STATE MIND

Symbiotic Core

03-06-2024 - 18-10-2024

Table of contents

1. Project brief

Title	Description
Client	Symbiotic
Project name	Symbiotic Core
Timeline	03-06-2024 - 18-10-2024

Project Log

Date	Commit Hash	Note
03-06-2024	4ebb4e15fc929c3d0141efcdd246549b63ecdee1	Initial commit
30-09-2024	fda1655c3694b9d1265e103ba1ecaa1b5e0c5ae5	Commit for the audit 2
18-10-2024	bdbb05ebd9ea6d96c671b672562ae23afcb2123b	Re-audit

Short Overview

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.

The Symbiotic protocol consists of 5 interconnected components:

- Collateral: The security layer of Symbiotic. Collateral is an abstraction used to represent underlying on-chain assets, which are chain- and asset-agnostic.
- Vaults: The (re)staking layer of Symbiotic. Delegation of collateral to operators across networks is handled by vaults that can be curated in a configurable manner
- Operators: Operators in Symbiotic are defined as entities running infrastructure for networks. In Proof-of-Stake, successful staking providers have established a brand identity and operate across networks. The Symbiotic protocol creates a registry of operators, as well as enabling them to opt-in to networks and receive economic backing from restakers through vaults.
- Resolvers: Resolvers are entities or contracts tasked to pass or veto slashing penalties incurred by operators on networks to which they provide services. They are agreed upon by vaults – representing providers of economic security – and the networks they provide security for.
- Networks: Networks in Symbiotic are defined as protocols that require a distributed set of node operators to provide trust-minimized services. Symbiotic enables network builders to define, control, and adapt their methodology for onboarding, incentivizing, and penalizing operators and their delegators (providers of economic collateral).

* The audit was conducted in two stages. After the first iteration, the core architecture was modified, and part of the logic was moved to new modules — Delegators and Slashers. Contracts related to reward distribution were moved to a separate repository. The second audit covered the contracts from updated core repository and the revised architecture.

Project Scope

The audit covered the following files:

- VaultConfigurator.sol
- VaultFactory.sol
- Subnetwork.sol
- MigratableEntityProxy.sol
- Entity.sol
- BaseSlasher.sol
- OptInService.sol
- NetworkRestakeDelegator.sol
- NetworkRegistry.sol
- VaultStorage.sol
- SlasherFactory.sol
- Checkpoints.sol
- StaticDelegateCallable.sol
- MigratablesFactory.sol
- MigratableEntity.sol
- Slasher.sol
- NetworkMiddlewareService.sol
- FullRestakeDelegator.sol
- Vault.sol
- DelegatorFactory.sol
- ERC4626Math.sol
- Factory.sol
- Registry.sol
- OperatorRegistry.sol
- VetoSlasher.sol
- MetadataService.sol
- BaseDelegator.sol
- VaultTokenized.sol

2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	3 (2 fixed, 1 acknowledged)
Informational	30 (13 fixed, 17 acknowledged)
Total	33 (15 fixed, 18 acknowledged)

4. Conclusion

During the audit of the codebase, 33 issues were found in total:

- 3 medium severity issues (2 fixed, 1 acknowledged)
- 30 informational severity issues (13 fixed, 17 acknowledged)

The final reviewed commit is bdbb05ebd9ea6d96c671b672562ae23afcb2123b

5. Findings report

MEDIUM-01	Reorg attack during **Vault** deployment	Fixed at: 985dccd

Description

Lines: MigratablesFactory.sol#L53-L58

The **MigratablesFactory** uses **create** instead of **create2**. The **create** uses the address of the factory and nonce to compute an address where the contract will be deployed. This is susceptible to reorg attacks.

The scenario:

1. Alice sends a transaction to deploy a **Vault**.
2. Bob sees Alice's transaction and sends a transaction to deploy a **Vault** with different arguments. For example, with another **owner**.
3. Alice's transaction is confirmed and **Vault** is deployed at **address1**.
4. Bob's transaction is confirmed and **Vault** is deployed at **address2**.
5. Alice sees that the **Vault** has been deployed with her arguments and deposits collateral.
6. A reorg occurs and Bob's transaction comes before Alice's transaction and Bob's **Vault** is at the **address1** and Alice's **Vault** is at the **address2**.
7. Bob can perform a slash without delay and burn Alice's deposit.

Other types of attacks are also possible because Bob controls the contract.

While reorgs on Ethereum are mostly of depth 1, on an L2 like Polygon reorgs depth can be very high.

Recommendation

We recommend using **create2** to deploy **MigratableEntityProxy** and the proxy constructor arguments for the salt creation.

MEDIUM-02	Incorrect amount of slashing and too early execution of slashing	Acknowledged

Description

Lines:

- Vault.sol#L275
- Vault.sol#L333

The documentation specifies **executionDuration** starts after **vetoDuration** IVaultStorage.sol#L164C75-L164C98 and is calculated according to the note Vault.sol#L305.

However, in the case **resolver == address(0) && Vault.vetoDuration > 0**, the request can be executed without waiting for the **Vault.vetoDuration**.

Also, **slashableAmount** and **SlashingRequest.amount** are calculated based on the start slashing request execution timestamp. Therefore, if the request can be executed without waiting, the calculation is incorrect.

Recommendation

We recommend waiting for **vetoDuration** anyway if installed in **Vault**.

If skipping **vetoDuration** is the expected behavior, adjust the calculations.

```
uint256 slashableAmount_ = slashableAmountIn(network, resolver, operator, resolver == address(0) ? 0 : vetoDuration);
```

Client's comments

> It is an intended behavior not to reduce networks' guarantees about their stakes. There is a resolver mechanism to veto malicious slashings from the funds' safety side.

MEDIUM-03	The Checkpoints.upperLookupRecentCheckpoint() function for Trace256 returns wrong value	Fixed at: e6a46bf

Description

Lines:

- Checkpoints.sol#L286
- Checkpoints.sol#L290

The **Checkpoints.at()** function for **Trace256** already dereferences **self._values[checkpoint._value]** when it returns a **Checkpoint256** struct. So when we access **checkpoint._value**, we're already getting the correct value, not an index into **self._values**. There is a mistake in the code where **self._values[checkpoint._value]** is being dereferenced twice unnecessarily.

```
Checkpoint256 memory checkpoint = at(self, hint);
if (checkpoint._key == key) {
    return (true, checkpoint._key, self._values[checkpoint._value], hint);
}

if (checkpoint._key < key && (hint == length(self) - 1 || at(self, hint + 1)._key > key)) {
    return (true, checkpoint._key, self._values[checkpoint._value], hint);
}
```

Recommendation

We recommend correcting return values.

```
if (checkpoint._key == key) {
    return (true, checkpoint._key, checkpoint._value, hint);
}

if (checkpoint._key < key && (hint == length(self) - 1 || at(self, hint + 1)._key > key)) {
    return (true, checkpoint._key, checkpoint._value, hint);
}
```

INFORMATIONAL-01	New deposits made after Vault.requestSlash() penalized	Acknowledged

Description

Lines:

- Vault.sol#L351
- Vault.sol#L357
- Vault.sol#L371

Users can deposit and withdraw collateral to the **Vault**.

In general, users expect the following behavior:

- The user's stake will be slashed if the funds were deposited before **Vault.requestSlash()** / incorrect operator behavior.
- The user's stake will not be slashed if the funds were deposited after **Vault.requestSlash()** / incorrect operator behavior.

Vault reduces **activeSupply** by **activeSlashed**. This means that all deposits will be penalized, even those deposited after **Vault.requestSlash()**.

This calculation is not fair for users. Also, we didn't find any mention that this behavior is expected.

Recommendation

We recommend providing information about active slash requests to the **Vault** so that external contracts can handle this case and the UI application displays extra information to the user about the risks of reducing the deposit.

An alternative option is to deny new deposits if there is an active slashing request.

Client's comments

> It is an intended behavior.

INFORMATIONAL-02	There are no sanity checks	Fixed at: 80999e5

Description

Lines:

- Vault.sol#L561 – **collateral** zero address check
- Vault.sol#L567 – **executeDuration** zero value check
- Vault.sol#L532 – **account** zero address check
- Vault.sol#L244 – **recipient** zero address check
- Vault.sol#L210 – **claimer** zero address check
- Vault.sol#L180 – **onBehalfOf** zero address check
- VaultStorage.sol#L179-L183– **networkRegistry**, **networkMiddlewareService**, **networkVaultOptInService**, **operatorVaultOptInService**, **operatorNetworkOptInService** zero address checks
- MigratableEntity.sol#L25 – **factory** zero address check
- DefaultRewardsDistributor.sol#L154 – **recipient** zero address check

Recommendation

We recommend adding these checks.

INFORMATIONAL-03	VaultStorage.ADMIN_FEE_BASE can break rewards claiming	Acknowledged

Description

Lines:

- Vault.sol#L507
- DefaultRewardsDistributor.sol#L141

The **Vault.setAdminFee()** function allows to set the admin fee up to **ADMIN_FEE_BASE**. When installing **adminFee == 100%**, the **VaultOwner** can take all the rewards issued by the network and users will not be able to claim them.

Recommendation

We recommend adding a limit for **adminFee** < 100%.

Client's comments

> We don't think limiting the admin fee's bounds is necessary. However, we've implemented a fix for not adding the redundant rewards distribution request.

INFORMATIONAL-04	Vault.totalSupplyIn() can be manipulated if futureEpoch > epoch + 1	Fixed at: 647bd8d

Description

Line: Vault.sol#L33-L34

Vault.totalSupplyIn() is available for calls to external contracts.
However, if **duration > Vault.epochDuration * 2** attacker can manipulate the result value. To do this, the attacker can make a call to **Vault.withdraw()** and **Vault.activeSupply()** will be decreased. Due to the long **duration**, the attacker can deposit funds again. Therefore, external contracts should use **Vault.totalSupplyIn()** with caution.

Recommendation

We recommend extending documentation for **Vault.totalSupplyIn()** for cases with **duration > Vault.epochDuration * 2**.
The alternating approach reverts if **duration > Vault.epochDuration**. Because of the architecture, **Vault** can't provide a reliable supply for the **duration > Vault.epochDuration**.

INFORMATIONAL-05	Unused error from IVault	Fixed at: 5658030

Description

Line: IVault.sol#L7

The **IVault** interface contains the **IVault.AlreadyClaimed()** error which is not used in any revert.

Recommendation

We recommend removing this error from the implementation or finding a use for it.

INFORMATIONAL-06	Checkpoints.upperLookupRecent() sanity check	Acknowledged

Description

After the network's rewards distribution, the depositor can call DefaultRewardsDistributor.claimRewards() with additional **uint32[] calldata activeSharesOfHints**. In case when at least one hint exceeds **_activeSharesOf[account].length()**, the whole transaction will revert instead of ignoring the hint.

Recommendation

We recommend adding the sanity check in **Checkpoints.upperLookupRecent()**.

```
function upperLookupRecent(Trace256 storage self, uint48 key, uint32 hint) internal view returns (uint256) {
    if (hint >= self._trace.length()) {
        return upperLookupRecent(self, key);
    }
    Checkpoint256 memory hintCheckpoint = at(self, hint);
    ...
}
```

Client's comments

It is an intended behavior.

INFORMATIONAL-07	Using SafeERC20 function during collateral transfers	Fixed at: 4c4d096

Description

Line:

- Vault.sol#L189
- Vault.sol#L257

Interaction with the **collateral** token occurs through standard **IERC20** methods. Based on the definition of interface **collateral** returns bool values, there is no guarantee that cases when it returns **false** instead of revert will be processed correctly.

Recommendation

We recommend using **SafeERC20** library when interacting with the **collateral** token.

INFORMATIONAL-08	Making calculations after checks	Fixed at: 2c70f0c

Description

Line: Vault.sol#L275

At Vault.sol#L275 **slashableAmount_** is calculated and is checked for zero, after that there are checks that are not related to calculated value which could revert execution.

Recommendation

We recommend calculating **slashableAmount_** value after the checks in different services at Vault.sol#L281-L299.

INFORMATIONAL-09	Event DistributeReward is emitted using the ambiguous amount	Acknowledged

Description

Line: DefaultRewardsDistributor.sol#L147

The **amount** parameter in the **DistributeReward** is supposed to be the amount of tokens to be distributed (admin fee is excluded). But in fact the admin fee is included when emitting the event. This will confuse the network as it may expect all **amount** to be distributed between stakers.

Recommendation

We recommend specifying the **amount** in docs.

```
/**
...
* @param amount amount of tokens (including admin fee)
*/
event DistributeReward(address indexed network, address indexed token, uint256 amount, uint48 timestamp);
```

Client's comments

The **IRewardsDistributor** interface is a recommended one for all the custom implementations. However, some of them know nothing about admin fees and, therefore, this comment is redundant.

INFORMATIONAL-10	Adding more strict limits on the reward timestamp	Acknowledged

Description

Line: DefaultRewardsDistributor.sol#L112

The reward **timestamp** is checked to be less than the current timestamp because the system should distribute rewards for epochs that were passed. However, there are no limitations for the beginning of distribution.

Recommendation

We recommend checking additionally the reward **timestamp** to be greater than Vault's **epochDurationInit**.

```
...
if (timestamp <= IVault(VAULT).epochDurationInit() || timestamp >= Time.timestamp()) {
    revert InvalidRewardTimestamp();
}

...
```

Client's comments

It'll still revert later on caching, and the **epochDurationInit** was considered a helper var to support epochs' duration updates in future versions. In other words, currently, it means the vault's creation time, but later it could be changed.

INFORMATIONAL-11	Checking all the necessary conditions in Vault when adding a reward checkpoint	Acknowledged

Description

Line: DefaultRewardsDistributor.sol#L120
Condition at lines DefaultRewardsDistributor.sol#L120-L122 checks that there was some **activeSupply** in the Vault at the **timestamp** moment to participate in staking for the **network**. However, this is not a sufficient condition to check eligibility in staking because it doesn't take into account **network-resolver** and **operator-network** limits.
In function Vault.minStakeDuring() three params are used to retrieve available stake amount (**activeSupply**, **networkResolverLimit** and **operatorNetworkLimit**). Based on the return value of the function **Vault.minStakeDuring()**, the network decides to include **Vault** and **operator** to participate in staking. There may be a case when **Vault** has **activeSupply**, but limits are set to zero or become zero, then it won't be eligible for validating network and shouldn't receive any rewards for that period. So during rewards distribution limits at **timestamp** should also be considered alongside checking **activeSupplyAt(timestamp)**.

Recommendation

We recommend adding checks for **network-resolver** and **operator-network** limits when distributing rewards.

Client's comments

> Rewards distribution is processed by the network middleware. Hence, validating the distribution timestamp is its responsibility (theoretically, there could be cases when the network or mock network doesn't have a stake but wants to distribute rewards as incentives).

INFORMATIONAL-12	Manipulation in the distribution of rewards	Acknowledged

Description

Lines:

- DefaultRewardsDistributor.sol#L117-L124
- DefaultRewardsDistributor.sol#L191

The current reward calculation mechanism uses timestamps to determine the reward amounts. This method is not fair for users and is also subject to manipulation.
Examples:

- Network developers can predict the next **timestamp** that will be used for the next distribution of rewards and make a pre-deposit.
- Block producers or nodes may intentionally have to delay reward distribution transactions to make a pre-deposit.

A pre-deposit will allow an attacker to claim most of the awards, although the funds didn't provide network security for the period.

Recommendation

We recommend calculating rewards based on periods and adding offset from the current timestamp to distribute rewards.

Client's comments

> It is stated that "the funds didn't provide network security for the period." However, it seems to us that it is incorrect as the funds were deposited into the vault and became slashable. Hence, the rewards are distributed fairly according to the gotten risks.

INFORMATIONAL–13	Future timestamps are misleading	Acknowledged

Description

Lines:

- Vault.sol#L55
- VaultStorage.sol#L224
- VaultStorage.sol#L238
- VaultStorage.sol#L252
- VaultStorage.sol#L259

The contract returns valid values when specifying a **timestamp** from the past.
When an external contract passes **timestamp > block.timestamp**, it expects the value to be calculated for this timestamp, however, this works similarly to the query for the **block.timestamp**.
The **Vault** uses the concept of checkpoints similar to OZ Votes which doesn't provide an API for getting values from the future.

Recommendation

We recommend removing the possibility of calling functions with future timestamps.

Client's comments

> It is an intended behavior. OZ also has GovernorVotesQuorumFraction where there is no such revert.

INFORMATIONAL–14	Limits update	Fixed at: 32fcc7e

Description

Lines:

- Vault.sol#L375
- Vault.sol#L379

Limits are updated only when **slashableAmount** is not zero, but applying **nextNetworkResolverLimit** and **nextOperatorNetworkLimit** won't affect any calculations or slashing even if there is no amount to slash.

Recommendation

We recommend always updating limits for state consistency.

INFORMATIONAL-15	Mitigating the effects of cross slashings	Acknowledged

Description

Lines:

- Vault.sol#L385
- Vault.sol#L388

At the end of Vault.executeSlash function limits are decreased by **slashedAmount**. **slashedAmount** could be less than **request.amount** in two cases when **totalSupply** is decreased due to cross slashings or limits of certain **network-resolver**, **operator-network** pairs are decreased due to continuous slashings of the same operator or resolver. To minimize the effects of cross or continuous slashings limits could be decreased by **request.amount**. This will allow us to correctly account for limits and eject malicious operators in cases when cross slashings lower the desired **slashedAmount**.

Recommendation

We recommend decreasing limits by the **request.amount** and also considering architectural changes that will mitigate cross slashing effects when executing slashings and accounting states for networks and operators.

```
...
if (networkResolverLimit_ != type(uint256).max) {
    _networkResolverLimit[request.network][request.resolver].amount = networkResolverLimit_ -
Math.min(networkResolverLimit_, request.amount);
}
if (operatorNetworkLimit_ != type(uint256).max) {
    _operatorNetworkLimit[request.operator][request.network].amount = operatorNetworkLimit_ -
Math.min(operatorNetworkLimit_, request.amount);
}
...
```

Client's comments

It is an intended behavior.

INFORMATIONAL-16	Withdrawals are not taken into account when distributing and claming rewards	Acknowledged

Description

Lines:

- DefaultRewardsDistributor.sol#L103
- DefaultRewardsDistributor.sol#L153

Based on rewards distribution and claiming logic only users with **activeSupply** at a certain **timestamp** are eligible to receive rewards. Users with non-zero withdrawals also participate in staking (because withdrawals are also object for slash), therefore they also should be able to claim rewards.

Recommendation

We recommend taking into account users with non-zero withdrawals during rewards distribution and claiming.

Client's comments

Withdrawals are slashable because they participated in staking in the past. Hence, they could've been rewarded in the past.

INFORMATIONAL-17	The BaseDelegator contract is not abstract	Fixed at: 1c1eeec

Description

Line: BaseDelegator.sol#L18

The **BaseDelegator** core virtual functions have an empty implementation BaseDelegator.sol#L229-L238. Therefore, it does not make sense to deploy **BaseDelegator** separately, and a potentially deployed contract is non-functional.

Recommendation

We recommend making a **BaseDelegator** abstract.

INFORMATIONAL-18	Calling AccessControlUpgradeable.hasRole() is redundant	Fixed at: 77ddd76

Description

Lines:

- FullRestakeDelegator.sol#L164
- FullRestakeDelegator.sol#L176
- NetworkRestakeDelegator.sol#L197
- NetworkRestakeDelegator.sol#L209
- OperatorSpecificDelegator.sol#L142

Calling **AccessControlUpgradeable.hasRole()** is redundant because **AccessControlUpgradeable._grantRole()** returns **false** AccessControlUpgradeable.sol#L205 if the role has already been assigned.

Recommendation

We recommend using the return value of the **AccessControlUpgradeable._grantRole()**.

```
if (!_grantRole(...)) {
  revert DuplicateRoleHolder();
}
```

INFORMATIONAL-19	Skipping slashing incidents can be an abuse of the network	Acknowledged

Description

Lines:

- VetoSlasher.sol#L157
- BaseSlasher.sol#L115

VetoSlasher allows the network to register slash requests. The resolver can execute or veto the request. The network may have several sequential requests to process. If the network executes one of the requests that is not the first, then all older unexecuted requests become invalid and cannot be executed. The request that the network wants to skip should be vetoed, not just skipped.

Recommendation

We recommend processing requests only sequentially because the network confirms the presence of an incident when the request is created. The contract's state machine can support the transition to the final state of each request. Thus, the network can also process all time-expired requests and mark them as completed.

Client's comments

> We assume that the network's middleware will handle the proper sequencing.

INFORMATIONAL-20	Call inconsistency IBaseDelegator.onSlash() for slashedAmount == 0	Fixed at: bf3c8fe

Description

Lines:

- VetoSlasher.sol#L178
- Slasher.sol#L45
- Slasher.sol#L53

VetoSlasher makes a call to **IBaseDelegator.onSlash()** if **slashedAmount == 0**, however, **Slasher** reverts if **slashedAmount == 0** and therefore doesn't make a call to **IBaseDelegator.onSlash()**.

Recommendation

We recommend using consistent logic to call **IBaseDelegator.onSlash()**.

INFORMATIONAL-21	Unused error in the IVetoSlasher contract	Fixed at: ef59705

Description

Line: IVetoSlasher.sol#L15

The **VetoSlasher** contract interface contains an unused error **IVetoSlasher.VaultNotInitialized()**.

Recommendation

We recommend deleting or implementing an unused error.

INFORMATIONAL-22	Slash request completion check can be moved earlier	Acknowledged

Description

Lines:

- VetoSlasher.sol#L159
- VetoSlasher.sol#L221

In the **VetoSlasher.executeSlash()** and **VetoSlasher.vetoSlash()**, the check to see if a slash request has already been completed is performed later in the function, after several other checks and operations. It makes unnecessary computations and gas consumption in this case.

Recommendation

We recommend moving the **request.completed** check immediately after loading request from storage.

Client's comments

> We assume that the frequency of this error is less than for others. Therefore, it may be optimal to check others first.

INFORMATIONAL-23	Missing timestamp validation in slash function	Acknowledged

Description

Lines:

- Slasher.sol#L38
- VetoSlasher.sol#L93

In the functions **Slasher.slash()** and **VetoSlasher.requestSlash()**, there is no validation to ensure that the **captureTimestamp** is greater than the latest slashed capture timestamp. Although **BaseSlasher.slashableStake()** checks this case, it will revert with an **InsufficientSlash** error instead of the more appropriate **InvalidCaptureTimestamp** error.

Recommendation

We recommend adding check **captureTimestamp < latestSlashedCaptureTimestamp[subnetwork][operator]** before calling **BaseSlasher.slashableStake()**.

Client's comments

> We don't want to increase gas costs and bytecode size in this case.

INFORMATIONAL-24	Registry._addEntity() optimization	Acknowledged

Description

Lines:

- NetworkRegistry.sol#L13
- OperatorRegistry.sol#L13

EnumerableSet.add() returns the result of adding EnumerableSet.sol#L239-L240. The **Registry._addEntity()** can also return the result of addition. In this way, the contract can use the result instead of a redundant **Registry.isEntity()** call.

Recommendation

We recommend returning the result of adding from **Registry._addEntity()** and using it instead of **Registry.isEntity()** calls.

```
function registerNetwork() external {
    if (!_addEntity(msg.sender)) {
        revert NetworkAlreadyRegistered();
    }
}
```

Client's comments

> In most cases, we use it for new addresses. Hence, will not proceed with this.

INFORMATIONAL-25	Vault.isInitialized() doesn't work correctly for some deployment scenarios	Acknowledged

Description

Line: Vault.sol#L37

The current statements are insufficient because the deployer can manually deploy the **Vault**, call **Vault.setDelegator()**, and **Vault.setSlasher()** without calling **MigratableEntity.initialize()**, but **Vault.isInitialized()** returns **true**.

Recommendation

We recommend extending statement to **isDelegatorInitialized && isSlasherInitialized && collateral != address(0)**.

Client's comments

This case is not relevant because **Vault** is deployed only by **MigratableFactory**, which performs the initialization.

INFORMATIONAL-26	Vault.onSlash() doesn't return slashableAmount	Fixed at: def15e0

Description

Vault.onSlash() accepts **slashedAmount**, but has an implicit calculation when the amount of final slashing may be less Vault.sol#L236, Vault.sol#L247.

Recommendation

We recommend returning the final slashed amount from the **Vault.onSlash()** function or describing implicit behavior in the function's documentation.

INFORMATIONAL-27	Additional checks during Vault initialization	Fixed at: 8dda1d3

Description

Lines:

- VaultTokenized.sol#L105 - **collateral != address(this)** because **VaultTokenized** is **ERC20** token.
- Vault.sol#L472-L473 - **Vault.setDepositLimit()** disallow the invariant **depositLimit != 0 && !isDepositLimit** Vault.sol#L331. However, if **params.defaultAdminRoleHolder != address(0) && params.isDepositLimitSetRoleHolder != address(0)** the invariant still needs to be checked.

Recommendation

We recommend adding these checks.

INFORMATIONAL-28	Inconsistent key size in Checkpoints._upperBinaryLookup()	Fixed at: d225636

Description

Line: Checkpoints.sol#L358

The **key** parameter is declared as **uint96**, but throughout the contract, keys are used as **uint48**.

Recommendation

We recommend correcting **key** type to be consistent with the rest of the contract.

INFORMATIONAL-29	Insufficient slashing amount	Acknowledged

Description

Lines:

- FullRestakeDelegator.sol#L126
- FullRestakeDelegator.sol#L138
- NetworkRestakeDelegator.sol#L157
- NetworkRestakeDelegator.sol#L171
- BaseSlasher.sol#L120
- OperatorSpecificDelegator.sol#L104
- OperatorSpecificDelegator.sol#L116

During slashing, the amount is calculated based on **activeStake** and doesn't consider the withdrawal amounts. However, **Vault** takes into account **activeStake** + withdrawals when calculating the slashable amount Vault.sol#L235, Vault.sol#L246. Thus, the estimated amount of slashing is less than it can be.

Recommendation

We recommend considering withdrawal amounts when calculating the slashable amount for delegators in the same way as **Vault**.

Client's comments

It is an intended behavior.

INFORMATIONAL-30	Lost signatures can be used by an attacker	Acknowledged

Description

The current structures of **OPT_IN_TYPEHASH** and **OPT_OUT_TYPEHASH** messages do not contain information about the calling address. The operator can pass the signature to a trusted participant. However, if the signature is stolen, the attacker can perform **OptInService.optIn()** or **OptInService.optOut()** with it, which is potentially dangerous behavior.

Recommendation

We recommend adding the **address authorizedSender** field to **OPT_IN_TYPEHASH** and **OPT_OUT_TYPEHASH**. Therefore, the operator can specify a specific address to control the caller. The contract can also accept **address(0)** if it is impossible to determine the calling contract's address when creating the signature.

Client's comments

We'll leave the current simplified implementation.

STATE
MIND