SYMBIOTIC

Relay Contracts

Security Assessment Report

Version: 2.1

September, 2025

Contents

Introduction 2
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary 3
- Scope . . . 3
- Approach . . . 3
- Coverage Limitations . . . 3
- Findings Summary . . . 3

Detailed Findings 5

Summary of Findings 6
- Missing Multi-Chain Voting Power Distribution Controls . . . 7
- Potential Contract Lock Due To Excessively High Epoch Duration . . . 8
- Missing Genesis Validation In `setGenesis()` Function . . . 9
- Potential DoS Through Malicious Epoch Value In `commitValSetHeader()` . . . 10
- Bytes Conversion Functions Only Encode Wrapped Value, Not Entire Struct . . . 12
- No Mechanism To Reduce Operator Jail Duration . . . 14
- Token Weight Functions Return Default Values For Unregistered Tokens . . . 15
- Unsafe Internal `_setEpochDuration()` Function Allows Data Corruption . . . 16
- `invert()` Function Reverts When Value Is Zero . . . 17
- Conflicting `BASE_DECIMALS` Constants . . . 18
- Missing Public Function To Update Slashing Data . . . 19
- `getOperatorJailedUntil` Returns Past Timestamps For Non-Jailed Operators . . . 20
- Potential Integer Underflow In `nonSignersLength` Calculation . . . 21
- Missing Constructors With `_disableInitializers()` In Upgradeable Contracts . . . 22
- Inappropriate Visibility On `__Network_init()` . . . 25
- Miscellaneous General Comments . . . 26

A Vulnerability Severity Classification 28

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Symbiotic components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Symbiotic components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open / closed / resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Symbiotic components in scope.

Overview

The Symbiotic Core protocol provides a modular onchain framework for creating flexible staking solutions, including collateral choice, rewarding, slashing and redistribution logic. It is already adopted by dozens of teams with over 15 live networks secured by over $1.5bn in TVL on Ethereum.

The Symbiotic Relay project, of which the relay contracts form a key component, serves as an extension to the core protocol that radically simplifies integrating Symbiotic's universal staking primitives and enables leveraging stake across any execution environment, expanding the design space for multichain-native decentralized protocols.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Symbiotic Relay Contracts repository.

The scope of this time-boxed review was strictly limited to files at commit 5dd871b.

The fixes of the identified issues were assessed at commit 9759b26.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 16 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Low: 4 issues.
- Informational: 11 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Symbiotic components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SYM-01	Missing Multi-Chain Voting Power Distribution Controls	Medium	Closed
SYM-02	Potential Contract Lock Due To Excessively High Epoch Duration	Low	Closed
SYM-03	Missing Genesis Validation In `setGenesis()` Function	Low	Resolved
SYM-04	Potential DoS Through Malicious Epoch Value In `commitValSetHeader()`	Low	Closed
SYM-05	Bytes Conversion Functions Only Encode Wrapped Value, Not Entire Struct	Low	Resolved
SYM-06	No Mechanism To Reduce Operator Jail Duration	Informational	Closed
SYM-07	Token Weight Functions Return Default Values For Unregistered Tokens	Informational	Resolved
SYM-08	Unsafe Internal `_setEpochDuration()` Function Allows Data Corruption	Informational	Closed
SYM-09	`invert()` Function Reverts When Value Is Zero	Informational	Resolved
SYM-10	Conflicting `BASE_DECIMALS` Constants	Informational	Resolved
SYM-11	Missing Public Function To Update Slashing Data	Informational	Resolved
SYM-12	`getOperatorJailedUntil` Returns Past Timestamps For Non-Jailed Operators	Informational	Resolved
SYM-13	Potential Integer Underflow In `nonSignersLength` Calculation	Informational	Resolved
SYM-14	Missing Constructors With `_disableInitializers()` In Upgradeable Contracts	Informational	Closed
SYM-15	Inappropriate Visibility On `__Network_init()`	Informational	Resolved
SYM-16	Miscellaneous General Comments	Informational	Resolved

SYM-01	Missing Multi-Chain Voting Power Distribution Controls		
Asset	`modules/valset-driver/ValSetDriver.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

There are no configuration options to limit the impact of multi-chain balancing for voting power, and this could cause problems in several scenarios:

1. If the total voting power is such that, to reach `minInclusionVotingPower`, one chain must be up, then the implementing network will be down whenever a chain is down. This could be an intentionally implemented feature, but not all implementing networks might wish for it. There is also the consequence that the network is as stable as its most unstable chain.
2. In the opposite case, if `minInclusionVotingPower` can be fulfilled entirely by the voting power of one chain, then it renders staking on other chains less desirable and relevant, and may not give the distributed staking configuration that the implementing network desires.
3. Most significantly, there is no cap on the voting power of a chain. So, if one chain's voting power is compromised (by, for example, a fundamental failure outside the scope of these contracts, like a vital private key compromise), then that chain can claim any voting power, making other chains irrelevant. As the quorum is defined in terms of voting power, this scenario could allow a single validator to claim enough voting power to commit a new `ValSetHeader` with just its own signature, which could then be successfully committed on all chains.

   This is mitigated by the fact that the `VotingPowerProvider` would need to be significantly compromised on one chain. However, as the SDK is designed to be usable on any EVM chain, it is conceivable that an implementing network might wish to deploy on an experimental or new chain whilst providing a secure settlement voting power base on a more established chain. In this scenario, the system as a whole would actually be as secure as the newer, experimental chain.

Recommendations

Consider implementing configuration options in the `ValSetDriver` `Config` struct that limit settlements by:

- Capping the voting power of individual chains
- Requiring a certain number of chains to provide voting power
- Requiring a minimum level of voting power from each participating chain

These options would need to be processed in the signature verifiers.

Resolution

The issue was acknowledged by the development team.

SYM-02	Potential Contract Lock Due To Excessively High Epoch Duration		
Asset	`modules/valset-driver/EpochManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `setEpochDuration()` function lacks validation for maximum epoch duration values, potentially allowing the contract to be locked in the current epoch indefinitely if an excessively high duration is set.

When `setEpochDuration()` is called, it schedules the new epoch duration to take effect at the next epoch start calculated by `getNextEpochStart()`. If an extremely large epoch duration is accidentally set then, once that epoch begins, `getNextEpochStart()` will return a timestamp impossibly far in the future. This creates a scenario where the contract becomes permanently locked in the current epoch since no reasonable timeline could reach the next epoch start.

While `setEpochDuration()` can still be called to add new entries to the storage arrays `_epochDurationDataByTimestamp` and `_epochDurationDataByIndex`, these entries would be scheduled for activation at timestamps far beyond any practical timeframe, effectively making them unreachable. This means the contract would be unable to progress to subsequent epochs, breaking any functionality that depends on epoch transitions.

This issue is mitigated if there is an external function which allows administrator access to this internal function:

EpochManager._setEpochDuration()

```
function _setEpochDuration(
        uint48 epochDuration,
        uint48 epochDurationTimestamp,
        uint48 epochDurationIndex
    )
```

In this case, it is possible to correct the misconfiguration.

There is also a more acute version of this issue if the epoch duration is equal to `uint48.max`, or close to it, as this will cause calls to `getNextEpochStart()` to revert from an integer overflow, as it adds the current epoch start to the duration and returns the result in a `uint48`. This would also cause `setEpochDuration()` to revert, as it calls this function.

Recommendations

Add validation to prevent excessively high epoch durations.

Resolution

The issue was acknowledged by the development team.

SYM-03	Missing Genesis Validation In `setGenesis()` Function		
Asset	`modules/settlement/Settlement.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `setGenesis()` function lacks proper validation to ensure it can only be called to set the initial genesis validator set header, allowing it to potentially overwrite existing committed headers.

The `setGenesis()` function directly calls `_setValSetHeader()` without verifying that no genesis has been previously set. While the function is protected by the `checkPermission` modifier, it does not validate that this is actually the first validator set header being committed. This creates a risk where an authorized caller could accidentally or maliciously use `setGenesis()` to commit non-genesis headers, bypassing the signature verification requirements that would normally be enforced through `commitValSetHeader()`.

The function should ensure that it can only be used for its intended purpose of setting the initial validator set state, not for updating existing validator sets, which should go through the proper signature verification process.

Settlement.setGenesis()

```
function setGenesis(
    ValSetHeader calldata header,
    ExtraData[] calldata extraData
) external checkPermission {
    _setValSetHeader(header, extraData); //@audit Missing validation that genesis not already set
}
```

Recommendations

Add validation checks to ensure `setGenesis()` can only be called when no genesis has been set.

Resolution

The development team stated that multiple calls to the `setGenesis()` function were intended functionality and added a comment to this effect at commit e35f836.

SYM-04	Potential DoS Through Malicious Epoch Value In `commitValSetHeader()`		
Asset	`modules/settlement/Settlement.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `_setValSetHeader()` function only validates that the new epoch is greater than the last committed epoch, allowing malicious validators to commit a header with an extremely large epoch value (such as `uint48.max`) that would effectively disable the contract.

Settlement._setValSetHeader()

```
function _setValSetHeader(ValSetHeader calldata header, ExtraData[] calldata extraData) internal virtual {
    if (header.version != VALIDATOR_SET_VERSION) {
        revert Settlement_InvalidVersion();
    }

    uint48 lastCommittedHeaderEpoch = getLastCommittedHeaderEpoch();
    if (lastCommittedHeaderEpoch > 0) {
        if (header.epoch <= lastCommittedHeaderEpoch) {
            revert Settlement_InvalidEpoch();
        } //@audit Missing validation for maximum epoch increment
    } else if (header.epoch == 0 && isValSetHeaderCommittedAt(0)) {
        revert Settlement_ValSetHeaderAlreadyCommitted();
    }
```

In the validation logic within `_setValSetHeader()`, the check `if (header.epoch <= lastCommittedHeaderEpoch)` prevents epochs from going backwards, but does not limit how far forward they can jump. If a validator set header with an epoch of `type(uint48).max` were ever committed, accidentally or maliciously, it would not be possible for future legitimate headers to be committed since no epoch value could be greater than the maximum.

While this attack requires the accidental or malicious commitment of a bad `ValSetHeader()`, the lack of epoch increment validation creates an unnecessary attack vector that could permanently disable the settlement contract's ability to process new validator set headers.

Recommendations

Implement stricter epoch validation to prevent large jumps. Consider allowing bounded increases.

Resolution

A check was added to the function `commitValSetHeader()` to prevent large epoch jumps.

Settlement.commitValSetHeader()

```
function commitValSetHeader(
    ValSetHeader calldata header,
    ExtraData[] calldata extraData,
    bytes calldata proof
) public virtual {
    uint48 valSetEpoch = getLastCommittedHeaderEpoch();
    if (header.previousHeaderHash != getValSetHeaderHashAt(valSetEpoch)) {
        revert Settlement_InvalidPreviousHeaderHash();
    }
    // @audit Checks that epoch is incremented
    if (header.epoch != valSetEpoch + 1) {
        revert Settlement_InvalidEpoch();
    }
    if (
        !verifyQuorumSig(
            abi.encode(
                hashTypedDataV4CrossChain(
                    keccak256(
                        abi.encode(
                            VALSET_HEADER_COMMIT_TYPEHASH,
                            SUBNETWORK(),
                            header.epoch,
                            keccak256(abi.encode(header)),
                            keccak256(abi.encode(extraData))
                        )
                    )
                )
            ),
            getRequiredKeyTagFromValSetHeaderAt(valSetEpoch),
            getQuorumThresholdFromValSetHeaderAt(valSetEpoch),
            proof
        )
    ) {
        revert Settlement_VerificationFailed();
    }


    _setValSetHeader(header, extraData);


    emit CommitValSetHeader(header, extraData);
}
```

This was added at commit e6c1705.

`_setValSetHeader()` does not check for large epoch jumps, however the development team stated that issue is now "fixed due to redesign of the off-chain piece".

SYM-05	Bytes Conversion Functions Only Encode Wrapped Value, Not Entire Struct		
Asset	libraries/keys/KeyBlsBn254.sol, libraries/keys/KeyEcdsaSecp256k1.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Key serialisation functions contain an implementation inconsistency that could cause future system instability or developer confusion when the codebase evolves, potentially leading to integration failures and unexpected behaviour changes.

The `toBytes()` function encodes `key.value`, not the entire struct:

KeyEcdsaSecp256k1.toBytes()

```
/**
 * @notice Converts a key to bytes.
 * @param key The key.
 * @return keyBytes The bytes representation of the key.
 */
function toBytes(
    KEY_ECDSA_SECP256K1 memory key
) internal view returns (bytes memory keyBytes) {
    keyBytes = abi.encode(key.value); //@audit Should encode entire struct: abi.encode(key)
}

/**
 * @notice Converts bytes to a key.
 * @param keyBytes The bytes representation of the key.
 * @return key The key.
 */
function fromBytes(
    bytes memory keyBytes
) internal view returns (KEY_ECDSA_SECP256K1 memory key) {
    key = abi.decode(keyBytes, (KEY_ECDSA_SECP256K1));
    if (keccak256(key.unwrap().wrap().toBytes()) != keccak256(keyBytes)) {
        revert KeyEcdsaSecp256k1_InvalidBytes();
    }
}
```

This makes the function `toBytes()` functionally identical to the serialisation function:

KeyEcdsaSecp256k1.serialize()

```
function serialize(
    KEY_ECDSA_SECP256K1 memory key
) internal view returns (bytes memory keySerialized) {
    keySerialized = abi.encode(key.value);
}
```

However, the intention appears to be to encode and decode the entire `KEY_ECDSA_SECP256K1` struct in `toBytes()`.

Because the struct in question consists of only a single `address` field, this issue has no current effect on the functioning of the code: the `bytes` array output is identical for encoding an `address` and encoding a struct consisting of a single `address`. This is why the relevant tests still pass and the code is not rendered unstable.

There does remain a risk, however, that future updates could render the code unstable if the struct is changed, or that implementors could be confused when using these libraries.

The code snippet above is from the `KeyEcdsaSecp256k1` library, however there is an almost identical issue in the `KeyBlsBn254` library.

Recommendations

In both libraries, modify `toBytes()` to encode the entire `key` struct:

KeyEcdsaSecp256k1.toBytes() (fixed)

```
function toBytes(
    KEY_ECDSA_SECP256K1 memory key
) internal view returns (bytes memory keyBytes) {
    keyBytes = abi.encode(key);
}
```

Resolution

The `fromBytes()` functions were modified to assign to `key.value` at commit 55d0cd7.

SYM-06	No Mechanism To Reduce Operator Jail Duration
Asset	`modules/voting-power/extensions/OperatorsJail.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `jailOperator()` function only allows extending jail periods but provides no mechanism to reduce an operator's jail duration, limiting administrative flexibility for managing operator punishments.

The function contains the logic `if (jailedUntil > getOperatorJailedUntil(operator))` which only updates the jail timestamp if the new duration would extend the current jail period.

OperatorsJail.jailOperator()

```
function jailOperator(address operator, uint48 jailedUntil) external {
    // ... permission checks ...

    if (jailedUntil > getOperatorJailedUntil(operator)) { //@audit Can only extend, not reduce jail time
        _jailedUntil[operator] = jailedUntil;
    }
}
```

a long duration, administrators cannot reduce that duration - they can only extend it further or completely unjail the operator using `unjailOperator()`.

This design creates an inflexible system where administrative errors or changing circumstances cannot be easily addressed. If administrators did wish to reduce a jail time, it would require unjailing the operator and then jailing them again.

Recommendations

Consider adding functionality to allow reducing jail duration when appropriate.

Resolution

The issue was acknowledged by the development team.

SYM-07	Token Weight Functions Return Default Values For Unregistered Tokens
Asset	`modules/voting-power/common/voting-power-calc/WeightedTokensVPCalc.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `getTokenWeightAt()` and `getTokenWeight()` functions return the default weight of `1e12` for any address, including unregistered tokens, scam tokens, or non-token addresses, without validating if the token is registered in the voting power provider.

When these functions are called with arbitrary addresses that have no weight explicitly set, they return `DEFAULT_TOKEN_WEIGHT` (`1e12`) instead of zero.

WeightedTokensVPCalc.getTokenWeight()

```
function getTokenWeight(address token) external view returns (uint256) {
    uint256 weight = _tokenWeights[token];
    return weight == 0 ? DEFAULT_TOKEN_WEIGHT : weight; //@audit Returns 1e12 for any unregistered token
}
```

called directly, as callers might assume that any address returning a non-zero weight is a valid, registered token. While the underlying `stakeToVotingPower*()` functions in the voting power provider do filter by registered tokens, direct calls to these weight functions bypass that validation.

This creates a potential for confusion or misuse, especially if external contracts or interfaces rely on these functions to determine token validity or importance. The behavior suggests that all tokens have equal weighting by default, even if they are not part of the system.

While this is mitigated by the voting power provider's own token filtering, the functions should either validate token registration or clearly document the expected usage pattern.

Recommendations

Validate token registration in the functions `getTokenWeightAt()` and `getTokenWeight()`. Alternatively, if this is undesirable, clearly document the behaviour to prevent implementors and other projects from relying on the values these functions return without validating the token.

Resolution

A comment was added to warn of the relevant behaviour at commit eeb47d3.

SYM-08	Unsafe Internal `_setEpochDuration()` Function Allows Data Corruption
Asset	`modules/valset-driver/EpochManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The internal `_setEpochDuration()` function lacks input validation, allowing implementors to corrupt epoch data and break contract functionality through various misuse scenarios.

Since the function is marked `internal`, inheriting contracts or implementors can call it directly with arbitrary parameters, bypassing the safer external interface. This creates several potential vulnerabilities:

1. The current epoch data could be corrupted. It would be possible to insert multiple ascending `epochDurationTimestamp` entries for the current `epochDurationIndex`, or vice versa.
2. Excessively high epoch indices can be set, causing many epoch numbers to be skipped and breaking sequential epoch progression;
3. Multiple future entries can be added, which breaks `getNextEpochDuration()` that relies on `_epochDurationDataByTimestamp.latest()` and also breaks `_getCurrentValue()`, which assumes at most one future entry.

   The `getNextEpochDuration()` function specifically expects only the current and next epoch durations to exist, but multiple future entries would cause it to return incorrect values. Similarly, `_getCurrentValue()` contains logic that assumes only one future checkpoint exists. With multiple future entries, this logic would fail and return incorrect epoch duration data, affecting the output of all functions that call `_getCurrentValue()`.

Recommendations

Add comprehensive validation to the internal `_setEpochDuration()` function or document the restrictions. Consider the following:

1. Prevent changing the entry for the current timestamp and epoch
2. Prevent skipping epochs
3. Prevent multiple future entries

Resolution

The issue was acknowledged by the development team.

SYM-09	`invert()` Function Reverts When Value Is Zero
Asset	`libraries/utils/Scaler.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `invert()` function performs a division by `value` without validating whether value is zero. This causes a runtime error if `value` is zero.

Scaler.invert()

```
function invert(uint256 value) internal pure returns (uint256) {
    return BASE / value; //@audit Division by zero will revert - no validation
}
```

This behaviour is appropriate in its current usage within the `ChainlinkPriceFeed` library, where reverting on invalid input is expected. However, in other potential use cases, it may be more appropriate to return 0 instead of reverting.

Recommendations

Consider refactoring the `invert()` function to support optional zero-handling behavior by introducing a bool `revertOnZero` parameter. This allows the function to either:

- Revert on `value == 0` with an appropriate revert message.
- Return 0 when a non-reverting response is acceptable or preferred in the calling context.

Alternatively, consider reverting with a named error to clarify that a price of zero was inverted.

Resolution

The issue was acknowledged by the development team and a notifying comment added at commit c1a5b0f.

SYM-10	Conflicting `BASE_DECIMALS` Constants
Asset	`ChainlinkPriceFeed.sol, NormalizedTokenDecimalsVPCalc.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Two different contracts define `BASE_DECIMALS` constants with conflicting values, creating potential confusion and calculation errors when these components interact within the voting power calculation system.

The `ChainlinkPriceFeed` library defines `BASE_DECIMALS = 18` for normalising Chainlink price data, while `NormalizedTokenDecimalsVPCalc` defines `BASE_DECIMALS = 24` for token decimal normalisation.

BASE_DECIMALS

```
// In ChainlinkPriceFeed.sol
uint256 internal constant BASE_DECIMALS = 18; //@audit Conflicts with NormalizedTokenDecimalsVPCalc

// In NormalizedTokenDecimalsVPCalc.sol
uint256 internal constant BASE_DECIMALS = 24; //@audit Different value for same constant name
```

Both constants serve similar normalisation purposes but use different decimal precision standards, which could lead to incorrect calculations when price feeds and token calculations are combined.

Recommendations

Resolve the naming conflict and clarify the purpose of each constant.

Resolution

The value of `BASE_DECIMALS` was set to 18 in `NormalizedTokenDecimalsVPCalc` at commit a096c7e.

SYM-11	Missing Public Function To Update Slashing Data
Asset	`modules/voting-power/VotingPowerProvider.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_setSlashingData()` function is internal with no corresponding public function, preventing updates to slashing configuration parameters after contract deployment unless implementors explicitly add one.

The `_setSlashingData(bool requireSlasher, uint48 minVaultEpochDuration)` function allows updating critical slashing configuration parameters that affect vault validation logic.

VotingPowerProvider._setSlashingData()

```
function _setSlashingData(
    bool requireSlasher,
    uint48 minVaultEpochDuration
) internal { //@audit Internal function with no public wrapper
    // ... validation and storage logic
}
```

However, this function is marked as `internal` and there is no public function in the base contract that calls it. This means that unless implementing contracts explicitly create a public wrapper function, the slashing data cannot be updated after the initial deployment.

This creates a significant limitation for contract governance, as the `minVaultEpochDuration` and `requireSlasher` parameters are important configuration settings that may need adjustment based on changing network conditions, security requirements, or governance decisions. The inability to update these parameters could force unnecessary contract upgrades or deployments when only configuration changes are needed.

The issue affects the flexibility and maintainability of the voting power provider system, particularly in dynamic environments where slashing requirements may evolve over time.

Recommendations

Add a public function with proper permission controls to update the slashing data. Alternatively, make clear in the comments and documentation that such a function will be required if implementors wish to update the settings in future.

Resolution

The development team stated that this is intended in the design.

SYM-12	`getOperatorJailedUntil` Returns Past Timestamps For Non-Jailed Operators
Asset	`modules/voting-power/extensions/OperatorsJail.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `getOperatorJailedUntil()` function returns historical jail timestamps even for operators who are no longer jailed, potentially confusing the operator's current jail status.

The function directly returns `_jailedUntil[operator]` without checking if the timestamp is in the future.

OperatorsJail.getOperatorJailedUntil()

```
function getOperatorJailedUntil(
    address operator
) external view returns (uint48) {
    return _jailedUntil[operator]; //@audit Returns past timestamps for previously jailed operators
}
```

This means that, for operators who were previously jailed but whose jail period has expired, the function returns a past timestamp rather than indicating that the operator is currently not jailed. Callers might misinterpret this return value, especially if they do not also check `isOperatorJailed()` to determine the current status.

Recommendations

Consider modifying the function to only return active jail timestamps.

Resolution

The development team stated that this is intended in the design.

SYM-13	Potential Integer Underflow In `nonSignersLength` Calculation
Asset	`modules/settlement/sig-verifiers/SigVerifierBlsBn254Simple.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The calculation of `nonSignersLength` in the `verifyQuorumSig()` function could potentially underflow if `proof.length` is less than `nonSignersOffset`, though this does not appear to lead to an exploitable vulnerability.

SigVerifierBlsBn254Simple.verifyQuorumSig()

```
uint256 nonSignersLength = (proof.length - nonSignersOffset) >> 1; //@audit Potential underflow if proof.length < nonSignersOffset

if (proof.length != nonSignersOffset + nonSignersLength * 2) {
    revert SigVerifierBlsBn254Simple_InvalidProofLength();
}
```

In this proof validation logic, `nonSignersLength` is calculated without first checking that `proof.length >= nonSignersOffset`. If an attacker provides a proof where `proof.length < nonSignersOffset`, this would cause an integer underflow, resulting in an extremely large `nonSignersLength` value. The subsequent reverse validation would still pass due to the arithmetic wraparound, but the function would eventually revert when attempting to process the extremely long list of non-signers in the following loop.

Despite this revert, it is nevertheless best practice to verify that the underflow is not occurring.

Recommendations

Add a bounds check before calculating `nonSignersLength`.

Resolution

The development team pointed out that the underflow would not occur because of the `proof[192:nonSignersOffset]` slice:

SigVerifierBlsBn254Simple.verifyQuorumSig()

```
if (
    keccak256(proof[192:nonSignersOffset])
        != ISettlement(settlement).getExtraDataAt(
            epoch, VERIFICATION_TYPE.getKey(keyTag, VALIDATOR_SET_HASH_KECCAK256_HASH)
        )
) {
    return false;
}
```

SYM-14	Missing Constructors With `_disableInitializers()` In Upgradeable Contracts
Asset	`modules/*`
Status	**Closed:** See Resolution
Rating	Informational

Description

Multiple contracts throughout the codebase inherit from `Initializable` but lack constructors that call `_disableInitializers()`, creating potential security vulnerabilities in proxy implementations.

The affected contracts include:

Base Module Contracts:

- `NetworkManager` - inherits from `Initializable`
- `OzEIP712` - inherits from `EIP712Upgradeable` (which extends `Initializable`)
- `PermissionManager` - inherits from `Initializable`

Common/Permissions Module:

- `OzAccessControl` - inherits from `PermissionManager` and `AccessControlUpgradeable`
- `OzAccessManaged` - inherits from `PermissionManager` and `AccessManagedUpgradeable`
- `OzOwnable` - inherits from `PermissionManager` and `OwnableUpgradeable`

Key Registry Module:

- `KeyRegistry` - inherits from `MulticallUpgradeable` and `OzEIP712`

Network Module:

- `Network` - inherits from `TimelockControllerUpgradeable`

Settlement Module:

- `Settlement` - inherits from multiple contracts including `NetworkManager` and `OzEIP712`

Valset Driver Module:

- `EpochManager` - inherits from `Initializable`
- `ValSetDriver` - inherits from `EpochManager` and `NetworkManager`

Voting Power Base Module:

- `VotingPowerCalcManager` - inherits from contracts that extend `Initializable`

Voting Power Common Calculations Module:

- `EqualStakeVPCalc` - inherits from contracts that extend `Initializable`
- `NormalizedTokenDecimalsVPCalc` - inherits from contracts that extend `Initializable`
- `PricedTokensChainlinkVPCalc` - inherits from contracts that extend `Initializable`
- `WeightedTokensVPCalc` - inherits from contracts that extend `Initializable`
- `WeightedVaultsVPCalc` - inherits from contracts that extend `Initializable`

Voting Power Module:

- `VotingPowerProvider` - inherits from `NetworkManager` and other `Initializable` contracts

Voting Power Extensions Module:

- `BaseRewards` - inherits from `VotingPowerProvider`
- `BaseSlashing` - inherits from `VotingPowerProvider`
- `MultiToken` - inherits from `VotingPowerProvider`
- `OperatorsBlacklist` - inherits from `VotingPowerProvider`
- `OperatorsJail` - inherits from `VotingPowerProvider`
- `OperatorsWhitelist` - inherits from `VotingPowerProvider`
- `OperatorVaults` - inherits from `VotingPowerProvider`
- `OpNetVaultAutoDeploy` - inherits from `VotingPowerProvider`
- `SharedVaults` - inherits from `VotingPowerProvider`

When using upgradeable contracts with OpenZeppelin's proxy pattern, the implementation contract should have a constructor that calls `_disableInitializers()` to prevent the implementation contract itself from being initialized. Without this protection, an attacker could potentially initialize the implementation contract directly.

Recommendations

Add constructors with `_disableInitializers()` to all affected contracts.

Resolution

The development team stated that this is intended in the design.

SYM-15	Inappropriate Visibility On `__Network_init()`
Asset	`modules/network/Network.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `__Network_init()` is currently declared as `public`. While the function is protected by the `onlyInitializing` modifier, this visibility level goes against common upgradeable contract design conventions and introduces unnecessary external exposure.

Network.__Network_init()

```
function __Network_init(
    INetworkManager networkManager
) public onlyInitializing { //@audit Should be 'internal' not 'public'
    __NetworkBase_init(networkManager);
}
```

According to OpenZeppelin's upgradeable contract patterns, initializer functions should typically have `internal` visibility unless they need to be called directly by external deployers, which is not the case here as this appears to be called from other initializer functions.

Recommendations

Consider updating the visibility from `public` to `internal`.

Resolution

The visibility was updated to `internal` at commit 7394f85.

SYM-16	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Return Variable Not Used As A Return Variable**

   Related Asset(s): libraries/utils/ValSetVerifier.sol

   Many of the functions in this file use the following structure:

   ```
   function verifyOperator(
       SszProof calldata validatorRootProof,
       uint256 validatorRootLocalIndex,
       bytes32 validatorSetRoot,
       SszProof calldata operatorProof
   ) internal view returns (bool isValid) {
       isValid = verifyValidatorRootLocal(validatorRootProof, validatorRootLocalIndex, validatorSetRoot);
       if (!isValid) {
           return false;
       }
       return verifyValidatorOperatorLocal(operatorProof, validatorRootProof.leaf);
   }
   ```

   Note that the function declares a return variable `isValid`. That variable, however, is never returned. It is instead used as a working variable to determine which boolean literal to return.

   Either assign to `isValid` throughout the function, or remove the variable name from the return block of the function declaration. Consider removing the variable entirely:

   ```
   if (!verifyValidatorRootLocal(validatorRootProof, validatorRootLocalIndex, validatorSetRoot)) {
       return false;
   }
   ```

2. **Events Emitted Without Status Changes In OperatorsWhitelist**

   Related Asset(s): modules/voting-power/extensions/OperatorsWhitelist.sol

   Multiple functions in the `OperatorsWhitelist` contract emit events even when no actual state changes occur, creating misleading event logs and reducing the reliability of the event system for tracking state transitions.

   The issue affects three functions.

   (a) `whitelistOperator()` unconditionally sets `_whitelisted[operator]` to `true` and emits the `WhitelistOperator` event even if the operator was already whitelisted;

   (b) `unwhitelistOperator()` unconditionally sets `_whitelisted[operator]` to `false` and emits the `UnwhitelistOperator` event even if the operator was already not whitelisted;

   (c) `_setWhitelistStatus()` unconditionally sets `_isWhitelistEnabled` to `status` and emits the `SetWhitelistStatus` event even if the whitelist status was already set to that value.

   Implement state change validation before emitting events.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows.

1. **Return Variable Not Used As A Return Variable**
   The development team acknowledged this issue, noting that fixing it seems to increase bytecode size.
2. **Events Emitted Without Status Changes In OperatorsWhitelist**
   The relevant checks were added, as well as some similar checks relating to jailing operators, at commits 32bea5e and 766cfde.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

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

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].