is unusable after initialization.
-

Code partially corrected:

1. This issue still holds.
2. `Vault._initialize()` has been updated to revert when `params.depositWhitelist` is `true` and neither of the roles `DEFAULT_ADMIN_ROLE` and `DEPOSITOR_WHITELIST_ROLE` are set.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Findings](#) section.

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	5

- [Slash Execution Denial](#)
- [Gas Griefing Hooks](#)
- [Hook Not Called on Zero Slashed Amount](#)
- [Migration Inconsistencies](#)
- [Slash Execution Order Inconsistency](#)

Informational Findings	5
------------------------	---

- [Non-indexed Event Parameter](#)
- [Gas Optimizations](#)
- [Missing Events](#)
- [No Receiver in Claim\(\) Function](#)
- [Original Slashed Amount in Hook](#)

6.1 Slash Execution Denial

CS-SYFC2-013

In a specific corner case, it is possible that a slash requested via the `VetoSlasher` can not be executed. Consider the following example:

1. A vault's epoch lasts 1 week.
2. A network onboarded on the vault has a network epoch of 1 day.
3. The vault uses `VetoSlasher` with a veto duration of 2 days.
4. The `VetoSlasher` has an active resolver for the network.
5. At vault epoch 1, the network resets the resolver back to the 0-address. This change will take effect at vault epoch 4.
6. The network starts epoch 1 at timestamp x at the end of vault epoch 3.
7. An operator performs a slashable offense during network epoch 1 that is requested in the slasher.
8. The network starts epoch 2 at timestamp $x + 1 \text{ day}$ at the beginning of vault epoch 4.
9. An operator performs a slashable offense during network epoch 2 that is requested in the slasher.

10. The second slashing can be executed immediately by anyone as the resolver is now set to the 0-address and `VetoSlasher.executeSlash()` does not enforce the veto duration when no resolver is set. The first slashing is still in veto period and cannot be executed.
 11. At timestamp $x + 2 \text{ days}$, the first slashing exits the veto duration. However, it cannot be executed as the slashings are only allowed to be executed in order of their capture timestamps.
-

Specification changed:

In [\[link\]](#), the access to `VetoSlasher.executeSlash()` has been restricted to the middleware address. The middleware is responsible for ensuring that the slashings are executed in the correct order. Additionally, veto periods now end prematurely once the change of a resolver back to the 0-address takes into effect.

6.2 Gas Griefing Hooks

CS-SYFC2-006

Vault owners can add a `hook` contract to a vault that is called at the end of a slashing process and can be used to re-arrange limits. The call to the hook has an allowance of 300.000 gas. Networks performing a slash on the vault might expect that the transaction will roughly consume the amount of gas for the actual slashing plus a maximum of 300.000 gas. Since solidity calls automatically copy return data, even when the return data is not used, it is possible that the actual amount of the hook call's gas usage is much higher than 300.000 if the hook uses all the gas to expand memory and returns this data. If networks choose tight gas limits, their transactions might revert.

Code corrected:

In [\[link\]](#), the hook contract is called from inline assembly and no return data is copied to memory:

```
/// @solidity memory-safe-assembly
assembly {
    pop(call(250000, hook_, 0, add(calldata_, 0x20), mload(calldata_), 0, 0))
}
```

6.3 Hook Not Called on Zero Slashed Amount

CS-SYFC2-012

At the end of a slashing, the vault owner provided hook contract is called to re-arrange network limits if necessary. If, for whatever reason, the `slashedAmount` is 0 at the point of the slash execution (possible in `VetoSlasher`), the hook is not called. This might, however, be valuable information for the hook.

Code corrected:

In [\[link\]](#) of the protocol, the `VetoSlasher` now calls the hook even if `slashedAmount` is 0.

6.4 Migration Inconsistencies

CS-SYFC2-007

`MigratablesFactory.migrate()` allows vault owners to upgrade the implementation of their vault when a new version is whitelisted. The function, however, does not make sure that intermediate versions are skipped. Since the `initialize()` function of each version is expected to upgrade the state of the previous version, such upgrades (e.g., a vault on version 1 is upgraded straight to version 3, skipping the upgrade to version 2) could lead to inconsistencies.

Code corrected:

The initial version of a contract is now passed to the `MigratableEntity._migrate()` function. Note that it is up to the implementation of the `_migrate` function to implement all the necessary steps to update the contract to a higher version.

6.5 Slash Execution Order Inconsistency

CS-SYFC2-002

`BaseSlasher.slashableStake()` calculates the stake that can be slashed at a given timestamp by subtracting all slashed amounts (starting from the timestamp until the current block's timestamp) from the available stake:

```
return stakeAmount
    - Math.min(
        cumulativeSlash(network, operator)
        - cumulativeSlashAt(network, operator, captureTimestamp, slashableStakeHints.cumulativeSlashFromHint),
        stakeAmount
    );
```

This calculation, however, does not include possible deposits that happened after the `captureTimestamp`. If another slash at a later captured timestamp was executed before the slash at `captureTimestamp` (possible with the `VetoSlasher`), the cumulative slash between `captureTimestamp` and `block.timestamp` can contain slashed deposits that were not available. Consider the following example:

1. A vault's epoch lasts 2 weeks.
2. A network onboarded on the vault has a network epoch of 1 week.
3. The vault has allocated 100% of funds to 1 operator on the onboarded network.
4. The vault uses `VetoSlasher` with a veto duration of 1 day.
5. The network starts epoch 1 at timestamp x with a stake of 1000 tokens.
6. The operator performs a slashable offense during network epoch 1. At timestamp $x + 4$ days, the network requests a slash for 1000 tokens for the operator.
7. 1000 additional tokens are deposited into the vault.
8. At timestamp $x + 7$ days, the network starts epoch 2 with a stake of 2000 tokens (the slash is not yet executed).
9. The operator, again, performs a slashable offense. At timestamp $x + 9$ days, the network requests another slash for 1000 tokens for the operator.
10. At timestamp $x + 10$ days, both slashes are executed.
11. If slash 1 is executed first, a total of 2000 tokens is slashed.

12. If slash 2 is executed first, a total of 1000 tokens is slashed. This happens because slash 2 adds 1000 tokens to the cumulative slash which is then subsequently subtracted from slash 1's stake amount during execution of slash 1.

The main problem here is that the cumulative slash can contain amounts that have been slashed on a stake that was not available at the time of `captureTimestamp` due to the fact that slashes can be executed out-of-order.

Code corrected:

In `BaseSlasher.executeSlash()` stores the `captureTimestamp` of the last executed slash and enforces that the current slash must not be older than the last executed slash. This ensures that slashes are executed in order.

```
function _checkLatestSlashedCaptureTimestamp(bytes32 subnetwork, uint48 captureTimestamp) internal view {
    if (captureTimestamp < latestSlashedCaptureTimestamp[subnetwork]) {
        revert OutdatedCaptureTimestamp();
    }
}
```

Note that the change can lead to failures when more than one `captureTimestamp` is used per network epoch (See: *Network security considerations*).

6.6 Gas Optimizations

CS-SYFC2-008

This section provides a non-exhaustive list of possible gas optimizations:

1. Some operations are redundant. For example, `Slasher.slash()` checks all opt-ins with `_checkOptIns()`, then calls `slashableStake()` which, in turn, calls `BaseDelegator.stakeAt()`. This function checks the opt-ins again. After that, `_callOnSlash()` then proceeds to call `BaseDelegator.onSlash()` which, again, calls `BaseDelegator.stakeAt()`.
2. The Network -> Vault Opt-In service is redundant as networks already signal their participation in vaults by calling `Vault.setMaxNetworkLimit()`.
3. `VetoSlasher` inherits from `AccessControlUpgradeable` but no access control is used in the contract.
4. Some functions (e.g., `VetoSlasher.executeSlash()`) are missing hints that can be used to reduce the complexity of checkpoint searches.
5. `depositWhitelist` in `VaultStorage` can be moved up (e.g., below `collateral`) to decrease the amount of used slots by 1.

Code corrected:

All mentioned optimizations have been implemented.

6.7 Missing Events



The following state-changing functions do not emit an event:

1. `Factory.whitelist()`
2. `MigratablesFactory.whitelist()`
3. `MigratablesFactory.migrate()`

Code corrected:

In all missing events have been added to functions above.

6.8 No Receiver in Claim() Function

CS-SYFC2-010

`Vault.withdraw()` allows user to set a receiver. This receiver can then, after the withdrawal waiting period, `claim()` the tokens. This design disallows the common use case of withdrawing funds to a third party without their interaction.

Code corrected:

In of the contract, the caller can specify a receiver in the `claim()` function.

6.9 Non-indexed Event Parameter

CS-SYFC2-005

For the `SetResolver` event in the `IVetoSlasher` interface, the address of the `resolver` is not indexed. Since parameters that should be searchable should be indexed, we recommend indexing the `resolver` address in the event.

Specification changed:

SymbioticFi stated:

Changed `setResolverShares()` to `setResolver()` with introduction of subnetworks and removal of shares for resolvers. Hence, the resolver is now not a “key”, but a “value”.

6.10 Original Slashed Amount in Hook

CS-SYFC2-011

At the end of a slashing, the vault owner provided hook contract is called to re-arrange network limits if necessary. The call to the hook includes the amount that has been slashed. As the `VetoSlasher` allows to reduce the slashed amount with vetoes (and also, potentially, slashes that happened after a

`captureTimestamp` but before the slash execution), it could make sense to also pass the amount that was originally requested.

Code corrected:

In `hookSlashRequest`, the `VetoSlasher` encodes the index of the slash request and passes it to the hook contract. The hook can then use this index to look up the original slash amount by calling `VetoSlasher.slashRequests()`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Vault DoS

CS-SYFC2-004

Vaults experiencing multiple full slashes could become unusable. As slashes are only adjusting stakes but not shares, the share computation on deposits can lead to inflated share decimals when deposits happen after slashes due to the following calculation:

```
assets.mulDiv(totalShares + 10 ** _decimalsOffset(), totalAssets + 1, rounding);
```

After a full slash, newly deposited assets are simply multiplied with the amount of existing shares. After multiple such repetitions, the share amount becomes larger than 256 bits.

However, we consider this unproblematic as vaults that have been slashed fully once or even twice likely will not attract new deposits anyways.

Acknowledged:

SymbioticFi is aware of this behavior but has decided to keep the code unchanged.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Network Security Considerations

Due to the open nature of the project's smart contracts, networks should make sure to consider the following points when integrating with Core:

1. Using `BaseDelegator.stake()` (or `BaseDelegator.stakeAt()` with `timestamp == block.timestamp`) in the network middleware can be susceptible to front- / backrunning. The protocol uses a checkpoint mechanism to store data such as the active stake or the network limit to storage. This allows fetching of immutable state at any point in the past. However, since checkpoints at the current block's timestamp can still be updated, transactions that happen before or after a call to `stake()` in the same block are able to change the data that the function uses to compute the stake. Hence, it is possible that the network middleware fetches a stake that is not equal to the stake at the end of the respective block. It is therefore advisable to only fetch stakes at past timestamps on-chain (through a middleware) or simply fetch stakes off-chain through an Ethereum node.
2. Rewards should be distributed based on stakes at exactly the `captureTimestamp` of a network epoch (the point at which networks calculate the available slashable stake). Otherwise, if capture timestamps can be (roughly) known in advance, the stake could be decreased at this point and then re-increased at the next possible timestamp, decreasing the slashable stake while still being eligible for rewards.
3. If a vault uses the `VetoSlasher`, requesting slashes at the latest possible timestamp (`captureTimestamp + vetoDuration - EPOCH_SIZE`) can result in the slash not being executable at all. Slashes should therefore be requested in a timely manner.
4. If a vault uses the `VetoSlasher`, it should be checked that the veto duration is considerably smaller than the epoch duration as otherwise, slashes might always fail to execute.
5. Network epochs should always be based on one `captureTimestamp`. If, for example, stakes for different operators are taken at different timestamps, then it is possible that operators can evade slashes if the `VetoSlasher` is used. This is possible due to the forced ordering of slashings.
6. Slashes should be executed in order of their `captureTimestamp` since slashes can become executable out-of-order when resolvers are removed from a subnetwork.

8.2 User Security Considerations

Due to the open nature of the project's smart contracts, vaults can be set up in ways that are not trustless. Certain configurations can allow vault owners to steal all user funds or networks to slash all user funds without justification. The following list details some points users should consider before investing in a given vault:

1. While networks can only slash the amount that has been guaranteed to them at a certain timestamp, the design of the protocol still leads to partial slashing of funds that were not participating in the staking at that timestamp. Deposits after and withdrawals before a capture timestamp are still slashed as long as they happened in the same epoch (and the epoch after in

case of deposits). It is also worth to note that such deposits most likely won't receive rewards for the given network epoch.

2. Vault owners can set any contract as `collateral` and `burner` in their vaults. If they choose to use an intermediary collateral contract (similar to `DefaultCollateral`), users should be aware that such contracts are not covered in this audit and could be used to steal funds. Additionally, as vault owners can set custom burner addresses, they have an incentive to add themselves as a network/operator, allocate all funds to them and then slash all funds. These funds would then be redirected to the vault owner controlled `burner` address. It is therefore important to validate the `collateral` and `burner` addresses before interacting with any vault.
3. Vaults using a `VetoSlasher` leave the impression that funds are safe from unjustified slashes a network might pursue due to faulty software. It should, however, be noted that this contract can be configured in a way that does not guarantee vetoes. Networks are able to add no resolvers to the slasher such that it behaves similarly to the `Slasher` contract.
4. Since `|ver2|`, vaults can add multiple subnetworks of the same network, restaking the same tokens to a network multiple times even when the `NetworkRestakeDelegator` is used. This can increase the slashing impact of malicious networks.

8.3 Vault Security Considerations

Certain configurations of vaults might be unsafe. This includes:

1. Using rebasing tokens without intermediary collateral contract. This could lead to incorrect accounting.
2. Using tokens that can transfer different amounts than anticipated (e.g., Compound III tokens) without intermediary collateral contract. This could be used to gain more shares than justified.
3. Using un-audited intermediary collateral contracts and/or burners.
4. Since `|ver2|`, special care should be taken when multiple subnetworks per network are configured as this allows restaking tokens in the same network even when `NetworkRestakeDelegator` is used.