Prepared for
Tyler Tarsi
Omni Network

Prepared by
Jinseo Kim
Vlad Toie
Zellic

March 29, 2024

Omni AVS

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Omni Network from March 18th to March 29th, 2024. During this engagement, Zellic reviewed Omni AVS's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How are the states synchronized between the main chain and the Omni side chain?
- What checks are in place to ensure that the Operators are not malicious?
- How is it determined that the Operators have staked enough ETH to be considered valid?
- Does the protocol follow all the EigenLayer security standards? Are all the assumptions valid?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

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

During this assessment, the currently limited integration of the reviewed code with the EigenLayer protocol, as well as the reliance on off-chain components for further cross-chain validation, was a limitation in the overall threat modeling.

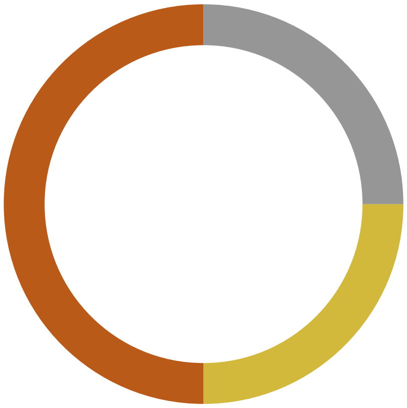
1.4. Results

During our assessment on the scoped Omni AVS contracts, we discovered four findings. No critical issues were found. Two findings were of high impact, one was of medium impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Omni Network's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	2
🟨 Medium	1
🟩 Low	0
⬜ Informational	1



2. Introduction

2.1. About Omni AVS

Omni Network contributed the following description of Omni AVS:

> Omni is an L1 blockchain secured by restaked $ETH. It secures cross-rollup messages, along with an EVM execution environment. For the purposes of this first audit, we won't be covering the Omni chain, just the restaking component of the protocol.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Omni AVS Contracts

Repository	https://github.com/omni-network/omni/ ↗
Version	omni: `6c5139354de2d3be65bb8db94a0460afdbcfe3e1`
Programs	• Omni • OmniAVS
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jinseo Kim
Engineer
jinseo@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 18, 2024	Kick-off call
March 18, 2024	Start of primary review period
March 29, 2024	End of primary review period

3. Detailed Findings

3.1. Share for unsupported strategy of operator triggers infinite loop in _getSelfDelegations

Target	OmniAVS		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

OmniAVS calculates the personal stake amount and delegated stake amount of an operator separately. To calculate an operator's personal stake amount, OmniAVS fetches the delegatable shares of an operator, which is fully delegated to the operator by the design of EigenLayer. Following is the code of the function _getSelfDelegations for this behavior:

```
function _getSelfDelegations(address operator) internal view returns (uint96)
    {
    (IStrategy[] memory strategies, uint256[] memory shares)
    = _delegationManager.getDelegatableShares(operator);

    uint96 staked;
    for (uint256 i = 0; i < strategies.length;) {
        IStrategy strat = strategies[i];

        // find the strategy params for the strategy
        StrategyParam memory params;
        for (uint256 j = 0; j < _strategyParams.length;) {
            if (address(_strategyParams[j].strategy) == address(strat)) {
                params = _strategyParams[j];
                break;
            }
            unchecked {
                j++;
            }
        }

        // if strategy is not found, do not consider it in stake
        if (address(params.strategy) == address(0)) continue;

        staked += _weight(shares[i], params.multiplier);
        unchecked {
            i++;
        }
    }

    return staked;
}
```

The function is implemented using a nested for loop; the outer loop iterates over the strategies of the operator, and the inner loop iterates over the strategy/strategies registered in OmniAVS.

If a strategy of an operator is not registered in OmniAVS, it continues the outer loop. However, because `i` is not incremented, the loop iterates for the same strategy again. This leads to an infinite loop.

Impact

If an operator deposits to the strategy unsupported in OmniAVS, `_getOperators` would run indefinitely instead of returning the result. This makes syncing the list of operators impossible until the operator withdraws from the strategy or is manually ejected.

Recommendations

Consider refactoring the logic to prevent the outer loop from getting stuck in an unsupported strategy.

Remediation

This issue has been acknowledged by Omni Network, and a fix was implemented in commit `fc19c261` ↗.

This finding was brought to our attention by Omni Network prior to the official report being submitted.

3.2. Minimum amount of staking is only checked in the registration

Target	OmniAVS		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

When an operator registers into OmniAVS, it is checked that the operator's stake is greater than the minimum amount of staking.

```
function registerOperator(
    bytes calldata pubkey,
    ISignatureUtils.SignatureWithSaltAndExpiry memory operatorSignature
) external whenNotPaused {
    address operator = msg.sender;

    // ...
    require(_getTotalDelegations(operator) >= minOperatorStake, "OmniAVS: min
    stake not met");
    // ...
}
```

Due to the nature of EigenLayer's architecture, the stake can change throughout the lifetime of a contract and should thus constantly be checked against the minimum threshold. There are multiple reasons for a possible decrease in stake, though the most likely scenario is that a staker (i.e., the normal user that delegates to operators) withdraws their stake from the operator.

Impact

An operator who does not have enough stake would remain in the list of operators, unless they are manually ejected by the owner of OmniAVS.

If the allowlist is disabled and anyone is allowed to register into OmniAVS, a malicious user can fill the maximum number of operators in order to interrupt the registration function without actually staking the minimum amount of staking for each user.

Recommendations

Consider performing the check when generating the list of operators and/or automatically deregistering operators who have stake below the threshold.

Alternatively, prune the list of authorized operators whenever `syncWithOmni` is called, so that the other side of the chain only allows legitimate and healthy operators.

Remediation

This issue has been acknowledged by Omni Network.

Omni Network stated that the minimum threshold to obtain the voting power on the consensus chain will be separately managed and checked. Omni Network also stated that they will enable the allowlist mechanism and work with trusted operators during the first phase, and they plan to implement the pruning operation, which ejects operators below the minimum amount of stake and/or the direct ejection from the consensus chain. Moreover, Omni Network has stated the following:

> In the long term, ejecting operators that are not actively validating (zero voting power), may be necessary. When our AVS operator set is less permissioned (allowlist disabled), it will be important to address scenario you've outlined - in which "malicious user can fill the maximum number of operators in order to interrupt the registration function without actually staking the minimum amount of staking for each user." Though we are hopeful EigenLayer's slashing design will restrict the ease with which users can undelegate to operators.
>
> Additionally, depending on EigenLayer's reward mechanism, it may be important to quickly prune operators that are no longer validating, so that rewards are not paid to inactive operators.
>
> For both cases, we can introduce consensus chain controlled ejections. Our consensus chain already relays validator set updates to portal contracts. Extending this mechanism to remove operators without voting power from our AVS contract is feasible.

3.3. States are not automatically synced

Target	OmniAVS		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The OmniAVS contract forwards the current validators from one side of the chain to the other via the OmniPortal. Currently, the only way of performing the sync between both sides of the chain is to call the `syncWithOmni` function manually, by essentially querying the current total stake for all the currently registered validators.

```
/**
* @notice Sync OmniAVS operator stake & delegations with Omni chain.
*/
function syncWithOmni() external payable whenNotPaused {
    Operator[] memory ops = _getOperators();
    omni.xcall{ value: msg.value }(
        omniChainId,
        ethStakeInbox,
        abi.encodeWithSelector(IEthStakeInbox.sync.selector, ops),
        _xcallGasLimitFor(ops.length)
    );
}
```

Impact

A potential issue may arise here if an operator has deregistered from one side of the chain, and no calls to `syncWithOmni` would occur in the meanwhile. In this case, the operator would still be considered valid on the other side of the chain, even though they have completely removed their stake.

Recommendations

We recommend syncing with the Omni chain on every operation that has to do with the update of the operator's stake or status (i.e., registration/deregistration).

Remediation

This issue has been acknowledged by Omni Network.

Omni Network stated that they will take one of the following actions:

- move the `deregisterOperator` function out of the AVS contract, to a separate contract on Omni's EVM.
- keep the `deregsiterOperator` function in the AVS contract, but transform it into a two-step process. In this case, Omni's EVM contract must acknowledge the de-registration before the actual removal from the AVS contract.

For the current release, Omni Network has removed the `deregisterOperator` function from the AVS contract, as per commit b4e82eb ↗. Therefore, the only way for an operator to deregister is to contact the Omni team, who will then perform the deregistration via `ejectOperator`.

Additionally, Omni Network stated that:

> If users can undelegate from operators freely, without being subject to any delay, we must consider this when designing how to incorporate user delegations into validator set updates. Granting voting power 1:1 with user delegations would not be appropriate. If operators can deregister / withdrawal at any time, and immediately remove their capital from risk of slashing, our AVS design is incompatible with EigenLayer. We do not believe this will be the case, as their current communications / documentation / source code suggest operator withdrawals / de-registrations will be subject to some delay, to account for slashing. We hope that user delegations are subject to this same delay.

3.4. Precision loss in weight-calculating logic with low multipliers

Target	OmniAVS		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

A staker deposits to a strategy to receive shares for the strategy and delegate their shares to an operator. OmniAVS calculates the weight of the registered operator by multiplying the shares and the predetermined multiplier:

```
uint256 internal constant STRATEGY_WEIGHTING_DIVISOR = 1e18;

function _getTotalDelegations(address operator) internal view returns (uint96)
    {
    // ...

    for (uint256 i = 0; i < _strategyParams.length;) {
        // ...
        total += _weight(shares, params.multiplier);
        // ...
    }

    return total;
}

function _weight(uint256 shares, uint96 multiplier)
    internal pure returns (uint96) {
    return uint96(shares * multiplier / STRATEGY_WEIGHTING_DIVISOR);
}
```

Note that the multiplication of `shares` and `multiplier` is divided by `STRATEGY_WEIGHTING_DIVISOR`, which is 10^18. If the multiplier is too low, such as one, a significant part of precision would be lost.

Impact

If the multiplier is too low, such as one, the part of shares will be dismissed in the weight-calculation logic. For example, if 1.9 stETH is staked, the shares for 0.9 stETH will not be considered for calculating weight.

Recommendations

Consider refactoring the `_weight` function to minimize precision loss. One approach would be to perform the division operation first and then the multiplication. This approach should only be used, however, on the assumption that the shares value is larger than `STRATEGY_WEIGHTING_DIVISOR`. Alternatively, another option would be ensuring that only large multipliers are used (i.e., larger than `STRATEGY_WEIGHTING_DIVISOR`).

Remediation

This issue has been acknowledged by Omni Network. Omni Network stated that they will use large multipliers, such as 10^18, for their deployments.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. EigenLayer is being actively developed

The core structural design of EigenLayer is not yet finalized and could undergo changes. Despite our efforts to assess any potential conflicts between Omni AVS and both the current and potential future designs of EigenLayer, our ability to conduct a comprehensive integration assessment was limited due to the lack of detailed information about EigenLayer's future design.

Omni Network stated they plan to deploy the protocol in two phases — they will deploy the restaking smart contracts without implementing slashing and reward mechanisms in the first phase, which is the scope of this audit. The second phase will be deployed after the design of EigenLayer is finalized and implemented.

One should note that Slasher, the key component of EigenLayer, is under active development and its design decisions may affect the functionality of Omni AVS. For instance, during our audit, we found that the current design of Slasher, which is deployed on mainnet, is not compatible with the Omni AVS because an operator must opt in to slashing and the AVS should confirm this before accepting their registration. We communicated this information to Omni Network, and we were informed that EigenLayer stated that the future design of Slasher will not involve an opt-in process, unlike the current Slasher contract on mainnet.

Additionally, since the slashing feature is not implemented yet, Omni AVS should enable the allowlist mechanism and collaborate with trusted operators during the initial phase. Omni Network confirmed their intention to do so.

Overall, it is our opinion that Omni Network should actively communicate with the EigenLayer community in order to ensure that the future design changes of EigenLayer do not break the functionality of Omni AVS.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: OmniAVS.sol

Function: `deRegisterOperator()`

Should allow deregistration of operators from the AVS.

Branches and code coverage (including function calls)

Intended branches

- Assumes the operator has not been restricted from deregistering after being caught as malicious. Currently not enforced as there is no implementation of the Slasher.
  - ☐ Test coverage
- Should remove the operator from the `operators` array as well as remove its pubkey from the `_operatorPubkeys` mapping.
  - ☑ Test coverage

Negative behavior

- Should not be callable if the contract is paused. Enforced through the `whenNotPaused` modifier.
  - ☑ Test coverage
- Should not be callable if `msg.sender` is not actually an operator. Enforced through the `require` statement.
  - ☑ Test coverage

Function: `initialize(address owner_, IOmniPortal omni_, uint64 omniChainId_, address ethStakeInbox_, uint96 minOperatorStake_, uint32 maxOperatorCount_, StrategyParam[] calldata strategyParams_)`

Initializes the OmniAVS contract.

Inputs

- `owner_`
  - **Validation**: None.

    - **Impact**: The address of the contract owner.
- `omni_`
    - **Validation**: None.
    - **Impact**: The Omni portal contract.
- `omniChainId_`
    - **Validation**: None.
    - **Impact**: The Omni chain ID.
- `ethStakeInbox_`
    - **Validation**: None.
    - **Impact**: The EthStakeInbox contract address.
- `minOperatorStake_`
    - **Validation**: None.
    - **Impact**: The minimum operator stake.
- `maxOperatorCount_`
    - **Validation**: None.
    - **Impact**: The maximum operator count.
- `strategyParams_`
    - **Validation**: None.
    - **Impact**: List of accepted strategies and their multipliers.

Branches and code coverage (including function calls)

Intended branches

- Set the OmniPortal contract.
  ☑ Test coverage
- Set the `OmniChainId`.
  ☑ Test coverage
- Set the `XCallGasLimits`.
  ☑ Test coverage
- Set the `EthStakeInbox`.
  ☑ Test coverage
- Set the `MinOperatorStake`.
  ☑ Test coverage
- Set the `MaxOperatorCount`.
  ☑ Test coverage
- Set the `StrategyParams`.
  ☑ Test coverage
- Enable the allowlist.
  ☑ Test coverage
- Transfer ownership to the `owner_`.
  ☑ Test coverage

Negative behavior

- Should not be callable multiple times. Enforced through the `initializer` modifier.
  - ☑ Test coverage

Function: `registerOperator(bytes calldata pubkey, ISignatureUtils.SignatureWithSaltAndExpiry memory operatorSignature)`

Allows the registration of operators within the AVS.

Inputs

- `pubkey`
  - **Validation**: Validated that it matches the msg.sender pubkey.
  - **Impact**: The public key of the operator.
- `operatorSignature`
  - **Validation**: Assumed to be validated within the `_avsDirectory`
  - **Impact**: The signature that would match the registration parameters.

Branches and code coverage (including function calls)

Intended branches

- Should forward the opt-in call to the Slasher. Currently not implemented, as EigenLayer's implementation of Slasher has not yet been finalized.
  - ☐ Test coverage
- Should add the operator to the AVS. Handled in `_addOperator`.
  - ☑ Test coverage
- Assumes that the signature is properly verified within the AVSDirectory.
  - ☑ Test coverage
- Should forward the registration call to the AVSDirectory. Handled in `_avsDirectory.registerOperatorToAVS`.
  - ☑ Test coverage

Negative behavior

- Should not be callable when paused. Enforced through the `whenNotPaused` modifier.
  - ☑ Test coverage
- Should not allow an operator that does not match the pubkey.
  - ☑ Test coverage
- Should not allow an operator that is not in the allowlist, should the allowlist be enabled.
  - ☑ Test coverage
- Should not allow an operator that is already an operator.
  - ☑ Test coverage

- Should not allow an operator if the maximum operator count has been reached.
  - ☑ Test coverage
- Should not allow an operator if the minimum stake has not been met.
  - ☑ Test coverage

Function: `syncWithOmni()`

Allows syncing of operators' stake and delegations with the Omni chain.

Branches and code coverage (including function calls)

Intended branches

- Retrieves the current list of operators.
  - ☑ Test coverage
- Forwards the call to OmniPortal via `xcall` to sync the operators' stake and delegations with the Omni chain.
  - ☑ Test coverage
- Assumes that `msg.value` is enough to cover the fee required for the sync.
  - ☑ Test coverage

Negative behavior

- Should not allow forwarding of the call if any of the operators are below the threshold. Currently not enforced.
  - □ Test coverage
- Should not be callable if the contract is paused. Enforced through the `whenNotPaused` modifier.
  - ☑ Test coverage

Function: `_setEthStakeInbox(address inbox)`

Allows setting the EthStakeInbox contract address for the AVS.

Inputs

- `inbox`
  - **Validation**: None.
  - **Impact**: The EthStakeInbox contract address.

Branches and code coverage (including function calls)

Intended branches

- Set the `ethStakeInbox`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone (ensured through the `onlyOwner` modifier in the function definition).
  - ☑ Test coverage

Function: `_setMaxOperatorCount(uint32 count)`

Allows setting the maximum operator count for the AVS.

Inputs

- `count`
  - **Validation**: None.
  - **Impact**: Sets the maximum operator count for the AVS.

Branches and code coverage (including function calls)

Intended branches

- Set the maximum operator count for the AVS.
  - ☑ Test coverage

Negative behavior

- Should not allow setting the max operator to a value smaller than the current number of operators. Currently not enforced.
  - □ Test coverage
- Should not be callable by anyone (ensured through the `onlyOwner` modifier in the function definition).
  - ☑ Test coverage

Function: `_setMinOperatorStake(uint96 stake)`

Sets the minimum operator stake for the AVS.

Inputs

- `stake`
  - **Validation**: None.
  - **Impact**: The minimum operator stake for the AVS.

Branches and code coverage (including function calls)

Intended branches

- Set the minimum operator stake for the AVS.
  - ☑ Test coverage

Negative behavior

- Should not allow setting the min operator stake to a value smaller than what the current operators have staked. Currently not enforced.
  - □ Test coverage
- Alternatively, should remove all operators that do not meet the new min operator stake. Currently not enforced.
  - □ Test coverage
- Should not be callable by anyone (ensured through the `onlyOwner` modifier in the function definition).
  - ☑ Test coverage

Function: `_setOmniChainId(uint64 chainId)`

Allows setting the Omni chain ID for the AVS.

Inputs

- `chainId`
  - **Validation**: None.
  - **Impact**: The Omni chain ID.

Branches and code coverage (including function calls)

Intended branches

- Set the `omniChainId`.
  - ☑ Test coverage
- Assumed to not change / very rarely change, as it would imply the AVS is moving to a different chain.
  - ☑ Test coverage

Negative behavior

- Should be different than `block.chainid`. Currently not enforced.
  - □ Test coverage
- Should not be callable by anyone (ensured through the `onlyOwner` modifier in the function definition).
  - ☑ Test coverage

Function: `_setOmniPortal(IOmniPortal portal)`

Allows setting the Omni portal contract for the AVS.

Inputs

- `portal`
  - **Validation**: None.
  - **Impact**: The Omni portal contract.

Branches and code coverage (including function calls)

Intended branches

- Set the `omni` contract.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone (ensured through the `onlyOwner` modifier in the function definition).
  - ☑ Test coverage

Function: `_setStrategyParams(StrategyParam[] calldata params)`

Allows setting the strategy parameters of the AVS.

Inputs

- `params`
  - **Validation**: Checked that each strategy is not zero and that there are no duplicates.
  - **Impact**: Sets the strategy parameters of the AVS.

Branches and code coverage (including function calls)

Intended branches

- Implies that all the operators have stakes in the new strategies. Currently not enforced.
  - □ Test coverage
- Should delete existing strategy parameters.
  - ☑ Test coverage

Negative behavior

- Should not add strategies that the operators do not have stakes in. Currently not enforced.
  - □ Test coverage

Function: `_setXCallGasLimits(uint64 base, uint64 perOperator)`

Allows setting the `xcall` gas limits for the AVS.

Inputs

- `base`
  - **Validation**: None.
  - **Impact**: The base `xcall` gas limit.
- `perOperator`
  - **Validation**: None.
  - **Impact**: The per-operator additional `xcall` gas limit.

Branches and code coverage (including function calls)

Intended branches

- Set the base `xcall` gas limit.
  - ☑ Test coverage
- Set the per-operator additional `xcall` gas limit.
  - ☑ Test coverage
- Assumed these are forwarded/used in the `xcall`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone (ensured through the `onlyOwner` modifier in the function definition).
  - ☑ Test coverage

5.2. Module: Omni.sol

Function: `constructor(uint256 initialSupply, address recipient)`

Defining the Omni ERC-20 token.

Inputs

- `initialSupply`

    - **Control**: Fully controlled by the deployer.
    - **Constraints**: None.
    - **Impact**: The initial supply of the token to be minted.
- `recipient`
    - **Control**: Fully controlled by the deployer.
    - **Constraints**: None.
    - **Impact**: The address that will receive the initial supply of the token.

Branches and code coverage

Intended branches

- Mint the entire initial supply to the `recipient`.
    - ☑ Test coverage
- Properly initialize the ERC20 and ERC20Permit contracts.
    - ☑ Test coverage

Negative behavior

- None.

6. Assessment Results

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

During our assessment on the scoped Omni AVS contracts, we discovered four findings. No critical issues were found. Two findings were of high impact, one was of medium impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.