



Aligned

Security Assessment (Summary Report)

December 5, 2024

Prepared for:

Elio Rutigliano

Aligned Layer

Prepared by: **Joop van de Pol, Alexander Remie, Marc Ilunga, Xiangang He, and Jim Miller**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at <https://github.com/trailofbits/publications>, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow [@trailofbits](#) on Twitter and explore our public repositories at <https://github.com/trailofbits>. To engage us directly, visit our "Contact" page at <https://www.trailofbits.com/contact>, or email us at info@trailofbits.com.

Trail of Bits, Inc.

497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215

<https://www.trailofbits.com>

info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2024 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

This report is considered by Trail of Bits to be public information; it is licensed to Aligned Layer under the terms of the project statement of work and has been made public at Aligned Layer's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

The sole canonical source for Trail of Bits publications is the [Trail of Bits Publications page](#). Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits	1
Notices and Remarks	2
Table of Contents	3
Project Summary	4
Project Targets	5
Executive Summary	6
Findings	8
1. Inconsistency between batcher and operator can lead to denial of service	8
2. The SDK silently ignores incorrect inclusion proofs from the batcher	8
3. Batcher does not perform sufficient input validation	8
4. Issues in operator Dockerfile	9
5. Null pointer dereference in the Operator codebase can lead to denial of service	9
6. Keystore encryption is malleable	9
7. Outdated dependencies	9
8. No enforcement of strong passwords for keystores	10
9. Missing ecrecover result validation	10
10. Missing events in Solidity contracts	10
11. Front-running createNewTask leads to loss of funds	11
12. Inconsistency in the balance unlocking mechanism	11
13. Usage of storage gaps in upgradeable contracts	12
14. Usage of block count instead of block timestamp	12
A. Missing Input Validation	13
Batcher	13
Operator	13
B. Code Quality Findings	14
C. Fix Review Results	17
Detailed Fix Review Results	17

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering directors were associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Xiangnan He, Consultant
xiangan.he@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 8, 2024	Pre-project kickoff call
August 19, 2024	Delivery of report draft
August 20, 2024	Report readout meeting
December 5, 2024	Delivery of final summary report with fix review

Project Targets

The engagement involved a review and testing of the following target.

Aligned

Repository	https://github.com/yetanotherco/aligned_layer
Version	325aef8c3f54ec596b4733956a8ac487d5535fc3
Type	Go, Rust, Solidity
Platform	Native, EVM

Executive Summary

Engagement Overview

Aligned Layer engaged Trail of Bits to review the security of **Aligned**. Aligned is a decentralized network to provide an Actively Validated Service (AVS) for verifying zero-knowledge proofs, with verification results posted to the Ethereum blockchain. Users can provide zero-knowledge proofs (ZKPs) generated using the proof systems that Aligned supports to a centralized batcher (or run their own batcher if they prefer). The batcher gathers up a batch of such proofs and provides this batch to the Aligned operators. The operators verify the proofs and provide digital signatures to the aggregator, which verifies whether a quorum has been reached in order to post the results on-chain.

A team of five consultants conducted the review from August 12 to August 16, 2024, for a total of three engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the Aligned Layer codebase with tag v0.4.0, using automated and manual processes.

Observations and Impact

The review focused on assessing whether users can cause invalid proofs to be accepted; whether user funds deposited into the centralized batcher can be stolen (except by the centralized batcher itself); and whether it is possible to bring down the system as part of a denial-of-service (DoS) attack. To this end, we reviewed the aggregator, batcher, configuration, and operator components, as well as the core smart contracts.

However, the scope of the review did not include the underlying dependencies that provide functionality, such as ZKP verification and EigenLayer Core tools (including but not limited to the BaseServiceManager, operator registration, and BLS signature aggregation). Additionally, Aligned Layer was already aware of the following issues in the codebase, placing them outside the scope of the review:

- Absence of input size validation leads to out-of-bounds accesses in the Foreign Function Interface (FFI) implementation of the operators' `verifies` and `verify` functions.
- Users creating a task are vulnerable to a front-running attack leading to a DoS, caused by replaying the same transaction in the mempool with different parameters.
- Operators are vulnerable to an attack that causes them to run out of memory (OOM) by sending them a crafted gzip file.
- A crafted HTTP response can bypass the operator's length check when retrieving proof data.

- A batcher can create a task without depositing money.

The main issues identified during the audit related to DoS attacks and the wasting of user funds. While we were unable to exploit these issues to steal user funds or cause invalid proofs to be accepted, these issues could still harm users and reduce user trust in the system. The remainder of the findings concern less-severe issues that, while not directly exploitable, should be fixed as part of a defense-in-depth approach.

Recommendations

Implementing the following recommendations will mitigate the findings in this report:

- Unify the code that is currently duplicated between batcher and operator. This unified implementation (and all other parts of the implementation) should additionally perform all the required user input validation checks to prevent crashes based on user input.
- Ensure that the SDK gives the user appropriate feedback when inclusion proofs fail. Replace code panics by error returns and the corresponding handling wherever possible.
- Adapt the operator dockerfile to upgrade the packages, clean up the package cache and lists, and restrict the user role of running processes where possible.
- Strengthen the smart contracts by marking sensitive functionality that interacts with user funds as private, and by including additional checks and events where applicable.
- Instruct the relevant system participants to choose secure passwords.
- Ensure that dependencies are up to date.

Finally, we highly recommend auditing critical dependencies, such as EigenLayer and the ZKP verification back ends, given the extent to which the Aligned system's security relies on these components.

Findings

1. Inconsistency between batcher and operator can lead to denial of service

The batcher and operator components each have their own implementations for verifying user proofs. The batcher optionally uses its implementation to pre-verify user proofs to prevent users from causing a whole batch to fail with a bad proof.

However, this only helps if the batcher will reject all proofs that operators will reject as well. The batcher implementation of the halo2 IPA and KZG proofs restricts the maximum sizes of the constraint system, verifier key, and parameters. The operator implementation additionally defines maximum sizes for the proof and public inputs. It then calls a Rust function via FFI, which defines its own (inconsistent) maximum sizes for these items.

As a result, a correct proof where either the proof parameter or the public input parameter exceeds 4KB will pass the batcher pre-verification but fail the operator verification, as the corresponding array will be truncated (or the operator will crash due to out-of-bounds access).

2. The SDK silently ignores incorrect inclusion proofs from the batcher

When the user submits one or more proofs to the centralized batcher via the SDK (or the CLI, which uses the SDK), the implementation sends a message for each proof and expects a corresponding response for each message. Each response contains the batch Merkle root and an inclusion proof for the user input. The implementation verifies the inclusion proof and returns the corresponding Aligned verification data only if the inclusion proof is correct. An incorrect proof will not cause an error, but merely prevents the return of the corresponding Aligned verification data.

An incorrect proof does not prove exclusion from the Merkle tree and therefore from the batch. If the user proof is in the batch, the centralized batcher will charge the user when submitting this batch. However, the user has no way to verify the on-chain results of this proof at a later time, and this may cause the user to resubmit the same proof and get charged again. Note that this is not considered a valid attack, because exploitation would require a malicious centralized batcher. However, it may lead to inadvertent loss of funds if there is ever a flaw in the batcher implementation.

3. Batcher does not perform sufficient input validation

Users that send malformed input to the centralized batcher will cause it to panic. Sending a non-text message or one that fails deserialization will cause a panic when the batcher handles the message. Another type of panic is caused by the fact that user messages are deserialized to a type that is meant to cover all proof systems. This type has options for each of the contained vectors, and the vectors can be arbitrary length. Users can submit

proofs with missing components or with vectors of the wrong length, causing the batcher to crash. For a full list of input validation issues, see [appendix A](#), whereas [appendix B](#) lists cases where the batcher needlessly panics instead of returning an error.

4. Issues in operator Dockerfile

The operator Docker implementation calls `apt-get update` twice, but this merely updates the package lists. It should also perform `apt-get upgrade` to update the packages themselves. Removing the package cache and lists after installing and upgrading packages will also reduce the image size. Additionally, the implementation does not specify a user in the `ENTRYPOINT`, so programs inside can run as root. This is a security hazard: if an attacker can control a process running as root, they may have control over the container.

5. Null pointer dereference in the Operator codebase can lead to denial of service

Users of the Aligned system may provide Risc0 proofs for verification. Operators verify Risc0 proofs by calling the FFI function `verify_risc_zero_receipt_ffi`. This function takes raw pointers to the proof receipt, an image ID, and the public input bytes. The Rust code that performs the verification does not check that the public input pointer (`public_input`) is not null before dereferencing. As a consequence, a malicious batcher may provide a maliciously crafted JSON that does not include a public input. As a consequence, all operators processing the malicious `VerificationData` will crash.

6. Keystore encryption is malleable

An operator's BLS keypair is stored encrypted in a configuration file. The encryption scheme used for encrypting keystores does not guarantee integrity of ciphertexts. Therefore, a local attacker may force an operator to use a different BLS signing key.

The keystore encryption scheme is provided by go-ethereum (decryption is implemented by the function `DecryptDataV3`). It attempts to build an authenticated encryption scheme by using AES in CTR mode and a Keccak-based MAC scheme. The go-ethereum implementation fails to MAC the IV. As a result, an attacker can change the IV while satisfying the MAC verification algorithm. The issue does not provide much advantage to an attacker. The properties of CTR modes guarantee that each new IV will lead to a random BLS key.

7. Outdated dependencies

`cargo-audit` identified outdated dependencies with advisories in the codebase. Using outdated dependencies increases the application's attack surface. Keeping dependencies updated, wherever possible, is a good practice to reduce the attack surface.

Dependency	Version	ID	Description
------------	---------	----	-------------

openssl	0.3.21	RUSTSEC-2024-0357	MemBio::get_buf has undefined behavior with empty buffers
ansi_term	0.12.1	RUSTSEC-2021-0139	ansi_term is unmaintained
dotenv	0.15.0	RUSTSEC-2021-0141	dotenv is unmaintained

Table 1: Outdated Rust dependencies

8. No enforcement of strong passwords for keystores

In various locations of the Aligned documentation, such as the [Submitting Proofs guide](#), users are directed to set up their password-protected keystore using the CLI tool `cast wallet`, which will encrypt a private key using a user-supplied password. To decrypt this keystore, the user supplies this password, and the Aligned codebase uses the `go-ethereum` keystore logic to decrypt the encrypted data. However, neither `cast wallet` nor `go-ethereum` verifies the password's strength. In particular, nothing prevents users from using extremely small, common, or empty passwords to protect their critical private keys. Consider adding a warning to the documentation to strongly recommend that users use strong passwords. Alternatively, consider adapting the `cast wallet` CLI tool to add requirements for the password's strength.

9. Missing ecrecover result validation

The `verifySignatureAndDecreaseBalance` function in the `BatcherPaymentService` contract uses `ecrecover` to retrieve the signer of each leaf in the Merkle tree. The Solidity `ecrecover` operation does not revert when the signature is invalid; instead, it returns the zero address. The current implementation does not validate that the `ecrecover` result is not address zero. Due to the surrounding code, this is currently not exploitable.

A second missing validation involves ensuring that the signature's s-value is in the upper range. This is to prevent a known malleable signature problem where flipping both the s-value and the v-value from the lower to the higher range results in the same address being recovered. For more information, see [this comment](#) in the OpenZeppelin ECDSA implementation.

To solve both of the above described problems, we recommend using the battle-tested OpenZeppelin [ECDSA](#) library, which includes both of these validations. This library has stood the test of time and is widely used in Solidity smart contracts.

10. Missing events in Solidity contracts

Several functions in the Solidity smart contracts do not emit events. Emitting events when important state variables are updated is considered a best practice and also helps to improve the ability for off-chain monitoring by listening for these events. The following functions should emit an event:

- `BatcherPaymentService.lock`
- `BatcherPaymentService.unlock`
- `Whitelist.add` and `Whitelist.remove` (located in the modified `EigenLayer` code that Aligned pushed into an [eigenlayer-middleware PR](#) for visibility during this audit)

11. Front-running `createNewTask` leads to loss of funds

The Aligned team is aware of a front-running issue in the `createNewTask` function in the `AlignedServiceManager` contract. However, a more damaging front-running issue is present in the `BatcherPaymentService` contract that could cause users to lose their deposited funds without receiving anything in exchange.

The `checkMerkleRootAndVerifySignatures` function should not be publicly callable. This function is called internally from the `BatcherPaymentService.createNewTask` function, which is a privileged function (i.e., it can only be called by the Batcher account).

The `checkMerkleRootAndVerifySignatures` function will verify the signatures of each leaf in the Merkle tree and deduct the `feePerProof` from the balance of the signer of each leaf. The purpose of this is to make each leaf entry pay for being included in a batch, which is what the `createNewTask` function will initiate after verifying the signatures and deducting balances.

An attacker could front-run any call to the privileged `createNewTask` function by calling the `checkMerkleRootAndVerifySignatures` function. This will correctly verify the signatures and deduct the balances. However, since it also increments the nonce of each signer, the legitimate privileged call to the `createNewTask` function will revert due to an incorrect nonce. In other words, the balance of each signer will be deducted, but no new task will be created, and users will have paid in exchange for nothing.

We recommend setting the visibility of the `checkMerkleRootAndVerifySignatures` function to internal or private..

12. Inconsistency in the balance unlocking mechanism

The `BatcherPaymentService` contract implements a lock/unlock mechanism. Whenever an account transfers ETH to the contract, the amount will be tracked in a mapping (where the key is the account's address). When an account wants to withdraw, the account needs to first be unlocked (which has a default delay of 100 blocks).

The default locking state of an account is to be locked. An account can only be locked/unlocked when there is a positive balance. When the balance of an account reaches zero, the current implementation does not automatically reset the locking state to "locked." This might confuse users, as even though you cannot lock/unlock when the balance is zero, the current account state might be unlocked even though the balance is zero.

We recommend updating the implementation to reset the locking status to locked whenever the balance reaches zero, either in the `withdraw` function or through the `createNewTask` function.

13. Usage of storage gaps in upgradeable contracts

The mechanism of adding a “gap” variable in an upgradeable contract is used to allow inherited contracts to also add new variables during an upgrade. The “gap” variable in each inherited contract is also usually set to 50 slots. Whenever a new upgrade adds variables, the corresponding “gap” variable is reduced by however many slots were used by the newly introduced variables.

We recommend updating all of the `__GAP` variables to be of size 50, and adhering to the above-described process for decrementing these variables during future upgrades. Second, we recommend removing the `__GAP` variable from both the `BatcherPaymentService` and `AlignedLayerServiceManager` contracts; this is because these contracts are not inherited by any other contracts, and the `__GAP` variable therefore serves no purpose.

14. Usage of block count instead of block timestamp

The account balance locking mechanism in the `BatcherPaymentService` contract uses a delay expressed in block count (i.e., the number of blocks that need to pass before the account becomes unlocked). This delay is hard coded to 100 blocks.

Ethereum block time is currently around 12 seconds, meaning the delay is around $12 * 100 = 1,200$ seconds, or 20 minutes. Using the block count is less intuitive than using a block timestamp, which informs the user of the exact time that his balance becomes unlocked right at the moment he calls the `unlock` function. By contrast, with a block count, the time cannot be exactly known up front due to the block time variability.

Consider using a delay in seconds instead of block count.

A. Missing Input Validation

As described in the executive summary, Aligned Layer was aware of missing length verification checks in the operator prior to this security review. However, to ensure that all missing length verification checks are resolved, we provide the following list of all missing checks that we observed during the review, including the operator.

Batcher

- In `zk_utils::verify_internal`, a user can trigger a panic:
 - in each of the expect statements in lines 29, 34, 43, 48, 76, and 81
 - in the function `copy_from_slice` on line 59 by providing an `image_slice` of incorrect length.

Operator

- In the `risc0` FFI function `verify_risc_zero_receipt_ffi`,
 - providing a null pointer for `public_input` will cause a crash as it is accessed in unsafe code in line 20.
 - a user can trigger a panic in the function `copy_from_slice` on line 23 by providing an `image_id` of incorrect length.
- In the `halo2` FFI function `verify_halo2_ipa_proof_ffi`, users can cause panics in the array slicing in lines 46, 48, 53, 55, and 59 by providing lengths larger than the maximum length.
- In the `halo2` FFI function `verify_halo2_kzg_proof_ffi`, users can cause panics in the array slicing in lines 48, 50, 55, 57, and 61 by providing lengths larger than the maximum length.
- In the `(*Operator).verify` function,
 - a user can trigger a panic in any of the slicing in lines 251, 256, 261, 267, 272, 277, 305, 310, 315, 321, 326, and 331.
 - the size comparisons for the buffer lengths and their actual lengths for `halo2` IPA and KZG are missing, which can cause panics in the corresponding FFI functions.
- While the `verify_sp1_proof_ffi` and `verify_merkle_tree_batch_ffi` FFI functions do not have any length checks, it is currently not possible to trigger any panic, as the lengths are set appropriately in the corresponding operator go code.

B. Code Quality Findings

The following findings are not associated with any specific vulnerabilities. However, they will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Public functions that dereference raw pointers are not marked as unsafe.** Several functions in the Rust codebase of operators take raw pointers as input. These functions should be marked as unsafe so the function caller verifies the inputs passed to these functions. Although some functions do internally check for null pointers, **not all pointers are checked**. The functions that should be marked as unsafe are: `verify_risc_zero_receipt_ffi`, `verify_sp1_proof_ffi`, and `verify_merkle_tree_batch_ffi`.
- **Long function bodies.** The **main function** in `batcher/aligned/src/main.rs` is fairly long, which impairs readability of the codebase. Consider breaking the function into subfunctions to improve the codebase's readability and maintainability.
- **Unnecessary panic in a critical component.** The Aligned batcher matches on the proof system ID to call the appropriate zero-knowledge back-end verifier. For Gnark proofs, the default match arm leads to a panic of the batcher, as shown below. This code path is currently not reachable. However, the batcher should not crash due to an unknown proof system ID, as this represents an easily exploitable denial-of-service vector.

```
match proving_system {
    ProvingSystemId::GnarkPlonkBn254 => unsafe {
        VerifyPlonkProofBN254(proof, public_input, verification_key)
    },
    ProvingSystemId::GnarkPlonkBls12_381 => unsafe {
        VerifyPlonkProofBLS12_381(proof, public_input, verification_key)
    },
    ProvingSystemId::Groth16Bn254 => unsafe {
        VerifyGroth16ProofBN254(proof, public_input, verification_key)
    },
    _ => panic!("Unsupported proving system"),
}
```

*Figure B.1: Default match arm result in a panic
(aligned_layer/batcher/aligned-batcher/src/gnark/mod.rs#40–51)*

- **Functions that return a Result should not panic.** Several Rust functions return a `Result`, but can also panic due to `expect` or `unwrap` statements. This is typically unnecessary, as these functions can return an error instead. The following functions are affected:

- `Halo2::ipa::read_fr` (as well as the FFI variant in the operator)
 - `halo2::kzg::read_fr` (as well as the FFI variant in the operator)
 - `Batcher::listen_connections`
 - `Batcher::listen_new_blocks`
 - `Batcher::handle_message`
 - `Batcher::finalize_batch`
 - `Batcher::submit_batch`
 - `Batcher::handle_nonpaying_msg`
 - `aligned_batcher::main`
 - `aligned::main`
- **It is not necessary to check `err != nil` when returning `err`.** In various places, the implementation performs a redundant check on the error return of a called function, as shown in figure B.2. Instead, `err` can be directly returned. This occurs in the functions `aggregatorMain`, `(*Aggregator).ServeOperators`, and `(*AvsWriter).SendTask`.

```
if err != nil {
    return err
}

return nil
```

Figure B.2: Superfluous `nil` err check before return
([aligned_layer/aggregator/cmd/main.go#61–65](#))

- **Unused variable.** The `_hashOfSigners` variable is not used in the source code. It is customary in Solidity to not mention such variables.

```
113    (
114        QuorumStakeTotals memory quorumStakeTotals,
115        bytes32 _hashOfNonSigners
116    ) = checkSignatures(
```

Figure B.3: Function return variable assignment of an unused value
([aligned_layer/contracts/src/core/AlignedLayerServiceManager.sol#L113–L116](#))

```
(
    QuorumStakeTotals memory quorumStakeTotals,
) = checkSignatures(
```

Figure B.4: Proposed change to not mention an unused variable

- **Merge the `AlignedLayerServiceManagerStorage` contract into the `AlignedLayerServiceManager`.** There is no added value in splitting these two contracts. If you were to use an upgradeability mechanism that did not use a proxy-based `delegatecall` pattern, having a separate storage contract would

make sense to upgrade the logic without upgrading the storage (by replacing the logic contract, but still using the same storage contract); however, in the current setup, it does not add any value.

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the system design, source code, and system configuration, not comprehensive analysis of the system. In other words, we examine the commit or pull request (PR) for each fix to determine whether it properly resolves the issue; however, we do not spend additional time investigating whether the fix introduced new issues into the system.

On November 4, 2024, Trail of Bits reviewed the fixes and mitigations implemented by Aligned Layer for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Detailed Fix Review Results

1. Inconsistency between batcher and operator can lead to denial of service

Resolved in [PR#738](#). The batcher and operator use the same implementation to read proof verification inputs. Furthermore, they both rely on utility functions from the `halo2_proofs` crate to read proof verification inputs.

2. The SDK silently ignores incorrect inclusion proofs from the batcher

Partially resolved. The fixes in v0.8.0 introduce new errors and a pre-verification option for the batcher that returns an error if the user submits an invalid proof. However, these added functionality do not address the initial issue. Namely, a faulty or malicious batcher can still send bogus proofs to the user. The SDK will still silently ignore incorrect inclusion proofs since the client functionality has not been updated to handle incorrect inclusion proofs. The Aligned Layer team is planning to introduce more mitigations in future releases.

3. Batcher does not perform sufficient input validation

Resolved in [PR#738](#). Length checks and pointer validations were added to validate user-provided inputs.

4. Issues in operator Dockerfile

Resolved in [PR#733](#). All Docker files have been deleted to remove support for running operators in Docker.

5. Null pointer dereference in the Operator

Resolved in [PR#738](#). The pointer `public_input` is checked. If it is a null pointer, it is set to be a pointer to a newly defined empty slice.

6. Keystore encryption is malleable

Unresolved. The keystore encryption functionality is implemented by go-ethereum. A fix, therefore, requires an upstream update.

7. Outdated dependencies

Resolved. The crate `dotenv` was replaced by `dotenvy`, and the crate `bytes` was updated to the latest version. Two upstream dependencies, `bytemuck` and `ansi-term`, cannot be updated without coordination with external development teams.

8. No enforcement of strong passwords for keystores

Resolved. Aligned Layer added a warning for users to consider choosing strong passwords to secure key stores. We recommend further investigating and implementing a length check on user passwords.

9. Missing ecrecover result validation

Resolved in [PR#787](#). The OpenZeppelin ECDSA library is now used to verify signatures.

10. Missing events in Solidity contracts

Resolved in [PR#840](#) and [PR#4](#). The missing events have been added to the implementation.

11. Front-running `createNewTask` leads to loss of funds

Resolved in [PR#801](#). The visibility of the `checkMerkleRootAndVerifySignatures` function has been updated to private.

12. Inconsistency in the balance unlocking mechanism

Resolved in [PR#821](#). The implementation has been updated to lock an account every time the `withdraw` function is called, even if the amount withdrawn is not the full withdrawable amount.

13. Usage of storage gaps in upgradeable contracts

Resolved in [PR#916](#). The unnecessary storage gaps have been removed.

14. Usage of block count instead of block timestamp

Resolved in [PR#1085](#). The implementation has been updated to use a timestamp instead of a block number to determine the point in time when a user's balance becomes "unlocked".

Appendix A. Missing input validation

Partially resolved in [PR#738](#). The fixes implemented are as follows:

Batcher: Partially resolved

- There is no length check ensuring that `image_id_slice` is the same length as the slice `image_id`. Therefore, a panic will occur when `image_id_slice` is an incorrect length.

Operator: Resolved

- The pointer `public_input` in the `risc0` FFI function `verify_risc_zero_receipt_ffi` is checked to ensure that it is not null before being accessed.
- Slicing has been removed from all of the relevant locations.
- All panic-inducing code in `verify_internal` has been replaced by appropriate error handling.