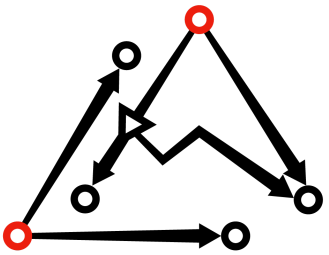

Least Authority

PRIVACY MATTERS

Avail Light Client

Security Audit Report

Polygon Matic

Initial Audit Report: 11 November 2022

Table of Contents

Overview

Background

Polygon Matic has requested that Least Authority perform a security audit of their Avail Light Client.

Project Dates

- **September 19 - November 9**: Code Review *(Completed)*
- **November 11:** Delivery of Initial Audit Report *(Completed)*
- **TBD:** Verification Review
- **TBD:** Delivery of Final Audit Report

The dates for verification and delivery of the Final Audit Report will be determined upon notification from the Polygon Matic team that the code is ready for verification.

Review Team

- Mehmet Gönen, Cryptography Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- DK, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer
- Nishit Majithia, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Avail Light Client 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:

- Avail Light Client Repository:
  https://github.com/maticnetwork/avail-light
- Avail Layer 1 Repository:
  https://github.com/maticnetwork/avail

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

- Avail Light Client Repository: `9132f5ead146ee138a336b3be055545272ce4c51`
- Avail Layer 1 Repository: `062d9467c6d17ed6e2bdea3993f670914aaf90e6`
- Updated Avail Layer 1 Repository: `3f509a3620945664a2fa3781adc4824dfaa42bef`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- Avail Light Client:
  https://github.com/LeastAuthority/Polygon-Matic-Avail-Light-Client
- Avail:
  https://github.com/LeastAuthority/Polygon-Matic-Avail
- Avail + Pallets:
  https://github.com/LeastAuthority/Avail-updated-commit

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:

- Avail repos docs: https://github.com/maticnetwork/data-availability
- Reference documentation with high level design (Whitepaper): https://github.com/maticnetwork/data-availability/blob/master/reference%20document/Data%20Availability%20-%20Reference%20Document.pdf
- Light Client readme: https://github.com/maticnetwork/avail-light

In addition, this audit report references the following documents and links:

- R. Barbulescu and S. Duquesne. "Updating Key Size Estimations for Pairings." *Journal of Cryptology,* 2019, [BD19]
- S. Bowe, "BLS12-381: New zk-SNARK Elliptic Curve Construction." 2017, [Bowe17]
- T. Pornin, A. Kircanski, M. Hemmel, D. Wong, J. Ghazizadeh, et al., "Zcash Overwinter Consensus and Sapling Cryptography Review." *NCC Group,* 2019, [TKH+20]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation, including cryptographic constructions and primitives;
- Common and case-specific implementation errors in the code;
- Networking and communication with external data;
- Performance problems or other potential impacts on performance;
- Value commitment integrity and value base integrity checks;
- Data privacy, data leaking, and information integrity;
- Resistance to DDoS (Distributed Denial of Service) and similar attacks;
- Vulnerabilities in the code leading to adversarial actions and other attacks;
- Protection against malicious attacks and other methods of exploitation;
- Performance problems or other potential impacts on performance;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a comprehensive audit of the Avail Light Client and Full Node implementations, in addition to the Solidity smart contracts that bridge the system to Nomad. The purpose of the Avail system is to reliably and consistently make layer 2 transaction data available for verification for a specific period of time. The Avail Full Node is built with Substrate and the implementation provides the full suite of services for the blockchain.

Our team investigated the design and implementation of the Avail Light Client and Full Node for potential security vulnerabilities that could impact the system or its users. We found that security has generally been taken into consideration in the design and implementation of both the Avail Light Client and Full Node. We also found continuous integration workflows that check for passing tests and proper formatting and evidence of test-driven development practices, in addition to comments about how to improve test coverage and quality in the codebases.

System Design

Full Node

The Avail Full Node, apart from its chain duties of staking, slashing, and general governance, is also capable of creating, verifying, and retrieving data from Kate commitment proofs, the atomic unit of the data availability scheme that Avail employs. The Kate commitments allow significantly smaller individual elements to prove the correctness of an entire set, creating an efficient and compact proof generation and verification scheme. Our team checked for correctness in the Kate commitment scheme design and implementation.

In our examination of the Kate commitment scheme implementation, we found that the elliptic curve used in the scheme does not provide a sufficient level of cryptographic security. We recommend that an elliptic curve with stronger security properties be used (Suggestion 3).

Our team examined the data submission flow of the Full Node. Data is submitted to an Avail Full Node and, after the transaction is finalized, the Full Node prepares a proof that the data is on chain with the transaction's response and submits it to the settlement layer. We searched for paths to abuse this process or any of the related proof or verification processes, as they underpin the security assumptions of the entire data availability layer, and did not identify any issues.

Network Layer

The Avail Full Node is configured to accept an InterPlanetary File System (IPFS) instance, which makes it a peer on the IPFS network. This allows the Full Node to receive blocks from gossip sources, which decentralizes the block protocol. However, it also makes it vulnerable to adversarial network behavior and peer messages. We examined the network for a variety of weaknesses, including distributed denial of services (DDoS) attacks, incorrect permissions or access, or other exploitable behavior, and identified an issue related to a DDoS attack abusing the Kate proof building process (Issue B).

Light Client

The Avail Light Client is a thin verification layer that tracks the tip of the chain and runs verification checks against the availability and thus correctness of the data recorded into the blocks. It continually queries for proofs and verifies them with a random sampling of 8 cells. If an IPFS instance is provided, the Light Client acts as a peer in the Distributed Hash Table (DHT), and blocks can be received by peer lookups. This decentralizes the block receipt process, providing redundancy to block discovery. Otherwise, they are simply queried from a Full Node. Subscribers are notified upon new block verification, and the verified block headers are then cached. This enables Light Clients to provide fast verification of new blocks and allows an application client to listen for these verification events.

Nomad Pallet

For additional verification and validation with an external service, the data availability layer can use a "data attestation bridge" that is anchored to the Ethereum blockchain. The scope of this audit included the Solidity smart contracts for this bridge integration. Nomad is used as a cross-chain settlement solution for operations.

The core functionality of these smart contracts is implemented in one function, `handle`, which is declared by the Nomad Bridge and must be fulfilled by any smart contract wishing to communicate across their Bridge. The `handle` function is responsible for receiving messages and parsing them as data roots.

We examined these Solidity contracts for a variety of issues, including reentrancy, arithmetic and rounding errors, and other vulnerabilities, and did not identify any issues. Bridges must accept arbitrary message payloads, which creates a potential attack vector and increases the attack surface. As a result, we recommend continued caution and regular security audits of the Solidity contracts, along with their related Bridge code, any time changes are made.

Use of Cryptography

Our team searched for issues in the cryptographic design choices, implementation, and usage and identified issues and suggestions related to the use of cryptography. We found that the Light Client generates insufficiently secure private keys. We recommend that private keys be generated according to best practice recommendations to prevent undermining the security of the cryptographic primitive (Issue C). We also identified instances of improper zeroization of sensitive inputs, which could make secret data vulnerable to compromise. We recommend using zeroization libraries appropriately (Issue A, Issue D).

We looked for incorrect uses of randomness and found that ChaChaRng is used and configured correctly to provide sufficiently secure random number generation.

Code Quality

Our team performed a manual code review of the repositories in scope and found that the coded implementation was generally well written and organized, and included comprehensive code comments. However, we found instances of unused code, which could cause confusion. We recommend that unused code be removed (Suggestion 4). We also found instances of inconsistent type usage within the implementation, which could potentially lead to issues (Suggestion 2).

In our review, we noted that some of the code is adapted from different projects (e.g., Substrate, Ethers.rs, etc.) We recommend that security developments in those projects be monitored closely to identify vulnerabilities that could affect the Avail Full Node and Light Client implementations.

Tests

The Avail Full Node has a comprehensive test suite, including a fuzz testing framework. The fuzz testing framework the Avail Full Node employs uses the afl fuzzer, a state-of-the-art fuzzer that leverages genetic mutation algorithms to create inputs for tests. Avail's fuzz testing framework is capable of parallelizing fuzz test execution, an important step towards an in-depth fuzz testing strategy. We found evidence of plans for future development of the fuzzing framework. We commend the Polygon Matic team for their due diligence and encourage them to pursue further utilization of the fuzzing framework.

In addition to fuzz testing, we found a thorough set of benchmark tests centered around the efficiency of the Kate proof creation and verification functions. Benchmark tests like these are valuable for understanding how optimizations affect the code over time. We also found a load test on the submission process that shows additional focus on performance.

The Light Client has minimal tests, as it is a thinner layer that mostly assembles libraries from the Avail Full Node and binds them to a server. We examined the tests in the Light Client and found they covered both error and happy path tests where applicable. Though the Light Client tests are minimal, the logic the Light Client utilizes is mostly imported from the Avail crate, where the tests for those libraries belong. As a result, we consider the Light Client's test coverage sufficient.

The Solidity smart contracts in scope had a full test suite that included both error paths and happy path testing, as well as a test for integration with Nomad.

Documentation

The documentation provided for this security review was sufficient. Specifically, the Avail whitepaper effectively described the Avail Full Node and Light Client theoretical underpinnings, and the architecture of the components that compose the system.

However, the current documentation does not explain the structure of the codebase. We recommend that the Polygon Matic team add a comprehensive description of the codebase, especially for cases where there are several repositories and reused components (Suggestion 5).

Scope

The scope of this review was sufficient and included all security-critical components.

Dependencies

Our team discovered some minor dependency issues (Suggestion 1) but could not identify any direct security vulnerabilities in the use of dependencies.

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: Seeds Are Not Zeroized	Reported
Issue B: Data Availability Component Is Vulnerable to DDoS Attacks	Reported
Issue C: Insecure Key Generation for IPFS Ed25519 Key in Light Client	Reported
Issue D: No Zeroization for IPFS Ed25519 Key in Light Client	Reported
Suggestion 1: Update and Maintain Dependencies	Reported
Suggestion 2: Use a Consistent Type for Cell and Position	Reported
Suggestion 3: Use the BLS12-461 Curve Instead of the Insufficiently Secure BLS12-381 Curve	Reported
Suggestion 4: Remove Unused Code and Packages	Reported
Suggestion 5: Improve Documentation	Reported

Issue A: Seeds Are Not Zeroized

Location

node/src/chain_spec.rs#L117

node/src/chain_spec.rs#L133

node/src/chain_spec.rs#L141

Synopsis

An attacker that is able to access the memory of the node (e.g., accessing core dump and exploiting vulnerabilities like in the Heartbleed attack) may be able to retrieve non-zeroized seed values and derive the private key or authorities keys.

Impact

The leakage of cryptographic keys could result in the loss of security properties, such as confidentiality and privacy.

Preconditions

An attacker must be able to read memory regions that contain sensitive data.

Mitigation

We recommend using the `zeroize` crate for zeroization.

Status

Reported.

Issue B: Data Availability Component Is Vulnerable to DDoS Attacks

Location

kate/src/com.rs#L286

rpc/kate-rpc/src/lib.rs#L29

rpc/kate-rpc/src/lib.rs#L178

Synopsis

`KateApi` contains the `query_proof` method, which accepts the `cells` parameter that is a vector (`Vec`) of `Cell`. The length of cells is neither limited nor validated and can be set to an arbitrarily large value by an attacker causing a large amount of RAM to be allocated. Moreover, the implementation of this method uses `reserve_exact`, which panics if the input value is `usize::MAX`.

Impact

An attacker with access to `KateApi` can cause a DoS attack.

Preconditions

The configuration of the system must allow an attacker to access `KateApi`.

Mitigation

We recommend making the input validation for the `cells` parameter consistent with the security and business logic needs of the implementation.

Remediation

We recommend that rate limiting be implemented for `KateApi`.

Status

Reported.

Issue C: Insecure Key Generation for IPFS Ed25519 Key in Light Client

Location

`main/src/data.rs#L116`

Synopsis

The size of the Ed25519 private key is 32 bytes. The current implementation provides a private key that is generated randomly and is 8 bytes long instead of 32 bytes. All other bytes are equal to 0.

Impact

The level of the cryptographic security of the private key is drastically decreased.

Mitigation

We recommend using 32 bytes long private keys generated with a cryptographically secure random number generator.

Status

Reported.

Issue D: No Zeroization for IPFS Ed25519 Key in Light Client

Location

`main/src/data.rs#L117`

Synopsis

The secret input bytes are not zeroized because the original wrapped function from the `libp2p` library does not zeroize them appropriately.

Impact

The leakage of cryptographic keys could result in the loss of security properties, such as confidentiality and privacy.

Preconditions

An attacker must be able to read memory regions that contain sensitive data.

Mitigation

We recommend passing a reference from `_bytes` to `keypair` in the `libp2p` library function to properly zeroize the secret data.

Status
Reported.

Suggestions

Suggestion 1: Update and Maintain Dependencies

Synopsis
The `chrono`, `rocksdb`, `time`, `owning_ref`, `lz4-sys`, and `lru` dependencies are outdated or have known vulnerabilities. A robust development process includes the regular maintenance and updates of dependencies in order to minimize the risk of exploits of known vulnerabilities within the codebase. To get the full list of the outdated or vulnerable dependencies, we suggest running the `cargo outdated` and `cargo audit` tools.

Mitigation
We recommend updating or replacing the reported vulnerable dependencies. We also recommend updating the relevant upstream package in the event that a dependency is used by an upstream dependency. In addition, we recommend regularly running the `cargo audit` and `cargo outdated` tools.

Status
Reported.

Suggestion 2: Use a Consistent Type for Cell and Position

Location
`kate/src/com.rs#L35`

`kate/recovery/src/com.rs#L510`

Synopsis
The Avail Light Client and Avail data availability components use different Rust primitive types (u32/u16) for the same notion declared as `Cell` and `Position` types. At present, this has not resulted in a vulnerability. However, it could lead to an integer overflow vulnerability or type confusion attacks.

Mitigation
We recommend using types consistently for similar objects.

Status
Reported.

Suggestion 3: Use the BLS12-461 Curve Instead of the Insufficiently Secure BLS12-381 Curve

Location
`kate/Cargo.toml#L15`

Synopsis
The BLS12-381 curve is a pairing-friendly elliptic curve and is the focus of significant development and deployment effort, especially in blockchain applications. Avail also uses the BLS12-381 curve in Kate

Commitment for pairing computation. BLS12-381 was intended to offer around a 128-bits security level. However, according to the paper [BD19], the actual security level is probably between 117-bits and 120-bits. This is still higher than the 112-bits required by NIST for new products. Detailed information about the security level of this curve can be found in the report, [TKH+20]. Furthermore, there is also an ongoing discussion about changing the bit length of the curve.

Mitigation

In such cases, it is a better option to take precautionary measures by considering that the attacks can be improved. Therefore, we recommend using the BLS12-461 curve instead of the BLS12-381 curve, as its security level is between 134-bits and 135-bits, which ultimately results in a higher security level.

Status

Reported.

Suggestion 4: Remove Unused Code and Packages

Synopsis

The codebases contain unused code and packages that make understanding and reasoning about the security of the implementation more difficult.

Mitigation

We recommend removing all unused code and packages.

Status

Reported.

Suggestion 5: Improve Documentation

Synopsis

The Avail project documentation lacks detailed descriptions of each of the codebases, including descriptions of the security-critical functions and components and how they interact with each other.

Mitigation

We recommend improving the documentation to include descriptions of the structure of each of the repositories that compose the Avail system, including detailed descriptions of security-critical functions and components.

Status

Reported.

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, and zero-knowledge 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.