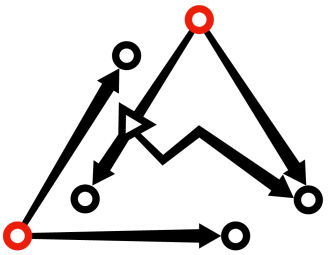

Least Authority

PRIVACY MATTERS

Operator AVS + Smart Contracts

Security Audit Report

Aligned Layer

Final Audit Report: 3 December 2024

Table of Contents

Overview

Background

Aligned Layer has requested Least Authority perform a security audit of their Operator AVS and smart contracts. Aligned Layer is a verification layer for zero-knowledge proofs using Eigen Layer.

Project Dates

- **August 13, 2024 - August 26, 2024:** Initial Code Review *(Completed)*
- **August 28, 2024**: Delivery of Initial Audit Report *(Completed)*
- **December 2, 2024** Verification Review *(Completed)*
- **December 3, 2024:** Delivery of Final Audit Report *(Completed)*

Review Team

- Will Sklenars, Security Researcher and Engineer
- Dominic Tarr, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Operator AVS and smart contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Aligned Layer:
  https://github.com/yetanotherco/aligned_layer

Specifically, we examined the Git revision for our initial review:

- `325aef8c3f54ec596b4733956a8ac487d5535fc3`

For the verification, we examined the Git revision:

- `1125be82b7c149bfe27932e6ee3a4c0ff00a1c5b`

For the review, this repository was cloned for use during the audit and for reference in this report:

- Aligned Layer:
  https://github.com/LeastAuthority/Aligned-Layer

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://alignedlayer.com

- Aligned Layer Documentation: https://docs.alignedlayer.com
- Aligned Layer Whitepaper: https://alignedlayer.com/whitepaper
- `Eigenlayer` Whitepaper: https://docs.eigenlayer.xyz/html/EigenLayer_WhitePaper-converted-xodo.html#bookmark34

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Aligned Layer is a protocol that provides fast and inexpensive verifications of zero-knowledge proofs. Aligned Layer is built on top of `EigenLayer`, which is a platform for bootstrapping new networks, leveraging the security of the Ethereum ecosystem through exposing validators that staked ETH to secondary slashing risk.

Aligned Layer is a distributed event-based system. The lifecycle of a batch of proofs ends once two-thirds of the Aligned Layer validators (called Operators) reach consensus, and have their signatures aggregated and written back to the blockchain.

In our review, we identified powerful attacks on all major components of the system that are able to prevent its correct operation. While the attacks cannot lead to the injection of false proofs, they can prevent proofs from being processed for a particular user, a particular batch, or the system overall. The attacks start when a component interacts with the outside world (or other components) and succeed because the components are not sufficiently defensive against invalid messages or misbehaving servers.

System Design

Aligned Layer is a proof aggregation system that allows users to submit zero-knowledge proofs and then checks them in a manner that is more cost-effective than verifying the proof in an Ethereum contract. Proofs are submitted to the Batcher, which collects them and creates a task on an Ethereum contract, uploading the batch data to Amazon s3. A third party may also create batches directly and provide the data via any `http` URL. Operators listen for new batches and download the data, then verify the batch and send a signed response to the Aggregator. The Aggregator collects BLS signatures and aggregates them.

Once the Aggregator has signatures from two-thirds of the Operators, the aggregated signature is submitted to the Ethereum contract.

In our review of the design of the system, we found that there is no sufficient defense against invalid messages, or other servers behaving incorrectly (either too slow or too fast).

In several cases (Issue B, Issue C), the vulnerability is due to mutating state before fully validating a message. However, receiving an invalid message should not affect the state of the system, even briefly. In addition, our team identified some concerns around the effect that downtime could have on the system. While it is inevitable that, sometimes, components will have to be taken down so that they can be updated for example, it is also possible that outages could cause components to stop functioning as intended (Issue D, Issue E). The solution is to design for eventual consistency; that is, to ensure that messages that cannot be delivered due to a failure or outage are delivered later in some way.

Although the verification of a zero-knowledge proof is effectively an objective test of the proof, Ethereum has limited support for on-chain proof verification, and it is expensive. To significantly lower the cost, and also support different proof systems, Aligned Layer instead relies on off-chain verification and a two-third majority consensus (note that this implies a one-third rejection threshold). Aligned Layer has not implemented any form of slashing (although it is described in their whitepaper) and has informed our team that they may launch without it. While this did prompt our team to consider the likelihood of economic consensus attacks, we later identified several, other straightforward attacks that can achieve the same objective but while incurring much less cost to the attacker.

Slashing would be handled via `EigenLayer`. However, `EigenLayer` is yet to implement slashing and extend that functionality to its users. We also noted that the `EigenLayer` whitepaper discussed veto committees, which have the power to reverse a slashing event should one occur. The use of a veto committee could make the system more prone to Proof Of Authority style vulnerabilities, as successfully slashing malicious behavior would either require members of the veto committee to be altruistic and not have had their accounts compromised, or otherwise require that the reputational damage of incorrectly vetoing a slashing event be less than the penalty of the slashing event combined with the potential profit of the associated attack. However, at the time of our audit, `EigenLayer` is yet to implement slashing or veto committees.

Additionally, it is not standard practice for a zero-knowledge system to accept unrestricted proofs, with the attacker controlling the proof, the inputs, the circuit, etc. This opens an attack vector for different kinds of attack, and we therefore recommend considering performing a comprehensive security audit specifically focused on this area of investigation.

Code Quality

We performed a manual review of the repositories in scope and found the codebases to be generally organized, well-written, and easy to read. The Operator and Aggregator are implemented in Go, the Batcher is implemented in Rust, and the `AlignedLayerServiceManager` contract is implemented in Solidity.

Tests

The in-scope repositories include one test for the `AlignedLayerServiceManager` contract; however, there are no other tests for the Batcher, or Aggregator (Suggestion 1).

Documentation and Code Comments

The project documentation provided by the Aligned Layer team was sufficient in describing the intended functionality of the system. Additionally, we found that code comments sufficiently describe the intended behavior of security-critical components and functions.

Scope

The scope of this review was sufficient, as it included the entire system.

Dependencies

Aligned Layer's main dependency is `Eigenlayer`. The `AlignedLayerPaymentService` contract and the Operator, as well as the Aggregator code, inherit from base classes provided by `Eigenlayer`. This imposes limitations on Aligned Layer. For example, in the Aggregator, it is necessary to use an integer task ID where the `merkleBatchRoot` would be a more natural identifier, and this makes the code more complex than necessary. Additionally, due to the limitations of `EigenLayer`'s `BlsAggregationService`, it is only possible for Operators to vote that a batch is accepted. Consequently, if a batch is invalid or unavailable, there is no way for an Operator to indicate that. So a problem with a batch is indistinguishable from Operators colluding to not respond. However, `EigenLayer` is still under active development, and features such as slashing or rewards based on Operator behavior have not been implemented yet.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Denial of Service (DoS) Attack on Operators via Slow Http Server	Resolved
Issue B: Attack on Aggregator Race Condition Censoring a Batch	Resolved
Issue C: Replay Attack on Batcher Disrupts a Particular User	Resolved
Issue D: Aggregator or Operator Downtime Can Break Eventual Consistency	Unresolved
Issue E: Aggregator Memory Leak Could Lead to Denial of Service	Resolved
Suggestion 1: Add More Tests	Unresolved

Issue A: Denial of Service (DoS) Attack on Operators via Slow Http Server

Location

`operator/pkg/operator.go#L154`

`operator/pkg/s3.go#L14-L60`

Synopsis

An attacker can create a batch with a data pointer to a URL on a server they control, then provide that data very slowly causing Operators to effectively stop.

Impact

Operators will start to download a batch and never finish. Because of the way they listen for events and process them, Operators will be stuck in the 'event receive' step and will not process new batches until they either receive a complete batch or the `http` request terminates in some way. Effectively, the attack can temporarily stop the system from functioning as intended.

Preconditions

The attacker would need a custom `http` server that responds fast enough to keep `http` requests alive but also as slowly as possible to stall the Operator for as long as they can. They would also need a small amount of Ethereum to create a verification batch directly. This is an active attack, and the attacker would need to restart it occasionally for full impact.

Feasibility

If the preconditions are met, the attack is trivial.

Technical Details

The attacker must bypass the Batcher by calling `BatcherPaymentService.createNewTask` and creating a batch on the blockchain directly. This is an officially supported option that users are invited to use if they suspect the official Aligned Layer Batcher of censoring proofs. The new task contains a 'data pointer,' which is a URL to the batch data. When the Aligned Layer Batcher creates batches, they will be stored on S3, but on an attacker-created batch, the URL can be anything – for example, a server they control. When the new task is created, a `NewBatch` event is emitted, and Operators listen to this. When they receive the message, they start downloading it from the `batchDataPointer` (URL) if the attacker responds to this very slowly. The attacker should respond fast enough that the connection is not dropped, but so slowly that it takes hours to download the data. A packet every few seconds would be sufficient.

The Operator handles events in a `select` inside a `for` while synchronously calling `ProcessNewBatchLog`. This means that the `for select` will wait until `ProcessNewBatchLog` returns, but the attacker will not let this happen for a long time. `ProcessNewBatchLog` starts by calling `getBatchFromS3`, which, despite its name, will download from any `http` URL. `getBatchFromS3` first performs a `HEAD` request and checks if the length header is less than the configured `MaxBatchSize`, and, if it is, it then proceeds to download the batch data from the URL, which is precisely where the attack occurs. The server provides the data very slowly; consequently, the Operator gets stuck and does not process new batches for as long as the request is ongoing. If one-third of the Operators are blocked in this manner, the entire Aligned Layer system, and thus any other systems built on it, become unable to process proofs until the request ends. In addition, checking the length in a separate `HEAD` request protects the system if the server hosting the data pointer correctly follows the `http` specification. However, there is no way to guarantee that the length given by the `HEAD` request is actually the length of the data given in the actual request.

Mitigation

There is no way for current users or Operators to mitigate this attack.

Remediation

We recommend running `ProcessNewBatchLog` inside a concurrent Go routine so that it does not block the main `for select` loop. As a result, a slow server will only make that particular batch slow and will not interfere with other batches.

Status

The Aligned Layer team has adjusted the operator, such that it does not currently trust the headers. The team is also using a streaming `http` reader, so it will only download less than the maximum amount. In addition, downloading the batch happens in parallel; hence, a slow server cannot block the operator.

Verification

Resolved.

Issue B: Attack on Aggregator Race Condition Censoring a Batch

Location

`aggregator/internal/pkg/server.go#L79-L96`

`aggregator/internal/pkg/server.go#L107-L123`

Synopsis

By exploiting a race condition in signature verification, it is possible to cause legitimate signatures to be dropped.

Impact

This issue can possibly lead to at least one-third of Operator signatures being dropped for a given batch and thus prevent that batch from being validated. The attacker can therefore censor a particular batch.

Preconditions

The attacker would need to know the BLS public keys of the Operators, the IP address of the Aggregator, and the `merkleBatchRoot` of a batch they wish to censor.

Feasibility

If the preconditions are met, the attack is trivial.

Technical Details

The Aggregator contains a race condition where BLS signatures are checked in a separate Go routine. However, there is also a `batchResponses` map to track signatures already received from a given Operator. `batchResponses[operator_id]={}` is set when a `SignedTaskResponse` is received, but before it has been fully validated. This means that an invalid `SignedTaskResponse` (STR) will still cause the `batchResponse` map to be set as if a valid STR has been received, but only briefly. If an attacker sends many invalid STRs, they could cause the `batchResponse` map to be set most of the time, so when a valid STR is received, it will likely be ignored as if it had already been received. To validate a batch, the Aggregator must receive valid STRs from two-thirds of the Operators, so if the attacker can cause one-third of the STR to be ignored, they can censor the entire batch.

The invalid STRs do not need valid signatures; they only need the current `merkleBatchRoot` and a correct `OperatorId`. To successfully block an entire batch, the attacker must prevent at least one-third of the Operator's STRs from being accepted. Hence, they must send invalid STRs from every `OperatorId`.

Mitigation

There is no way for current users or Operators to mitigate this attack.

Remediation

We recommend refraining from mutating the state until an incoming message has been fully validated. One approach would be to authenticate the Operators sending SRT in some way that is faster than a BLS signature so that it does not need to be done in a separate goroutine. It may also be advisable to throttle incoming connections to a reasonable amount for each particular IP address, thereby preventing the attack from being effectively executed using just a single or handful of servers.

Status

The Aligned Layer team has addressed this issue by ensuring that no state is modified until the signature is verified.

Verification

Resolved.

Issue C: Replay Attack on Batcher Disrupts a Particular User

Location

`batcher/aligned-batcher/src/lib.rs#L260-L314`

Synopsis

Replaying valid proof requests can cause the expected proof cost to be misestimated, thus preventing a valid proof from being accepted.

Impact

This issue can prevent a particular user from being able to request a proof.

Preconditions

The attacker would need to possess a valid proof request for the target user, which can be extracted from a batch in which they submitted a proof.

Feasibility

The attack is trivial, given that the balance of the target user is not high.

Technical Details

In the Batcher, in `handle_message`, a `ClientMessage` is received, the signature is checked (but the nonce is not checked yet), then the proof count is checked and incremented for that user. The proof count is used to estimate whether the user has enough Ethereum balance to pay for the proof. The proof count is not reset until after the batch is submitted, so any subsequent proof requests within this batch will have an incremented proof count. The proof is then validated, and finally, the nonce is checked and incremented. If the `ClientMessage` was replayed, the proof will be valid but the nonce will not be. Consequently, the message will be dropped and will not be included in the batch. However, because the proof count is still incremented, it will appear as though the user needs more balance to pay for the next proof that they submit. If the attacker submits many replays, they can eventually increase the cost estimate, such that a real proof will not be accepted.

An Aligned Layer user is expected to have a minimum balance per proof of `13,000*100 gwei = 0.0013` Ether, which, as of the writing of this report, is equivalent to `3.36` USD. Hence, if the attacker is

able to submit 30 invalid proofs (increasing the proof count to 30), the Batcher will then require the user to have more than 100 USD in their balance. Moreover, since the balance for a user is public information, the attacker can check how many invalid proofs need to be replayed.

Mitigation

If the user's Ethereum balance is very large, it may not be possible for the attacker to manipulate the proof count to make it high enough, such that the user would appear unable to afford a proof. However, since the attack is essentially free for the attacker (as it requires simply making web socket connections and sending messages), it would likely require a balance equivalent to thousands of dollars.

Remediation

We recommend checking the nonce at the start when checking the signature but refraining from incrementing it until the end, once the `ClientMessage` is fully validated. Since this attack depends on the attacker sending many requests, throttling connections and data by IP address would also have a positive impact. However, we strongly recommend refraining from mutating the state until message authentication is complete.

Status

The Aligned Layer team is currently refraining from incrementing the proof count until the message is fully validated and added to the batch.

Verification

Resolved.

Issue D: Aggregator or Operator Downtime Can Break Eventual Consistency

Location

`Aligned-Layer/tree/operator`

`Aligned-Layer/tree/aggregator`

Synopsis

Instances within any distributed system can experience downtime. This can occur for a variety of reasons, such as a denial of service (DoS) attack, bug, or software upgrades. Distributed systems should be designed to tolerate the downtime of the nodes in the system, such that the consistency of the system is not invalidated during a downtime event. We noted several parts of the `EigenLayer` system that, in the event of downtime, could break the eventual consistency of the system.

Impact

Batches that are otherwise valid could fail to pass through the whole system and would not become confirmed. A user would have to re-submit the batch, paying for the proof again.

Feasibility

It is expected that a node will experience downtime given a long enough time period, and as such, the scenarios outlined below are feasible.

Technical Details

Operators learn of new batches through subscribing to `NewBatch` events emitted by the `AlignedLayerServiceManager` contract. If an Operator experiences downtime, `NewBatch` events

emitted when the Operator was offline will not get processed by the Operator. If over one-third of the Operators experience downtime and subsequently miss a `NewBatch` event (e.g., due to a poorly rolled out upgrade), it will be impossible for the Aggregator to accumulate a quorum of verifications for the corresponding batch. This will result in `respondToTask` not being called on `AlignedLayerServiceManager`, breaking eventual consistency, despite the batch being valid.

If the Aggregator were to experience downtime, it could similarly cause batches to be skipped. If an Operator's message fails to reach the Aggregator, the Operator will continue to attempt to reach the Aggregator for `100` seconds. If the Aggregator fails to become available within this time, data loss will occur. The Operator also subscribes to `NewBatch` events emitted by the `AlignedLayerServiceManager` contract, which would be missed during downtime. These events trigger setting up the necessary data structures in the Aggregator to handle messages from the Operators. Should the `NewBatch` event be missed by the Aggregator, Operator messages for that batch will be unable to be processed, and eventual consistency will not be reached.

Similarly, the Aligned Layer Batcher could experience downtime. However, this would not break eventual consistency, as the user CLI or SDK would report an error to the user, and no data loss would occur.

Remediation

To address Operator downtime, we recommend that an Operator log timestamps to a file periodically. When the Operator is back online after downtime, it could read the file to learn when it went offline and process all the batches since the downtime that do not already have a corresponding `BatchVerified` event.

Additionally, we recommend having the Operator write to a log any batch it has previously verified and skip re-verifying those batches after downtime, which the Aggregator has already verified, but which have not yet triggered a `BatchVerified` event by the smart contract either because a quorum has not yet been reached, or because the `respondToTask` transaction is still in the mempool.

To address Aggregator downtime, we recommend solving the Operator-to-Aggregator message loss by placing a highly available queue in front of the Aggregator, such as Amazon SQS. However, Operator signatures accumulated by the Aggregator that had not yet reached a quorum would be lost in an Aggregator crash event. To remediate this, Operator signatures could be logged to a file, which could be read on service startup. However, if an Aggregator that is experiencing issues is replaced by another Aggregator, these accumulated messages would not be available, as the log file would be on a separate instance. A shared datastore, such as Redis, would be required to facilitate node replacement. For catching up on missed events from `AlignedLayerServiceManage`, the timestamp logging solution should suffice.

Status

The Aligned Layer team has acknowledged this finding. They have proposed some solutions, such as the use of a `write ahead` log (WAL) in the Aggregator where incoming Operator signatures will be written, and only deleting once the Aggregator has completed the BLS signature verification. When the Aggregator comes online again after an outage or an update, the WAL will be read from, and the Operator signatures can again be used for the Aggregator BLS signature.

Verification

Unresolved.

Issue E: Aggregator Memory Leak Could Lead to Denial of Service

Location

aggregator/internal/pkg/aggregator.go#L360-L363

Synopsis

When a new task is ingested by the Aggregator, several maps are written to, with task-specific variables. However, this data is not deleted when a task is completed. This means that the maps will grow indefinitely, using up memory.

Impact

If the service runs for long enough, it could run out of memory, causing the Aggregator to stop functioning. This would result in failed batches, as Operator signatures would not get aggregated, and `respondToTask` would not get called in the `AlignedLayerServiceManager` smart contract.

Feasibility

The data being stored in memory does not take up much space. Consequently, it would take a considerably long time to run out of memory, which makes this issue unlikely to occur.

Technical Details

These are the maps that are written to upon the creation of a new task:

```
agg.batchesIdxByRoot[batchMerkleRoot] = batchIndex

agg.batchCreatedBlockByIdx[batchIndex] = uint64(taskCreatedBlock)

agg.batchesRootByIdx[batchIndex] = batchMerkleRoot
```

Remediation

We recommend deleting batch-specific information from the Aggregator's memory once the batch has been completed. This could be done in the `sendAggregatedResponse` function.

Status

The Aligned Layer team has added logic to periodically delete values from the Aggregator memory maps that are no longer required by the system. This causes the memory to be released, and so the memory leak is no longer present.

Verification

Resolved.

Suggestions

Suggestion 1: Add More Tests

Location

All components of the system.

Synopsis

Currently, there are very few tests in the Aligned Layer system, and there are no tests at all for the Batcher or Aggregator. Having good test coverage can help with the early identification of bugs and also facilitate refactoring.

Mitigation

We recommend writing comprehensive tests for all important parts of the Aligned Layer system.

Status

While our team notes that there is some testing (for example, for the Operator), and that the testing suite has been added to, there are still no unit tests for the Batcher or Aggregator.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.