Least Authority
PRIVACY MATTERS

Plonky2

Security Audit Report

Polygon Zero

Final Audit Report: 8 December 2022

Table of Contents

Overview

Background

Polygon Zero has requested that Least Authority perform a security audit of Plonky2, a library for recursive SNARKs.

Project Dates

- **July 27 - September 16:** Initial Code Review *(Completed)*
- **September 21:** Delivery of Initial Audit Report *(Completed)*
- **December 1 - 6:** Verification Review *(Completed)*
- **December 8:** Delivery of Final Audit Report *(Completed)*

Review Team

- Mehmet Gönen, Cryptography Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Plonky2 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:

- Plonky2:
  https://github.com/mir-protocol/plonky2/tree/main/plonky2
- Util:
  https://github.com/mir-protocol/plonky2/tree/main/util
- Field:
  https://github.com/mir-protocol/plonky2/tree/main/field

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

```
eae94c5a6b7199776cc1803414de44835e3e5e06
```

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

https://github.com/LeastAuthority/Polygon_Zero_Plonky2

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:

- Plonky2: Fast Recursive Arguments with PLONK and FRI:
  https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf
- Plonky2 repo:
  https://github.com/mir-protocol/plonky2

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

- A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec, "Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols." *IACR Cryptology ePrint Archive*, 2019, [AAB+19]
- T. Attema, S. Fehr, and M. Klooß, "Fiat-Shamir Transformation of Multi-Round Interactive Proofs." *IACR Cryptology ePrint Archive*, 2022, [AFK22]
- R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, 2005, [ACD+05]
- A. Bariant, C. Bouvier, G. Leurent, and L. Perrin, "Algebraic Attacks against Some Arithmetization-Oriented Primitives." *IACR Transactions on Symmetric Cryptology*, 2022, [BBL+22]
- E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast Reed-Solomon Interactive Oracle Proofs of Proximity." *Electronic Colloquium on Computational Complexity*, 2017, [BBH+17]
- E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf, "DEEP-FRI: Sampling Outside the Box Improves Soundness." *arXiv ePrint Archive*, 2019, [BGK+19]
- A. Canteaut, T. Beyne, I. Dinur, M. Eichlseder, G. Leander, et al., "Report on the Security of STARK-friendly Hash Functions (Version 2.0)." *HAL*, 2020, [CBD+20]
- A. Chiesa and E. Yogev, "Subquadratic SNARGs in the Random Oracle Model." *IACR Cryptology ePrint Archive*, 2021, [CY21]
- A. Gabizon and Z. J. Williamson, "Proposal: The Turbo-PLONK program syntax for specifying SNARK programs." *ZKProof Resources*, 2020, [GW20]
- A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge." *IACR Cryptology ePrint Archive*, 2022, [GWC22]
- L. Grassi, D. Khovratovich, R. Lüftenegger, C. Rechberger, M. Schofnegger, et al., "Reinforced Concrete: A Fast Hash Function for Verifiable Computation." *IACR Cryptology ePrint Archive*, 2021, [GKL+21]
- L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems." *IACR Cryptology ePrint Archive*, 2019, [GKR+19]
- L. Grassi, C. Rechberger, and M. Schofnegger, "Proving Resistance Against Infinitely Long Subspace Trails: How to Choose the Linear Layer." *IACR Transactions on Symmetric Cryptology*, 2021, [GRS21]
- D. Hopwood, "Selector combining." 2022, [H22]
- M. Schofnegger, "Number of Rounds Calculation for Poseidon." 2021, [S21]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Data privacy, data leaking, and information integrity;
- 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; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Plonky2 is a transparent, recursive SNARK-SDK implemented in Rust, aiming to be faster than existing alternatives and to be natively compatible with Ethereum. The Polygon Zero team has implemented several custom optimization techniques and combined PLONK, Turbo-PLONK, and the Goldilocks field, in combination with a FRI-based polynomial commitment scheme, to implement an efficient recursive verifier circuit.

Our team performed a security review of the Plonky2 design and implementation and found that the system's design is firmly based in sound research, and that the coded implementation is well organized and generally adheres to best practice. Our team found that a lack of detailed, project-specific documentation of the system, its components, and the research behind its design inhibited learning about the system and reasoning about its security efficiently. However, in our discussions with the Polygon Zero team, all our questions concerning the overall design of the system were addressed in a way that demonstrates that security has been taken into consideration in the design and implementation of the system.

System Design and Implementation

Our team performed a close review of each of the components that compose the Plonky2 system and identified areas of improvement that would increase the overall security of the system.

Fields

Our team reviewed the field traits and their implementations for the base field types and their extensions. We analyzed the different representations of field elements and all the operations defined on them (such as batch arithmetic). We analyzed the coded implementation and compared it to the underlying algorithms as defined, for example, in [ACD+05]. We identified an incorrect definition of `DTH_ROOT` in the extension field representation of the base field (Issue B). We recommend developing test cases to verify the equivalence of different representations, given random inputs.

Polynomials

We reviewed the implementation of polynomials, both in their coefficient and point-value representations. We analyzed all operations defined on those structures and reviewed the implementations of FFT and IFFT. Although we did not identify any issues in the implementation of polynomials, we recommend developing test cases to verify the equivalence of different representations, given random inputs.

Helper Functions

We reviewed the correctness of the implementation of helper functions in the `plonky2_util` crate. In part, this was done by writing understandable tests that check that the optimized versions of the code produce the same outputs as the naive. We recommend implementing test cases to verify the equivalence of different representations, given random inputs.

Gates and Gadgets

Our team reviewed the implementation of custom gates and gadgets to identify any instances where gates and gadgets add wrong or insufficient expected constraints or permutations. We found a missing check in the `le_sum` function in the `split_base` gadget (Suggestion 9). Additionally, missing documentation on realized optimizations made it time-intensive to review and verify custom gates and gadgets. Therefore, we recommend updating the relevant documentation for custom gates and gadgets (Suggestion 5).

The Prover

We analyzed the prover function and compared it against the specifications in [GWC22] and the adaptations made in Plonky2, with an emphasis on the correct implementation of the Fiat-Shamir transformation and the challenger, as explained in [AFK22]. We found that the maximal computational bound, the proof of work witness, and a universal domain separator were missing in the Fiat-Shamir construction of the protocol (Suggestion 1). In our view, it would also be beneficial for the prover to include a nonce into the initial transcript, to prevent situations where the prover algorithm stalls (Suggestion 6). Otherwise, we could not identify any issues with the Fiat-Shamir construction or the prover.

The Verifier

We analyzed the verifier as well as the recursive verifier and compared them against the specifications in [GWC22] and the adaptations made in Plonky2. We focused on missing checks and constraints but could not identify any issues. We noticed that the verifier does not enforce the exclusion of the opening points from the subgroup H. In here, the prover is encouraged to check this exclusion but without a verifier constraint, this is not enforced. Since the prover is free to disclose their witness, we did not identify any issues with this.

Hash Functions

We did not find differences in the hash function implementations and respective algorithm descriptions specified in [AAB+19] and [GKR+19]. However, we did find an issue in the hash function parameters based on the security bounds presented in the respective research papers (see Issue A).

Code Quality

Our team performed a code review of the implementation and found the codebase is very well organized and well written. The abstractions implemented are clean, enhancing the readability of the code. We found that the implementation generally adheres to best practice. However, we found the variable and function naming convention could be improved in some places. For example, the naming often makes it unclear whether the operations relate to a gate or a gate instance (Suggestion 4).

Documentation

The project documentation that describes the coded implementation of Plonky2 is insufficient. The documentation provides a shallow level of detail and assumes an advanced knowledge of PLONK, in addition to recent optimization research related to the protocol.

The documentation provided should be more detailed. In the current state, one needs to read and understand several research papers, such as [ACD+05], [BBH+17], [BGK+19], [CBD+20], [CY21], [GRS21], [GW20], and [GWC22], most of which are referenced in the documentation. To help improve the understandability, and hence the security of the system, the documentation should describe the technique in sufficient detail in order for researchers to understand the code and reference the paper and specification as a justification for soundness (Suggestion 5).

Code Comments

There are functions and components implemented in the codebase that have accurate code comments, which describe the function or the component well. However, this is not consistent within the codebase, with some important functions and components not being documented at all. We recommend that code comments be implemented consistently (Suggestion 4).

Tests

Although the implementation includes some tests, many parts of the code are under-tested or not covered by unit tests at all. Unit test coverage should be comprehensive. As such, we recommend that unit test coverage be increased (Suggestion 3). In addition, we recommend implementing test cases to verify that the optimized algorithms produce the same output as the unoptimized algorithms, given random inputs for each component (Suggestion 8).

Scope

Due to timing constraints resulting from a lack of project-specific documentation, our team was unable to perform a thorough review of the files in the security-critical `field/src/arch` folder.

Dependencies

Plonky2 relies on external dependencies that are largely common Rust crates, such as `serde`, `anyhow`, `itertools`, and `rand`. We did not identify any known vulnerable dependencies in use. However, the dependencies `serde_cbor` and `ansi_term` are unmaintained. Although this could be a source of increased security risk, there are currently no known security issues resulting from the use of these dependencies (Suggestion 2).

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: Poseidon Hash Function Needs More Rounds	Unresolved
Issue B: Wrong Definition of DTH_ROOT in FieldExtension for Base Fields	Resolved
Suggestion 1: Include All Public Knowledge into the Fiat-Shamir Heuristic	Resolved
Suggestion 2: Avoid Use of Unmaintained Dependencies	Resolved
Suggestion 3: Increase Unit Test Coverage	Unresolved
Suggestion 4: Increase Code Comments	Partially Resolved
Suggestion 5: Update Documentation	Partially Resolved
Suggestion 6: Include Nonce in Computation of Proof	Resolved
Suggestion 7: Use Non Standard Type Cast from BigInt to Degree 2 Extension Fields	Resolved
Suggestion 8: Test Optimizations Using Property-Based Testing	Resolved
Suggestion 9: Perform Checks in le_sum to Avoid Overflowing a Field Element	Resolved

Issue A: Poseidon Hash Function Needs More Rounds

Location

src/hash/poseidon.rs

Synopsis

In Plonky2, the Poseidon hash function is implemented with 8 full rounds and 22 partial rounds, resulting in a total of 118 S-boxes for the 128-bit security margin. According to best practices and guidelines as noted in [S21], this number of rounds for Poseidon is sufficient for 128-bit security. However, it is possible to reduce this security level by applying an interpolation attack, which is defined in [BBL+22]. In section 4.3 of this research paper, the authors conduct a complexity analysis for this attack by assuming `r` is the total number of rounds for Poseidon and `t` is the degree of S-Boxes. In this case, the security level for the Poseidon hash function is equal to d`*log(`d`)*(log(`d`) + log(`p`))*log(log(`d`)))` where `d=t^(r-2)`, which is the degree of the univariate polynomial, and p is the field size. In Plonky2, `t=7`, `r=30`. Hence, the complexity is approximately equal to `2^95`. This is not enough for 128-bit security.

Impact

This attack detailed in the research would reduce the expected security of the Poseidon hash function and, by extension, the overall security of the system.

Remediation

To achieve a higher security level, we recommend that the Polygon Zero team either increase the total number of rounds for Poseidon, or the degree of S-Boxes. According to the calculations in [BBL+22], Poseidon should have at least 42 rounds for the 128-bit security level.

Status

The Polygon Zero team stated that they target 100-bits of security and therefore consider 95-bits as reasonably secure. Their main problem with resolving the issue is that increasing rounds would result in nontrivial changes to the protocol, as it would affect the arity of the circuit. Since different teams have different security estimations, the Polygon Zero team prefers to do more analysis in order to get a better understanding of which estimate is the most appropriate.

Moreover, the Polygon Zero team said that since the authors of the reinforced concrete hash function in the paper [GKL+21] are working on a variant of the hash function that supports the Goldilocks field, they might switch to this hash function in the future.

To increase transparency for users, the Polygon Zero team added a brief note into their README in order to highlight the 95-bits of security as a potential risk.

Verification

Unresolved.

Issue B: Wrong Definition of DTH_ROOT in FieldExtension for Base Fields

Location

src/extension/mod.rs#L84

Synopsis

Implementations of the `FieldExtension` trait require the definition of a public constant `DTH_ROOT`, such that `DTH_ROOT^D=1` (hence `DTH_ROOT` is a d-th root of unity) for the extension degree `D`. However, in the case of `D=1`, the constant `DTH_ROOT` is incorrectly defined as `DTH_ROOT=0`.

Impact

The `DTH_ROOT` of extension degree 1 is used to incorrectly represent `DTH_ROOT` in extension degrees 2, 3 and 5 (see here, here, and here). Hence, all generic representations of the `FieldExtension` trait use a wrong definition of `DTH_ROOT`, which might result in incorrect computations.

Remediation

We recommend that the Polygon Zero team define `DTH_ROOT = 1`.

Status

The Polygon Zero team implemented the recommended remediation.

Verification

Resolved.

Suggestions

Suggestion 1: Include All Public Knowledge into the Fiat-Shamir Heuristic

Location

`src/plonk/circuit_builder.rs#L793`

Synopsis

Plonky2 uses the Fiat-Shamir heuristic to transform a public-coin interactive proof into a non-interactive random oracle proof. The general idea is to compute the i-th challenge as a hash of the i-th prover message and (some part of) the previous communication transcript (See 2.5 in [AFK22]). Applying this rule rigorously, we identified that the following information is missing as input to the challenger:

1. The maximum computational bound n is initial public knowledge, but not part of the initial transcript. In classic KaTe Plonk [GWC22], n is part of the Common Reference String (at least implicitly by the size of the vector `([x]_1, [x^2]_1,...,[x^n]_1)`. However, in Plonky2, the maximal computational bound is not part of the initial transcript.
2. The proof of work witness is part of a prover message in the interactive protocol but is not used as input to the challenger in the Fiat-Shamir heuristic.
3. The initial transcript lacks a universal domain separator, which enables the potential for proofs of identical circuits to be reused between different proof systems.

Note that we could not identify specific attacks resulting from the missing inputs and therefore cannot estimate the likelihood that this would result in security-critical vulnerabilities or issues.

Mitigation

We recommend that the Polygon Zero team include the maximum computational bound and the proof of work witness into the challengers state and provide a customizable domain separator for system users in the initial transcript.

Status
The Polygon Zero team included the circuit size, the proof of work witness, and a universal domain separator into the initial transcript of the Fiat-Shamir heuristic.

Verification
Resolved.

Suggestion 2: Avoid Use of Unmaintained Dependencies

Synopsis
Plonky2 makes use of dependencies that are unmaintained (See RUSTSEC-2021-0127, RUSTSEC-2021-0139), which is not a recommended practice. Unmaintained dependencies are not updated with security patches and can increase the attack surface area.

Mitigation
We recommend that the Polygon Zero team keep dependencies up to date and use libraries that are well maintained. For example, newer versions of `clap` (i.e., version 3 and higher) do not depend on `ansi_term`.

Status
The Polygon Zero team updated their version of `clap` and removed the `ansi_term` dependency. However, the team pointed out that these dependencies are `development` dependencies, not `build` dependencies, and hence pose no security risk, which our team agrees with.

Verification
Resolved.

Suggestion 3: Increase Unit Test Coverage

Synopsis
There are very few tests implemented that check the correctness of the implementation and that the components that compose the system behave as expected. Many areas remain untested, as most of the tests that exist are end-to-end tests, not unit tests.

Mitigation
We recommend that the Polygon Zero team implement additional unit tests.

Status
The Polygon Zero team agrees that test coverage should be improved, but noted that they are currently unable to make much progress within the timeframe of the audit.

Verification
Unresolved.

Suggestion 4: Increase Code Comments

Location
`field/src/goldilocks_extensions.rs#L115`

src/plonk/circuit_data.rs#L272

src/polynomial/mod.rs#L119

Synopsis

The index conventions in the code deviate from those of [GWC22]. In fact, in [GWC22], the authors represent polynomials in their point-value representation as `[(g,P_1),(g^2,P_2),...,(g^n,P_n)]` under the assumption that `g^n=id_G`, while the Plonky2 code uses the convention `[(1,P_0),(g,P_1),...,(g^(n-1),P_(n-1))]`. Although both representations are equivalent, pointing this out in the comments makes it easier to compare this code against the reference [GWC22]. With regard to this index scheme, some variable names, as in here (where `L_1` should be called `L_0` conceptually), should be updated accordingly.

Moreover, in places like this, the readability of the code would be greatly improved by explaining which values the variables `oracle_index` and `polynomial_index` are expected to take on.

Mitigation

We recommend that the Polygon Zero team make the coverage of the code comments comprehensive (see examples `field/src/cosets.rs#L15`, `field/src/types.rs#L101`). In addition, some comments are misleading or incomplete. Therefore, we further recommend including a detailed description of highly optimized functions, ideally starting with a set of preconditions and ending with a set of postconditions, along with a link that references the abstract algorithms in a research paper.

Status

The Polygon Zero team accepted our suggestion regarding the different formats in which polynomials are represented in the code and in [GWC22], and updated the associated code comments accordingly. However due to time constraints, they were not able to improve all the code comments at the time of our verification.

Verification

Partially Resolved.

Suggestion 5: Update Documentation

Location

Plonky2

Synopsis

The existing project documentation is difficult to read since it describes updates to, and merges of, several existing cryptographic protocols without presenting the theoretical foundation to those updates or merges. Nor does the documentation provide justifications regarding how the changes and merges of the used cryptographic protocols relate to the soundness analysis of the individual used protocols and related building blocks. Additionally, a project specification documenting, for example, the details of used parameters, is missing.

Mitigation

We recommend that the Polygon Zero team improve the documentation by adding:

1. A theoretical description of the new protocol Plonky2 that not only references the protocol's building blocks or the parts described in other work, like [GWC22] or [H22], but that is also complete and readable on its own account. More specifically, auditors must be able to perform

and describe a soundness analysis based on this theoretical description of the entire Plonky2 protocol.

2. A project specification including:
    a. Descriptions of the information flow within the system, in addition to a clear explanation regarding which functions are called (especially for custom gates and gadgets);
    b. Descriptions of custom gates and gadgets;
    c. Descriptions of how constraints are combined through the chaining argument;
    d. Descriptions of used parameters; and
    e. A soundness analysis performed as a function of soundness relevant parameters as `rate`, `num_challenges`, etc.

Status

The Polygon Zero team partially accepted our suggestion by updating the `Plonky2` paper. In particular, the team included a section to describe all the steps in the protocol (section 7 in `Plonky2`), which helps readers and maintainers to understand the system. What is still lacking, though, from an auditor's perspective is a complete Plonk analysis that is stand-alone and walks the reader through all the steps of the reasoning without referencing related work, as much as possible.

The Polygon Zero team also stated that since batched FRI analysis is a deep topic, and not specific to Plonky2, it is best to leave FRI analysis to external research. While our team partially agrees with this, we also want to point out that, from a user's perspective, a method that explains the dependency of the system's security from its parameters is desired.

Verification

Partially Resolved.

Suggestion 6: Include Nonce in Computation of Proof

Location

`src/plonk/circuit_builder.rs#L793`

Synopsis

In case zero-knowledge is disabled, generating the Z and `Pi` polynomials depends deterministically on the witness and the initial transcript. However, on rare occasions, a proof might not be generatable, as, for example, this might fail. In such a case, the system is stuck unrecoverably.

Mitigation

We recommend that the Polygon Zero team include a nonce into the initial transcript, so the prover can change it in case the algorithm fails for the reasons described above. We further recommend publishing the nonce as part of the public input.

Status

The Polygon Zero team accepted our suggestion and resolved it by randomizing some unused witness elements. In the rare case that a proof fails as described above, any following attempt will have an independent chance of success.

Verification

Resolved.

Suggestion 7: Use Standard Type Cast from BigInt to Degree 2 Extension Fields

Location

`src/extension/quadratic.rs#L93`

Synopsis

The function transforms a number of type `BigInt` into an element of a quadratic extension of some base field by dividing the number by the field order and then using the dividend and the remainder as the two elements in the two dimensional vector of the extension field element. While it is possible to map `BigInts` onto quadratic extension field elements this way, the transformation appears unnatural.

Mitigation

We recommend that the Polygon Zero team treat `BigInts` as representatives of base field elements.

Status

The Polygon Zero team replaced the function `from_biguint` with the function `from_noncanonical_biguint`, which represents a `BigUInt` as a base field element, interpreted as an element of the extension field by the canonical inclusion.

Verification

Resolved.

Suggestion 8: Test Optimizations Using Property-Based Testing

Location

`plonky2/src/arch/x86_64`

`field/src/fft.rs#L97-L161`

`field/src/packed.rs`

`util/src/lib.rs`

Synopsis

There are multiple problems that can be solved in both a straightforward and, at least one, optimized way. Specifically, the FFT and packed field elements have a separate implementation for CPUs with SIMD and AVX2/512 support, and there are optimized implementations of bit reversing functions. Due to making use of special functions, this code is very difficult to read.

Besides testing edge cases in unit tests (Suggestion 3), we also recommend testing that for random inputs, the optimized algorithms give the same results as the straightforward implementations. This provides a probabilistic argument that the code behaves in the same way as the better-auditable, yet unoptimized, algorithm.

Mitigation

For a few random input values, we recommend that the Polygon Zero team test whether the optimized and unoptimized algorithms produce the same outputs. We provide such tests for the `util` subcrate here.

Status

The Polygon Zero team added bit-reverse tests. For FFTs, the team implemented a test that compares against a native (quadratic time, scalar, pure Rust) implementation.

Verification

Resolved.

Suggestion 9: Perform Checks in le_sum to Avoid Overflowing a Field Element

Location

[src/gadgets/split_base.rs#L31-L72]

Synopsis

The function `le_sum` uses the `BaseSumGate` to combine multiple limbs into a single field element. These limbs are usually binary. The function does not check whether the result overflows a field element. It may therefore produce non-canonical field elements if the sum is in the range `ord(F)..2^64` or generate wrong results if the sum exceeds 2^64.

We did not find any such issues in the current codebase, but it is likely that such code is added later by Plonky2 users.

Mitigation

We recommend that the Polygon Zero team perform checks in `le_sum` to test that the sum is less than `ord(F)`.

Status

The Polygon Zero team implemented a check in order to assert that a field element cannot overflow.

Verification

Resolved.

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.