Aligned Layer

Date	August 2024
Auditors	Martin Ortner, George Kobakhidze

1 Executive Summary

This report presents the results of our engagement with **Aligned** to review **Aligned Layer AVS Smart Contracts**.

The review was conducted over two weeks, from **August 12** to **August 16**. A total of 2x5 person-days were spent.

Aligned Layer AVS is a system that validates Zero-Knowledge and Validity proofs and publishes the results on Ethereum. Its core is implemented as an EigenLayer Actively Validated Service (AVS) that implements EigenLayer's `ServiceManager` interface and verifies operator Stakes with EigenLayer's `BLSSignatureChecker`. Operator registration is whitelisted by implementing non-standard standard changes (!) to EigenLayer's `RegistryCoordinator`. These changes are not in scope for this review. Aligned Layer controls access to operator registration and is responsible for ensuring minimum staked quorum and operator distribution is fair and safe for the purpose of this protocol. For this purpose, Aligned AVS uses hardcoded quorum ID `0` for consensus verification and stake quorum checks with a threshold of `67/100`. Quorum setup is not in scope for this review. Users are advised to review Aligned Layer's on-chain AVS configuration, especially the `operatorSetParams`, `minimumStake`, and `strategyParams` for quorum id `0`, as this defines the security of the protocol.

An additional contract is introduced to batch user signatures together under a single root on their behalf by a centralized batcher service. Similarly, functionality to prove inclusion is also present.

The system uses EigenLayer's Merkle tree library to verify the inclusion of leaves in a root. The AVS is based on a modified version of `ServiceManagerBase` and `BLSSignatureChecker` (`checkSignatures`). The security of the system depends on the correctness of these foreign functions, libraries, and contracts.

Non-standard modifications to EigenLayer Middleware are used. They are tracked with PR-295. As noted in the Findings we strongly recommend subclassing middleware functionality in the Aligned namespace instead of forking upstream to patch-in custom, standard deviating middlelayer changes. Namely, API changes, changes to Function Signatures, and Error Conditions can be problematic and may lead to the privately forked version diverging from upstream up to a point where it becomes unmaintainable to backport security patches.

2 Scope

This review focused on v0.4.0 (325aef8c3f54ec596b4733956a8ac487d5535fc3). The list of files in scope can be found in the Appendix.

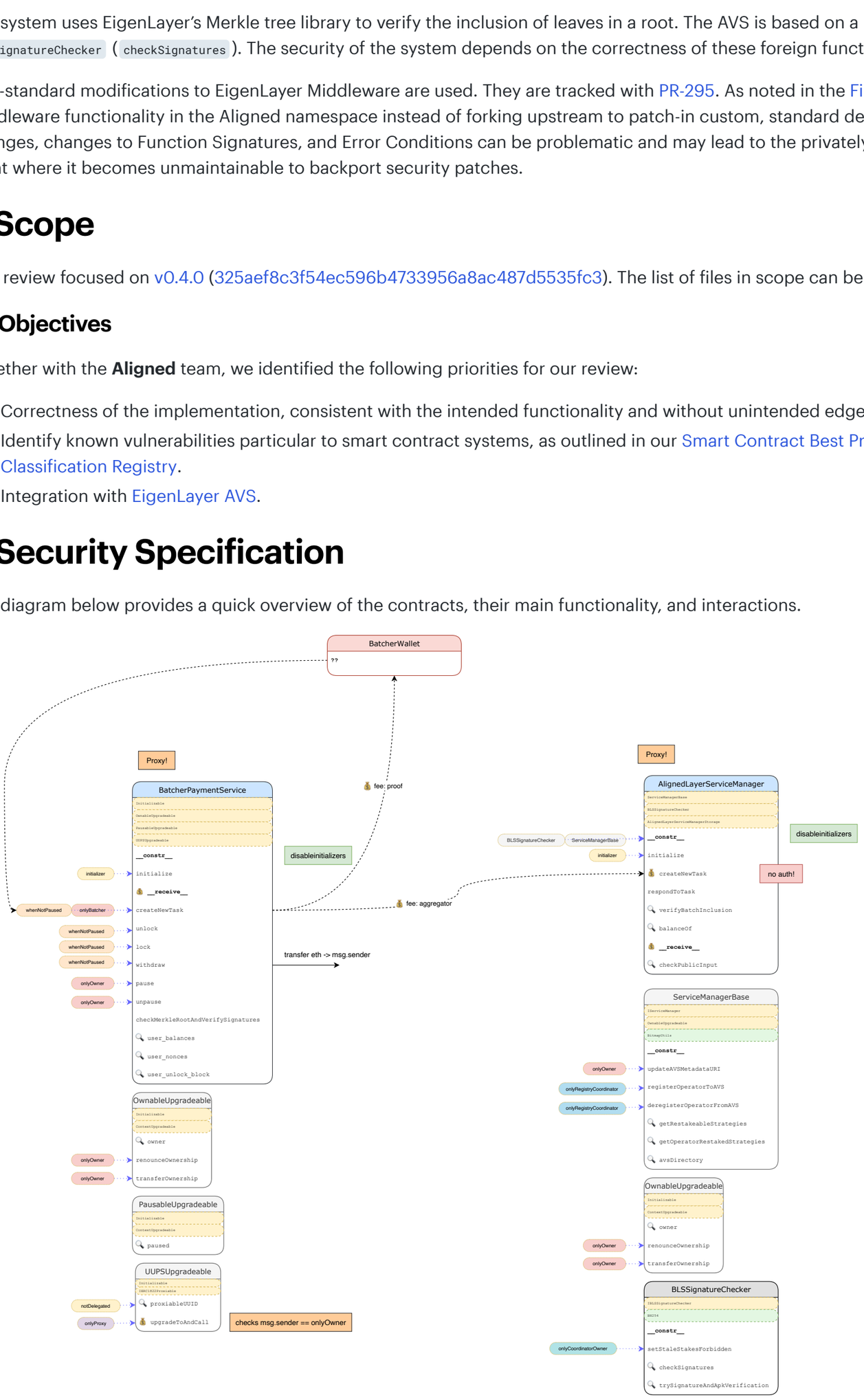
2.1 Objectives

Together with the **Aligned** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Integration with EigenLayer AVS.

3 Security Specification

The diagram below provides a quick overview of the contracts, their main functionality, and interactions.



Overview: Aligned Contracts

This section describes, **from a security perspective**, the behavior of the system under audit. It is not a substitute for documentation.

3.1 Actors

The relevant actors are listed below with their respective abilities:

- **The Aligned team**: The Aligned team has privileged access to upgrade upgradeable contracts in the system and also has owner access to the contracts by default. They can pause the batcher contract and are trusted not to be compromised or malicious. They must ensure that owner parameters are not changed

in a way that negatively impacts users and must not abuse the upgradeability of the batcher contract. It should be noted that the owners might upgrade the contracts at will, changing behavior without prior notifications for users.

- **The batcher**: The centralized batcher is responsible for batching user signatures and their signed leaves together to create tasks in the service manager. They are the only entity that can batch user signatures together to create a task, although users can also create a task independently. The batcher is trusted not to censor user signatures and to maintain a healthy balance on the contracts to ensure continuous execution and the ability to refund user transactions. Users have the option to submit roots directly to avoid censoring by the batcher, however, this functionality may allow for a variety of attacks (see Findings).
- **The aggregator**: The aggregator responds to tasks with aggregated BLS signatures of the voted nodes in the Aligned layer. They are trusted to correctly collect all signatures and submit them timely to the service manager.
- **The users**: Users provide their signatures and signed leaves to the batcher to be included in the proofs. They are required to deposit funds to facilitate batcher gas expenditure, and their withdrawals are time-locked to prevent frontrunning the batch by withdrawing their funds. There is no particular trust provided to users beyond these requirements.

3.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- **The Aligned team**: Trusted to manage upgradeable contracts responsibly and not to misuse their privileged access.
- **The batcher**: Trusted to batch user signatures fairly, maintain contract balances, and ensure continuous operation.
- **The aggregator**: Trusted to collect and submit aggregated BLS signatures accurately and timely.
- **The users**: Required to deposit funds and adhere to time-locked withdrawals, with no additional trust assumptions.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Public Visibility on `checkMerkleRootAndVerifySignatures` Allows for DOS Attacks and User Fund Loss Critical ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#801 by changing the function name from `checkMerkleRootAndVerifySignatures` to `_checkMerkleRootAndVerifySignatures` and visibility from `public` to `private`. The function `verifySignatureAndDecreaseBalance` also had its name changed to `_verifySignatureAndDecreaseBalance` to follow style. The client provided the following statement:

> Visibility was changed to private, and private functions were refactored to start with _ . Contracts were also redeployed in anvil.

Description

When a new task is created through `BatcherPaymentService`, the `createNewTask()` function is called with appropriate signature arguments to validate user submissions, subtract funds from their accounts, and ultimately create the task via a call to `AlignedLayerServiceManager.createNewTask()`.

However, the signature validation and balance adjustment happens in an intermediate function called `checkMerkleRootAndVerifySignatures()` which takes almost the same parameters:

contracts/src/core/BatcherPaymentService.sol:L173-L178

```
function checkMerkleRootAndVerifySignatures(
    bytes32[] calldata leaves,
    bytes32 batchMerkleRoot,
    SignatureData[] calldata signatures,
    uint256 feePerProof
) public {
```

The issue is that this function is `public` with no modifiers, unlike `createNewTask()` which is both `onlyBatcher` and `whenNotPaused`. There are no further checks for `msg.sender`. As a result, any user can call `checkMerkleRootAndVerifySignatures()` directly with correct signature parameters by frontrunning the `createNewTask()` transaction, and the contract would validate the signatures and subtract user balances.

Two additional outcomes here are that:

- The user nonces are now updated, so those same signatures arguments that were taken can no longer be used, so the proper transaction that got frontran will be reverted in the same part of the code where the nonce got updated:

contracts/src/core/BatcherPaymentService.sol:L248-L249

```
require(user_data.nonce == signatureData.nonce, "Invalid Nonce");
user_data.nonce++;
```

- The `feePerProof` argument is given directly in `checkMerkleRootAndVerifySignatures()`, so a malicious user can simply pick a large number to subtract all funds from user balances. Just as a note - in `createNewTask()` the `feePerProof` argument is constructed from arguments by an authenticated called which is the batcher, so, while the resulting `feePerProof` could still be unfairly large, this is at least less possible from an authenticated called which has access to other admin-like functionality on the contract.

The impact of this function's visibility is that users may lose all their funds, and proper transactions can be reverted.

Recommendation

Adjust visibility and other argument checks on the `checkMerkleRootAndVerifySignatures()` function.

4.2 `BatcherPaymentService` - `onlyBatcher` Restriction, `batchMerkleRoot` Verification, `pausable`, and `feePerProof` Can Be Bypassed Completely by Calling `AlignedLayerServiceManager` Directly Critical Partially Addressed

Resolution

Having the ability to bypass the batcher is an intended design, while the checks (such as leaf and signature verification) are addressed by off-chain logic. So, while on-chain contract checks aren't implemented, the Aligned layer nodes perform the necessary review of data, such as verification of submitted `batchMerkleRoot` with associated `batchDataPointer` data. The client provided the following statement:

> The batcher is not part of the core protocol, is just an addition to let people batch proofs with other people. No check that is done in that contract should be needed for the system to work. The checks are on the operator, it recalculates the root from the leaves of the data it downloads and checks it matches with the one on the blockchain, else the batch proof is rejected. Each operator:
>
> - Downloads all the data with the proofs
> - Recalculates the merkle tree from the data (the proofs), and checks that it's root matches the root in Ethereum. Else the batch of proofs is not correct, because the leaves are not associated with the root.
> - Verifies all the proofs
> - Send the signed root if the batch is valid to the Aggregator

Description

Several vulnerabilities are present due to insufficient access controls and a lack of proper authentication checks between `BatcherPaymentService` and `AlignedLayerServiceManager`. These issues allow unauthorized actions and bypass essential validations. Also, see issue 4.3 .

1. Unauthorized Access to createNewTask()

The function `AlignedLayerServiceManager.createNewTask()` is not protected by authentication, allowing any user to call it directly without going through the `BatcherPaymentService` interface. This enables bypassing the `onlyBatcher` restriction applied in `BatcherPaymentService.createNewTask()`.

contracts/src/core/BatcherPaymentService.sol:L68-L76

```
// PUBLIC FUNCTIONS
function createNewTask(
    bytes32 batchMerkleRoot,
    string calldata batchDataPointer,
    bytes32[] calldata leaves, // padded to the next power of 2
    SignatureData[] calldata signatures, // actual length (proof sumbitters == proofs submitted)
    uint256 gasForAggregator,
    uint256 gasPerProof
) external onlyBatcher whenNotPaused {
```

contracts/src/core/AlignedLayerServiceManager.sol:L56-L60

```
function createNewTask(
    bytes32 batchMerkleRoot,
    string calldata batchDataPointer
) external payable {
    require(
```

2. Bypassing Merkle Root and Signatures / Balance checks

Direct interaction with `AlignedLayerServiceManager.createNewTask()` allows users to bypass Merkle root and signature checks enforced by `BatcherPaymentService.createNewTask()`. This omission provides no incentive to use `BatcherPaymentService`, undermining the intended validation mechanisms.

contracts/src/core/BatcherPaymentService.sol:L98-L114

```
checkMerkleRootAndVerifySignatures(
    leaves,
    batchMerkleRoot,
    signatures,
    feePerProof
);

// call alignedLayerServiceManager
// with value to fund the task's response
(bool success, ) = AlignedLayerServiceManager.call{
    value: feeForAggregator
}(
    abi.encodeWithSignature(
        "createNewTask(bytes32,string)",
        batchMerkleRoot,
        batchDataPointer
    )
```

3. Bypassing Pausable Functionality

`BatcherPaymentService` has pausable functionality restricted to `onlyBatcher`, but `AlignedLayerServiceManager` does not enforce such restrictions and is callable by anyone. This exposes the system to risks during maintenance or emergency situations.

contracts/src/core/BatcherPaymentService.sol:L69-L76

```
function createNewTask(
    bytes32 batchMerkleRoot,
    string calldata batchDataPointer,
    bytes32[] calldata leaves, // padded to the next power of 2
    SignatureData[] calldata signatures, // actual length (proof sumbitters == proofs submitted)
    uint256 gasForAggregator,
    uint256 gasPerProof
) external onlyBatcher whenNotPaused {
```

`AlignedLayerServiceManager` has no pausable restrictions, callable by anyone

contracts/src/core/AlignedLayerServiceManager.sol:L56-L68

```
function createNewTask(
    bytes32 batchMerkleRoot,
    string calldata batchDataPointer
) external payable {
    require(
        batchesState[batchMerkleRoot].taskCreatedBlock == 0,
        "Batch was already submitted"
    );

    if (msg.value > 0) {
        batchersBalances[msg.sender] += msg.value;
    }
```

4. Fee Evasion

Users may avoid paying the `feePerProof` by interacting directly with `AlignedLayerServiceManager.createNewTask()` rather than using `BatcherPaymentService`, thus circumventing fees intended for the `BatcherWallet` in `BatcherPaymentService`.

contracts/src/core/BatcherPaymentService.sol:L105-L121

```
// call alignedLayerServiceManager
// with value to fund the task's response
(bool success, ) = AlignedLayerServiceManager.call{
    value: feeForAggregator
}(
    abi.encodeWithSignature(
        "createNewTask(bytes32,string)",
        batchMerkleRoot,
        batchDataPointer
    )
);

require(success, "createNewTask call failed");

payable(BatcherWallet).transfer(
    (feePerProof * signaturesQty) - feeForAggregator
);
```

Recommendation

Implement access control and authentication checks in `AlignedLayerServiceManager` to ensure that all critical functions are only callable through authorized channels, such as `BatcherPaymentService`. Specifically, consider the following actions:

- Restrict direct access to `createNewTask()` in `AlignedLayerServiceManager` and enforce access through `BatcherPaymentService`.
- Ensure that `AlignedLayerServiceManager` incorporates all necessary checks, including Merkle root and signature validations, to maintain system integrity.
- Review fee structures and enforce them consistently across both contracts to prevent fee evasion.

4.3 Frontrunning DoS on `createNewTask` Allows Anyone to Block Legitimate `batchMerkleRoot` Submissions Critical ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#804 by introducing a new ID for tasks called `batchIdentifierHash` that is based on both the task creator `msg.sender` and the `batchMerkleRoot` via `keccak256(abi.encodePacked(batchMerkleRoot, senderAddress))`. This addresses the DOS problem as a frontrunner would create a new and different `batchIdentifierHash,` and the original transaction's sender's task can still be responded to. This does allow previously submitted and responded to tasks to be re-created. Subsequently, their associated Merkle roots may be re-verified, emitting previously seen events, namely `BatchVerified(batchMerkleRoot)`. However, this isn't a problem since it is just a matter of re-verification of roots & signatures. The client provided the following statement:

> AlignedLayerServiceManager's map to store batches state (previously merkle_root -> batchState), now uses msg.sender as part of its key (now hash(merkle_root, msg.sender) -> batchState).

Description

The `AlignedLayerServiceManager.createNewTask()` function is vulnerable to front-running attacks due to a lack of authentication. Together with insufficient balance management this can easily lead to Denial of Service (DoS) attacks. Malicious actors can preempt legitimate batcher submissions by setting themselves as the batcher for a merkle root with minimal funds, potentially causing DoS conditions or enabling extortion. The function only requires a non-zero value, allowing tasks that can't cover transaction costs reimbursed in `respondToTask()`. Additionally, the user-provided batchDataPointer in emitted events may impact off-chain components if manipulated.

- `BatcherPaymentService.createNewTask()` calling `AlignedLayerServiceManager.createNewTask()` (presumably, this is the main call flow)

contracts/src/core/BatcherPaymentService.sol:L109-L115

```
}(
    abi.encodeWithSignature(
        "createNewTask(bytes32,string)",
        batchMerkleRoot,
        batchDataPointer
    )
);
```

- `AlignedLayerServiceManager.createNewTask()` is not authenticated, and batch states are global and merkle root indexed. The function does not enforce a meaningful minimum balance for the batcher; `1 wei` is enough to set yourself as the batcher for a merkle root while this task can never be fulfilled because it is not enough to cover transaction costs reimbursed when responding.

contracts/src/core/AlignedLayerServiceManager.sol:L56-L80

```
function createNewTask(
    bytes32 batchMerkleRoot,
    string calldata batchDataPointer
) external payable {
    require(
        batchesState[batchMerkleRoot].taskCreatedBlock == 0,
        "Batch was already submitted"
    );

    if (msg.value > 0) {
        batchersBalances[msg.sender] += msg.value;
    }

    require(batchersBalances[msg.sender] > 0, "Batcher balance is empty");

    BatchState memory batchState;

    batchState.taskCreatedBlock = uint32(block.number);
    batchState.responded = false;
    batchState.batcherAddress = msg.sender;

    batchesState[batchMerkleRoot] = batchState;

    emit NewBatch(batchMerkleRoot, uint32(block.number), batchDataPointer);
}
```

- `AlignedLayerServiceManager.respondToTask()` reimbursing transaction costs

contracts/src/core/AlignedLayerServiceManager.sol:L135-L148

```
// 70k was measured by trial and error until the aggregator got paid a bit over what it needed
uint256 txCost = (initialGasLeft - finalGasLeft + 70000) * tx.gasprice;

require(
    batchersBalances[batchesState[batchMerkleRoot].batcherAddress] >=
        txCost,
    "Batcher has not sufficient funds for paying this transaction"
);

batchersBalances[
    batchesState[batchMerkleRoot].batcherAddress
] -= txCost;
payable(msg.sender).transfer(txCost);
```

The `batchDataPointer` is user provided data that is emitted with the event. By front-running legitimate task creations a malicious actor could submit the same root with a different `batchDataPointer` which might have consequences for off-chain components picking up that data. Additionally, the solidity type `string` might not be the most efficient data type for this purpose (`string` is assumed utf8 encoded, `bytes` would be more efficient, fixed length bytes would be most gas efficient)

Recommendation

Implement a meaningful minimum balance requirement for createNewTask(). Consider removing transaction cost reimbursement or implement a more robust system. Authenticate the createNewTask() function or index tasks by a unique batchId instead of merkle root. Implement measures to prevent front-running of batcher assignments. Validate and sanitize the batchDataPointer input. Consider using bytes instead of string for batchDataPointer for improved efficiency.

4.4 `tx.gasprice` Allows a Malicious User to Steal All Funds From the Batcher Balance Major ✓ Fixed

Resolution

Addressed in:

- PR 869 by allowing users to specify a maximum fee to be taken for processing their signatures
- PR 910 by limiting the fee that the aggregator can take for processing the batch

Additionally, PR 883 was introduced that further protects the system by placing an authentication check on `respondToTaskV2()` so that only the aggregator may call it to safeguard against frontrunners that would otherwise be able to steal the fee meant for the aggregator.

Description

To calculate the refund for a user, the contracts use `tx.gasprice`. However, this is an effective gas price that the user pays, not a system-set price. In particular, it consists of two gas prices:

- base fee, as set by the system
- priority fee, which is tipped to the miner and defined by the user.

This gas price is used to calculate the transaction cost to refund the user:

contracts/src/core/AlignedLayerServiceManager.sol:L137

```
uint256 txCost = (initialGasLeft - finalGasLeft + 70000) * tx.gasprice;
```

In the event that a user==miner, they can:

1. set an absurdly high transaction price
2. call `respondToTask()`
3. perform the transaction
4. get the inflated transaction refund which would siphon all the funds out from the `batchersBalances[batchesState[batchMerkleRoot].batcherAddress]`

Recommendation

Consider redesigning the refund mechanism to not be susceptible to user-given parameters, such as a flat refund or a base fee-focused refund.

4.5 Batcher Balance May Be Insufficient Major ✓ Fixed

Resolution

Addressed in yetanotherco/aligned_layer#876 by introducing a new function `depositToBatcher(address account)` that allows to increase the balance of a given `account` by `msg.value` sent along with the transaction. This allows to top off the batcher's balance so the task can be responded to. Additionally, PR yetanotherco/aligned_layer#804 also helps alleviate the problem of insufficient batcher balance, as now previously submitted Merkle roots can be re-submitted with another sender with sufficient balance as the associated batcher to create the task.

Description

The `AlignedLayerServiceManager` contract maintains a balance tracker, `batchersBalances[]`, which is updated either through the `receive()` function from a user or during `createTask()`. The latter requires a non-zero value to be sent with the message:

contracts/src/core/AlignedLayerServiceManager.sol:L65-L69

```
if (msg.value > 0) {
    batchersBalances[msg.sender] += msg.value;
}

require(batchersBalances[msg.sender] > 0, "Batcher balance is empty");
```

Once a task is created for a batcher address, it remains associated with that address and cannot be removed. When responding to the task, the system verifies that the batcher has sufficient funds to cover the transaction:

contracts/src/core/AlignedLayerServiceManager.sol:L104-L107

```
require(
    batchersBalances[batchesState[batchMerkleRoot].batcherAddress] > 0,
    "Batcher has no balance"
);
```

contracts/src/core/AlignedLayerServiceManager.sol:L139-L143

```
require(
    batchersBalances[batchesState[batchMerkleRoot].batcherAddress] >=
        txCost,
    "Batcher has not sufficient funds for paying this transaction"
);
```

The initial check ensures a non-zero balance, while the second check requires a specific amount and will revert if the balance is insufficient. This can lead to a situation where a batcher's task becomes unrespondable if their balance is depleted, whether due to malicious intent or an error. Furthermore, the balance tracker is only updated through `receive()` or `createTask()`, and there is no mechanism to top off a batcher's balance without their direct involvement.

Recommendation

Consider

- asking for specific funds during createTask so the rest of the flow goes through
- allowing for admins to remove tasks
- scratching the refund mechanism

4.6 Use EIP-712-style Signed Hashing to Prevent Cross-Chain Signature Replaying Major ✓ Fixed

Resolution

Addressed in stages:

- PR 822 by adding the chainId to the hashed data.
- PR 916, PR 1041, and PR 1054 by modifying the signed struct to be fully EIP-712 compliant.

Description

The current implementation uses a raw `ecrecover` function to verify whether a user signed a data structure by hashing the data directly. This approach is susceptible to cross-network replay attacks. To enhance security and prevent such issues, the implementation should use a domain separator as specified in EIP-712.

contracts/src/core/BatcherPaymentService.sol:L230-L244

```
function verifySignatureAndDecreaseBalance(
    bytes32 hash,
    SignatureData calldata signatureData,
    uint256 feePerProof
) private {
    bytes32 noncedHash = keccak256(
        abi.encodePacked(hash, signatureData.nonce)
    );

    address signer = ecrecover(
        noncedHash,
        signatureData.v,
        signatureData.r,
        signatureData.s
    );
```

Recommendation

Utilize the OpenZeppelin EIP712 and ECDSA libraries to implement domain-bound signatures. These libraries provide robust and battle-tested solutions for signature validation, mitigating common issues associated with ECDSA and preventing cross-network replay attacks.

4.7 `BatcherPaymentService` - Non-Atomic Contract Deployment and Initialization Forge Script Can Be Front Run Medium

✓ Fixed

Resolution

Fixed with yetanotherco/aligned_layer#828 by initializing the Proxy directly with the internal `upgradeToAndCall` during deployment.

Description

The `BatcherPaymentService` deployment script using Forge performs a two-step deployment and initialization process. Between the `startBroadcast` and `stopBroadcast` directives, the script executes three transactions:

1. Deploys a new `BatcherService` contract.
2. Deploys a new `ERC1967Proxy` without initialization calldata.
3. Initializes the `BatcherPaymentService` contract through the proxy.

contracts/script/deploy/BatcherPaymentServiceDeployer.s.sol:L31-L44

```
vm.startBroadcast();

BatcherPaymentService batcherPaymentService = new BatcherPaymentService();
ERC1967Proxy proxy = new ERC1967Proxy(
    address(batcherPaymentService),
    ""
);
BatcherPaymentService(payable(address(proxy))).initialize(
    alignedLayerServiceManager,
    batcherPaymentServiceOwner,
    batcherWallet
);

vm.stopBroadcast();
```

Since these transactions are broadcast individually, there is a risk that a malicious actor could front-run the third transaction to claim control of the newly deployed contract. This issue could force the deployer to re-deploy the contracts.

For further details, refer to the Forge script documentation.

Recommendation

To mitigate this risk, include the initialization calldata when deploying the `ERC1967Proxy`. This approach ensures that the contract is initialized in the same transaction as its deployment, preventing potential front-running attacks.

4.8 No Admin Withdraw Functions on Contracts Medium ✓ Fixed

Resolution

While admin withdraw functionality for stuck ETH in the contract would help to retrieve funds simply, it would also give more control over the contracts to the team. To limit admin functionality, admin withdrawals won't be implemented, but the team acknowledges that in extreme situations a workaround may be needed, which is achievable via a proxy upgrade on the `BatcherPaymentService` contract. On the other hand, while the `AlignedLayerServiceManager` contract also interacts with ETH deposits for individual batchers, there is no withdraw functionality for the users. In the event some batchers over-deposit and decide to exit the system, a withdrawal may be required. This is addressed in yetanotherco/aligned_layer#872 by introducing a `withdraw(uint256 amount)` function that retrieves `amount` of ETH from `msg.sender`'s `batchersBalance`.

Description

Both contracts are intended to receive, store, and use ETH on behalf of users. The calculations for ETH subtractions and usage are for gas purposes, which are difficult to calculate accurately.

As a result, it is probable that some amount of ETH will be stuck in the contracts with nobody being able to use or claim them.

Recommendation

Consider adding an admin authentication-protected `withdraw()` function that just sends a privileged address the funds.

4.9 After Unlocking Once, User Balances Remain Unlocked Forever Medium ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#821 by resetting the unblockBlock to zero on `withdraw()` . The client provided the following statement:

> This PR addresses an issue where user balances remained unlocked indefinitely after the first time withdraw was called. This meant that once a user's funds were unlocked when withdrawing, they remained in an unlocked state permanently, even after the initial unlock period had passed. To solve this, after the funds are withdrawn, the unlockBlock value is reset to 0 to lock the user's account again.

Description

The `unlock()` function is designed to set a timelock parameter for a user, allowing them to unlock their account at a specified future time:

contracts/src/core/BatcherPaymentService.sol:L124-L131

```
function unlock() external whenNotPaused {
    require(
        UserData[msg.sender].balance > 0,
        "User has no funds to unlock"
    );

    UserData[msg.sender].unlockBlock = block.number + UNLOCK_BLOCK_COUNT;
}
```

However, when the unlock occurs through the `withdraw()` function, the timelock status is not reset. Consequently, the user remains in an "unlocked" state indefinitely, which allows them to deposit and withdraw without restrictions:

contracts/src/core/BatcherPaymentService.sol:L138-L150

```
function withdraw(uint256 amount) external whenNotPaused {
    UserInfo storage user_data = UserData[msg.sender];
    require(user_data.balance >= amount, "Payer has insufficient balance");

    require(
        user_data.unlockBlock != 0 && user_data.unlockBlock <= block.number,
        "Funds are locked"
    );

    user_data.balance -= amount;
    payable(msg.sender).transfer(amount);
    emit FundsWithdrawn(msg.sender, amount);
}
```

After a conversation with the team, it appears that the batcher service will only take signatures from those users that are actively in the locked state. In other words, there is offchain logic that helps alleviate the impact of this issue. The only caveat is that the users would always need to lock themselves after unlocking and withdrawing.

Recommendation

Consider changing the `withdraw()` function so it resets the user's unlock status after the unlock period expires. This will prevent users from remaining in an unrestricted state indefinitely.

4.10 Direct Usage of `ecrecover()` Medium ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#787 by using openzeppelin's `ECDSA` lib for signature verification.

> Use open Zeppelin lib for signature verification instead of ecrecover

Description

The system currently uses the direct `ecrecover()` function to determine the signer of `signatureData`:

contracts/src/core/BatcherPaymentService.sol:L239-L244

```
address signer = ecrecover(
    noncedHash,
    signatureData.v,
    signatureData.r,
    signatureData.s
);
```

This approach has the following issues:

- **Error Handling**: A failed signature verification returns `0` rather than reverting with an error, potentially leading to incorrect error handling downstream.
- **Signature Manipulation**: Direct use of `ecrecover()` is vulnerable to manipulation. For example, adjusting the `signatureData.r` parameter can yield a random address, allowing for incorrect or malicious signatures to be processed as valid.

Using a function from a well-established library, such as OpenZeppelin's, can address these concerns.

Recommendation

To improve security and error handling, use the OpenZeppelin ECDSA library for signature verification. This library provides a secure and reliable implementation of `ecrecover()` that includes robust error handling and mitigates the risk of signature manipulation. Incorporating this library will ensure more accurate and secure signature verification in the contract.

4.11 [Not-in-Scope] Avoid Changing Eigenlayer-Middleware Contracts Directly Medium Acknowledged

Resolution

The implementation of Aligned contracts requires changes in logic of functions such as:

- `BLSSignatureChecker.checkSignatures`
- `RegistryCoordinator.registerOperator`
- `RegistryCoordinator.initialize`
- Contract constructors Which are non-virtual functions in the Layr-Labs created contracts. As a result, it won't be possible to override them, and a fork is the simplest option. However, the Aligned team should take note that this approach should be done cautiously, and significant care should be applied during upgrades if and when Layr-Labs contracts are updated.

Description

The Aligned team has indicated that they rely on eigenlayer-middleware patches to implement whitelisting features in the Registry Coordinator. The team provided a draft patch for review. However, we advise against directly patching the audited Layr-Labs repository for the following reasons:

- src/BLSSignatureChecker.sol and src/interfaces/IBLSSignatureChecker.sol change the function signature of `checkSignatures` to exclude `quorumNumbers` as they are hardcoded to `ALIGNED_QUORUM_NUMBER`. This is a very protocol-specific changeset that changes the Layer-Labs API which should be avoided by all costs as this will be very hard to maintain when trying to update to newer eigenlayer-middleware releases (i.e. security updates, etc.). Hardcoding the quorums to `ALIGNED_QUORUM_NUMBER` (hex 0x00) diverges from how eigenlayer-middleware is meant to be used.
- src/Whitelist.sol adds a `Whitelist` contract. The contract lacks clear visibility specifiers for the internal `whitelist` mapping (defaults to internal). Adding/Removing addresses from/to whitelist does not yield any events although it is an important administrative action (monitoring/trail of events). The contract should be `abstract` as it is meant to be inherited by other contracts.
- src/RegistryCoordinator.sol patches the stock `RegistryCoordinator` inside the eigenlayer repository (!) to include Whitelist functionality. Instead of patching the source contract in the foreign-maintained repository, it would be much better to sub-class the RegistryCoordinator by building an `AlignedRegistryCoordinator` that inherits from the original `RegistryCoordinator` overriding functionality with the patched whitelisting features. Furthermore, the changeset changes/shortens error messages emitted by the stock `RegistryCoordinator` which basically makes the contract diverge from the original API and specification. This may be problematic as the contract does not match the eigenlayer API documentation anymore.
- In general, reformatting the codebase may make it increasingly hard to diff future versions of the aligned version from the upstream.

Recommendation

Avoid making direct modifications to third-party codebases like the Layr-Labs repository. Instead, import the required components, subclass where necessary, and override functionality within the Aligned codebase. This approach preserves compatibility with the original API, reduces maintenance complexity, and ensures alignment with eigenlayer documentation and future updates.

4.12 Unnecessary Low-Level Call Medium ✓ Fixed

Resolution

Fixed with yetanotherco/aligned_layer#820 by replacing the low-level call with a contract-typed interface call.

> This PR refactors the AlignedLayerServiceManager contract to improve maintainability, code clarity, and adherence to best practices. The changes involve defining an interface for AlignedLayerServiceManager and replacing the low-level .call usage with interface calls in BatcherPaymentService.

Description

It is recommended to avoid using the low-level, untyped, and unchecked `address.call()` method when interacting with a contract. Instead, consider invoking the external contract through its defined contract type. This approach ensures type safety, correct function signature matching, and proper revert propagation.

```
AlignedLayerServiceManager.createNewTask{value:feeForAggregator}(batchMerkleRoot, batchDataPointer);
```

contracts/src/core/BatcherPaymentService.sol:L105-L117

```
// call alignedLayerServiceManager
// with value to fund the task's response
(bool success, ) = AlignedLayerServiceManager.call{
    value: feeForAggregator
}(
    abi.encodeWithSignature(
        "createNewTask(bytes32,string)",
        batchMerkleRoot,
        batchDataPointer
    )
);

require(success, "createNewTask call failed");
```

4.13 Critical State-Altering Operations Lack Event Emissions, Reducing Transparency and Auditability. Minor ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#840 by introducing the relevant events. Specifically:

- `BatcherBalanceUpdated`
- `TaskCreated`
- `BalanceLocked`
- `BalanceUnlocked`

Description

Critical state-altering operations in the contracts do not emit events, which diminishes transparency and makes it harder to audit changes effectively.

Examples

- The creation of a new task.
- The `locking` and `unlocking` of balances before withdrawal:

contracts/src/core/BatcherPaymentService.sol:L123-L136

```
function unlock() external whenNotPaused {
    require(
        UserData[msg.sender].balance > 0,
        "User has no funds to unlock"
    );

    UserData[msg.sender].unlockBlock = block.number + UNLOCK_BLOCK_COUNT;
}

function lock() external whenNotPaused {
    require(UserData[msg.sender].balance > 0, "User has no funds to lock");
    UserData[msg.sender].unlockBlock = 0;
}
```

- The `checkMerkleRootAndVerifySignatures` function, which is public and updates user balances in `verifySignatureAndDecreaseBalance` but does not emit any events.

Recommendation

Implement event emissions for all significant state changes. This will improve off-chain monitoring, enhance debugging capabilities, and increase transparency in contract interactions.

4.14 `batchDataPointer` Unchecked Minor Acknowledged

Resolution

Handled offchain by each operator. As explained by the client team:

> Operators run the checks that the data is valid. If the data is invalid, batch gets discarded. Each operator:
>
> - Downloads all the data with the proofs
> - Recalculates the merkle tree from the data (the proofs), and checks that it's root matches the root in Ethereum. Else the batch of proofs is not correct, because the leaves are not associated with the root.
> - Verifies all the proofs
> - Send the signed root if the batch is valid to the Aggregator

Description

In `createNewTask()` there is an argument `batchDataPointer` which is passed along with the `batchMerkleRoot`. After all the checks, the `batchDataPointer` is emitted as part of the event, seemingly as the data that is associated with this root exactly.

However, this isn't checked or validated, so any valid `batchMerkleRoot` may be provided to emit the event with any arbitrary, valid, or invalid `batchDataPointer` as the user would want.

This could cause confusion in some backend event-listening services.

Recommendation

Validate `batchDataPointer` as needed.

4.15 `AlignedLayerServiceManager.initialize` Should Call `ServiceManager.__ServiceManagerBase_init` Minor ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#842 by calling `__ServiceManagerBase_init()` which transfers ownership. The client provided the following statement:

> This PR updates the AlignedLayerServiceManager contract to improve the consistency and security of the initialization process by leveraging the existing mechanisms in the ServiceManagerBase contract.

Description

The `AlignedLayerServiceManager` should use the initializer from the base class `ServiceManagerBase` to handle ownership transfers rather than performing the transfer directly. This ensures that ownership changes are processed consistently and in accordance with the base class's established mechanisms.

Examples

contracts/src/core/AlignedLayerServiceManager.sol:L51-L54

```
function initialize(address _initialOwner) public initializer {
    _transferOwnership(_initialOwner);
}
```

Recommendation

Modify the `AlignedLayerServiceManager` to invoke the ownership transfer initializer provided by `ServiceManagerBase` instead of directly transferring ownership. This approach maintains consistency and leverages the base class's established initialization processes.

4.16 Solidity Coding Style Guideline Violations Minor ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#843 and later with yetanotherco/aligned_layer#892 by adhering to standard Solidity style practices using `solhint` hinter. For example:

- Custom errors are introduced
- Proper variable & argument capitalization

Description

Following the Solidity Style Guide wherever possible is advisable, especially when it comes to variable name conventions and code layout.

This can be consistently enforced across the entire code base by means of an automatic linter, such as Solhint.

Examples

- Variable names should begin lower case. Type declarations (including Contracts) begin upper case.

contracts/src/core/BatcherPaymentService.sol:L35-L40

```
address public AlignedLayerServiceManager;
address public BatcherWallet;

// map to user data
mapping(address => UserInfo) public UserData;
```

contracts/src/core/BatcherPaymentService.sol:L49-L61

```
function initialize(
    address _AlignedLayerServiceManager,
    address _BatcherPaymentServiceOwner,
    address _BatcherWallet
) public initializer {
    __Ownable_init(); // default is msg.sender
    __UUPSUpgradeable_init();
    _transferOwnership(_BatcherPaymentServiceOwner);

    AlignedLayerServiceManager = _AlignedLayerServiceManager;
    BatcherWallet = _BatcherWallet;
}
```

contracts/src/core/BatcherPaymentService.sol:L63-L66

```
receive() external payable {
    UserData[msg.sender].balance += msg.value;
    emit PaymentReceived(msg.sender, msg.value);
}
```

- Order of functions and modifiers

Declare modifiers before functions and don't mix them.

contracts/src/core/BatcherPaymentService.sol:L159-L180

```
function _authorizeUpgrade(
    address newImplementation
) internal override onlyOwner {}

// MODIFIERS
modifier onlyBatcher() {
    require(
        msg.sender == BatcherWallet,
        "Only Batcher can call this function"
    );
    _;
}

function checkMerkleRootAndVerifySignatures(
    bytes32[] calldata leaves,
    bytes32 batchMerkleRoot,
    SignatureData[] calldata signatures,
    uint256 feePerProof
) public {
    uint256 numNodesInLayer = leaves.length / 2;
    bytes32[] memory layer = new bytes32[](numNodesInLayer);
```

4.17 Storage Layout Contract Should Be Declared `abstract` Minor ✓ Fixed

Resolution

Fixed with yetanotherco/aligned_layer#841 by declaring the contract `abstract` .

Description

The `AlignedLayerServiceManagerStorage` contract defines the Service Manager storage layout without implementing contract logic. It's not intended for standalone deployment but isn't declared abstract, allowing potential misuse. This violates the principle of designing contracts for their intended purpose and could lead to unexpected behavior if deployed independently.

Examples

contracts/src/core/AlignedLayerServiceManagerStorage.sol:L5-L20

```
contract AlignedLayerServiceManagerStorage {
    struct BatchState {
        uint32 taskCreatedBlock;
        bool responded;
        address batcherAddress;
    }

    /* STORAGE */
    mapping(bytes32 => BatchState) public batchesState;

    // Storage for batchers balances. Used by aggregator to pay for respondToTask
    mapping(address => uint256) internal batchersBalances;

    // storage gap for upgradeability
    uint256[48] private __GAP;
}
```

Recommendation

Modify the contract declaration to include the 'abstract' keyword, changing it to: `abstract AlignedLayerServiceManagerStorage` . This will prevent direct deployment of the contract and clearly indicate its intended use as a base contract for inheritance.

4.18 Unused Imports Minor ✓ Fixed

Resolution

Fixed with yetanotherco/aligned_layer#843 by removing the unused imports. We would like to note that the `ServiceManagerStorage` contract may optionally inherit `IServiceManager` .

Description

The following source units are imported but not referenced in the contract:

- `AlignedLayerServiceManager` unused `IPauserRegistry` , unused `Pausable`

contracts/src/core/AlignedLayerServiceManager.sol:L4-L6

```
import {Pausable} from "eigenlayer-core/contracts/permissions/Pausable.sol";
import {IPauserRegistry} from "eigenlayer-core/contracts/interfaces/IPauserRegistry.sol";
```

- `AlignedLayerServiceManagerStorage.sol` should inherit `interface` `IServiceManager`

contracts/src/core/AlignedLayerServiceManagerStorage.sol:L3

```
import "eigenlayer-middleware/interfaces/IServiceManager.sol";
```

Recommendation

Remove unused imports and implement the IServiceManager interface in `AlignedLayerServiceManagerStorage` . Conduct a thorough review of all contracts to identify and eliminate any other unused imports or missing interface implementations.

4.19 Provide an Interface for `AlignedLayerServiceManager` ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#820 by providing an interface for the `AlignedLayerServiceManager` .

Description

The `AlignedLayerServiceManager` contract extends the `ServiceManagerBase` contract by adding public interfaces. To ensure proper contract interaction and adherence to best practices, it is recommended to define an interface contract, `IAlignedLayerServiceManager`, that outlines these public methods. The `AlignedLayerServiceManager` contract should then inherit from this interface.

contracts/src/core/AlignedLayerServiceManager.sol:L18-L22

```
contract AlignedLayerServiceManager is
    ServiceManagerBase,
    BLSSignatureChecker,
    AlignedLayerServiceManagerStorage
{
```

Recommendation

Define an interface contract `IAlignedLayerServiceManager` that includes the public methods exposed by `AlignedLayerServiceManager`. Ensure that `AlignedLayerServiceManager` inherits from this interface to formalize the contract's API and enhance code clarity and interaction with external contracts.

4.20 Where Possible, a Specific Contract Type Should Be Used Rather Than `address` ✓ Fixed

Resolution

Addressed with yetanotherco/aligned_layer#820 and yetanotherco/aligned_layer#853 by declaring `AlignedLayerServiceManager` with its contract type instead of `address`.

Description

Consider using the best type available in the function arguments and declarations instead of accepting `address` and performing low-level calls or later casting it to the correct type.

- `address AlignedLayerServiceManager` should be `AlignedLayerServiceManager serviceManager`

contracts/src/core/BatcherPaymentService.sol:L34-L36

```
// STORAGE
address public AlignedLayerServiceManager;
address public BatcherWallet;
```

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/src/core/AlignedLayerServiceManager.sol	`9846645b47a4a899dbf42caf72a3800f71b5ed4e`
contracts/src/core/AlignedLayerServiceManagerStorage.sol	`50a9257f5d313e2dda87285e95a2d3a7346dc9d6`
contracts/src/core/BatcherPaymentService.sol	`d96289bd7c092138a2f677c26b2f99863e47bc0c`

Appendix 2 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.