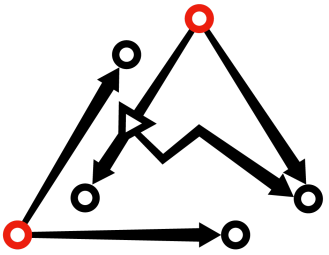

Least Authority

PRIVACY MATTERS

Multi-Asset Shielded Pool

Security Audit Report

Metastate AG

Audit Funded by Tezos Foundation

Final Report Version: 18 August 2020

Table of Contents

Overview

Background

The Tezos Foundation has requested that Least Authority perform a security audit of the Metastate Extension of the Electric Coin Company's (ECC) Sapling circuit, a user-defined asset extension that allows the shielded pool to support many denominations at once.

Project Dates

- **July 6 - July 24**: Code review *(Completed)*
- **July 29**: Delivery of Initial Audit Report *(Completed)*
- **August 12 - 14:** Verification *(Completed)*
- **August 18:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Sapling circuit extension followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

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

- Rust-language assets for Zcash: https://github.com/metastatedev/librustzcash
    - Sapling Circuit Extension: https://github.com/metastatedev/librustzcash/pull/3

However, third party vendor code is considered out of scope.

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

`6f216035e7b93db3272285677d69328e3ca7cd36`

For the verification, we examined the Git revision:

`88332e2be768026ae9339d4d51b15e17aee4d0c5`

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

Supporting Documentation

The following documentation was available to the review team:

- Forum post ("User Defined Asset Extensions for Sapling"): https://forum.zcashcommunity.com/t/user-defined-asset-extensions-for-sapling/36360
- Sapling Specification changes document: https://github.com/metastatedev/crypto-research/blob/master/masp/spec.pdf

- Zcash Protocol Specification Version 2020.1.11 [Overwinter+Sapling]: https://github.com/zcash/zips/blob/master/protocol/sapling.pdf

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation based on the changes made to the specification;
- Common and case-specific implementation errors;
- Performance problems or other potential impacts on performance;
- Changes made to the Spend circuit and Output circuit;
- Value commitment integrity and value base integrity checks;
- Data privacy, data leaking, and information integrity;
- Resistance to DDoS and similar attacks;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Review Scope

In assessing the circuit extension, our team closely analyzed the changes made to the original Sapling implementation in order to support user defined assets and many denominations at once. These changes included the introduction of the dynamic value commitment bases to represent asset types, which is a novel idea introduced by the Zcash community. Metastate's updates, in collaboration with Zcash commits, will be the first production implementation. As a result, a careful review of the changes is necessary in order to assess and determine the potential security implications.

According to Metastate's Sapling Specification changes document and our observations of the original Zcash Protocol Specification [Overwinter+Sapling], the security of these multiple asset value commitments relies on assumptions underlying the security of the homomorphic Pedersen commitments and Pedersen hashes, in addition to the security of the BLAKE2s hash function, which are similar to the security assumptions of the original Sapling protocol.

We examined the introduction of multiple asset types in confidential transactions and their impact on security, with a particular emphasis and focus on the changes to the Spend and the Output circuit. Errors in this area of the code are very difficult to correct due to the need for a trusted setup ceremony prior to deploying the code to production. Thus, reviewing circuits requiring trusted setup zk-SNARKS is particularly important prior to launch.

Code Quality + Documentation

Our team found the code to be well organized and identified no major errors in the implemented changes, with the exception of a few minor coding problems such as inadvertently mixing up the definition of two constants and clearing a cofactor (Issue A; Issue B). We also found the code comment coverage to be sufficient and comments are present in more complex positions of the code (i.e. computation), which made it particularly helpful in understanding the methodology and reasoning behind the Metastate team's approach to the coded implementation.

While test coverage from the original source code was present, some tests do not pass due to not being comprehensively adapted for the extension. As a result, some tests are passing successfully while others fail or do not compile due to requiring an update in order to be compatible with the asset type upgrades (Suggestion 5).
The existing project documentation available to our team was helpful, including posts in the Zcash community forum and the Sapling Specification changes document which we found to be particularly helpful and recommend that it be published and integrated into the project documentation, including the code comments and the README. We also suggest that Metastate's Sapling Specification changes document should better specify the security definition of the interconnection of the asset types name, identifier and generator. Furthermore, the general security model should be improved (Suggestion 4).

We commend the Metastate team for their diligence in considering the potential security implications of the implemented changes to the Sapling circuit in their extension. Furthermore, by taking a minimalist approach in making changes to the code base and only changing as much as is necessary, the Metastate team has reduced the potential introduction of new security risks and attack vectors. We also appreciate that the team was helpful and responsive in answering our question as we worked through our review and analysis.

Investigation + Analysis

We analyzed the mathematical assumption that different asset types can be represented by different generators in the Jubjub curve, as long as no discrete log relation is known between them and the `randomness_base` generator. Without any such discrete log relation, deviation from the intended behavior (binding, hiding, and non-exchangeability) would be as difficult as breaking the binding/hiding property of the general Pederson commitment scheme over Jubjub. This implies that it is possible to hide the `asset_type` generator (hence the `asset_type`) in a transaction.

On the implementation level, the absence of any discrete log relation needs to be enforced. The Metastate team achieved this by allowing only `asset_type` generators that are provably derived as BLAKE2s images of `asset_type` identifiers and this relation is checked in zero knowledge in the Output circuit. As a result, constructing new `asset_type` generators from previously known `asset_type` generators or the `randomness_base` generator in any manner that gives the attacker a discrete log relation is as difficult as finding BLAKE2s preimages or breaking the hardness of ECDL over Jubjub. We also verified that the known preimage (`GH_FIRST_BLOCK||"r"`) of the `randomness_base` generator can not be used as an `asset_type` identifier, since it is not of 32 byte size.

In comparison to the original Sapling implementation, the extension of the Output circuit introduces a considerable number of new constraints (31205 constraints in the new circuit vs. 7827 in the original one). However, our team recognizes this as a necessity in order to prove in zero knowledge, without revealing the `asset_type`, that the `asset_type` generator is derived as a BLAKE2s image of the `asset_type` identifier. This is absolutely necessary since knowledge of an `asset_type` generator preimage under BLAKE2s proves that the generator is a pseudo random point on the curve, which makes knowledge of a discrete log relation to another `asset_type` generator or the `randomness_base` improbable.

We also tested the dependencies for known security vulnerabilities against the RustSec Advisory Database and did not identify any issues. Furthermore, our team observed that version 1.0.2. of the crate `quote` has been yanked, however, it is unclear whether this has potential security implications and we suggest it be further investigated.

System Design

It is clear that the Metastate team has thoroughly considered security and taken the necessary precautions in order to limit the introduction of vulnerabilities in their approach to the design. While dynamic value commitment bases to represent asset types are novel, the Metastate team has not introduced new and unnecessary functionality and has utilized preexisting, well established, and previously audited gadgets, cryptographic primitives, and implementations.

Nevertheless, the process used for computing the generator for each asset is an ad hoc design and the construction does not run in constant time. While it is unlikely that applications provide a timing side-channel to an attacker, using constant-time algorithms rules out the whole class of attacks based on this kind of side-channel. Therefore, it is worthwhile to use a mechanism based on a well-understood construction, such as Elligator 2. Additionally, using a standard process means that it will already be implemented in a number of languages, moving the attack surface to a shared dependency. Thus, should an attacker find a vulnerability, it is likely that there are more valuable targets than the Sapling extension. Unfortunately, no procedure to hash to points on elliptic curves has been standardised at the time of writing this report, however, it is worth noting that an RFC is currently underway (Suggestion 1). We acknowledge that the Metastate team has chosen this approach to circumvent the potential pitfalls of implementing a new gadget that would require an extensive security evaluation and encourage them to monitor the progress of the RFC.

Areas for Further Considerations

In contrast to the original Sapling implementation, the asset type circuit extension is used for meta tokens in Tezos smart contracts. Unlike the Zcash cryptocurrency for which the Sapling circuit was initially designed, Tezos meta tokens are not required to have intrinsic value and are therefore not necessarily scarce. This implies that one can think of confidential transactions with more or less arbitrarily large amounts of those tokens, including zero amount transactions. Hence, bound assumptions made in the original Sapling implementation may not always apply in this new setting and should be further evaluated as an area of risk.

Furthermore, the original Sapling implementation assumes that any transferred value can be expressed as a signed 64 bit integer and that the maximum token supply is bounded by `MAX_MONEY`, which is 2.1*10^15. This is not altered in Metastate's implementation, which implies that the maximum token supply of each asset type also has to be bounded by `MAX_VALUE`. This is considerably different from the maximum token supply of asset types in other Tezos smart contracts, which are typically represented by the unbounded Michelsons types `int` or `nat`. Due to the necessary boundedness of the Sapling circuit, unbounded amounts of coins are not possible. As a result, implementations must ensure to cast Michelson `int` or `nat` safely into Sapling's `Amount` type or to implement a different approach that accounts for that discrepancy. At a minimum, users and developers of confidential `multi_asset` type contracts must be made aware of those situations.

Although our team evaluated some scenarios for problems that might arise from potentially unbounded supplies of meta tokens, we did not identify any security issues. The function `compute_value_balance` assumes that its parameter "value" is not `-i64::MAX`. Furthermore, since values are restricted to `i64::MAX`, certain theoretical problems (e.g. sending an amount of coins that is equal to Jubjub's large prime order, effectively multiplying with the large cofactor, or sending an amount of coins larger than the Jubjub or the base field order), can not be executed in the implementation. As a result, we do not consider this to be a security issue, at this point. Nonetheless, we encourage the Metastate team and future developers utilizing the extension to further consider the impacts of potentially unbounded token supplies in a circuit originally designed for a bounded token.

Specific Issues

We list the issues we found in the code 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: Personalization of any asset_type Generator and the randomness_base Generator is Equal	Resolved
Issue B: Cofactor Not Cleared in Value Commitment Calculation	Resolved
Suggestion 1: Monitor Progress of and Consider Implementing a Gadget for the Hash to Curve RFC	Resolved
Suggestion 2: Remove Unnecessary Fixed Base Generator	Resolved
Suggestion 3: Provide Documentation on Asset Type Identifier Hashing and Equality Constraints	Unresolved
Suggestion 4: Consider More Detailed Definition of Security Model of Asset Type and Adjusted Value Commitments	Unresolved
Suggestion 5: Update Failing and Non-Compiling Tests With Asset Type Upgrades	Unresolved

Issue A: Personalization of any `asset_type` Generator and the `randomness_base` Generator is Equal

Location

Definition of `fixed_base_generators[FixedGenerators::ValueCommitmentValue]` and `fixed_base_generators[FixedGenerators::ValueCommitmentRandomness]` in file zcash_primitives/src/jubjub/mod.rs

Synopsis

Since it must be guaranteed that no `asset_type` generator has a discrete log relation to the `randomness_base` generator, the implementation must ensure that the derivation of the `asset_type` generator from the `asset_type` identifier is different then the derivation of the `randomness_base` generator from its group hash preimage. Otherwise, an attacker might be able to use the `randomness_base` generator itself as an `asset_type` generator.

The Metastate team has stated that they are aware of this and have correctly accounted for it by introducing the two different personalisations `VALUE_COMMITMENT_GENERATOR_PERSONALIZATION` and `VALUE_COMMITMENT_RANDOMNESS_PERSONALIZATION` for domain separation between the `asset_type` generators and the `randomness_base` generator. However, due to a small coding error, the value `fixed_base_generators[FixedGenerators::Value CommitmentValue]` was changed, such that it uses the wrong constant, while the value

`fixed_base_generators[FixedGenerators::ValueCommitmentRandomness]` remains unchanged.

Impact

If an attacker is able to compute a 32 byte preimage `A` of the `randomness_base` generator `R`, then (A,R) serves as a valid `asset_type` (identifier, generator)-pair with a known discrete log relation (the trivial one) to the `randomness_base`. This would break the binding property of the homomorphic Pederson commitment.

Feasibility

The feasibility of the attack depends on the way in which the `randomness_base` generator is computed. It is not of any practical concern, as long as the obvious preimage of the `randomness_base` generator is not of size 32 byte (as explained in the Technical Details section), or if different personalizations are used.

Technical Details

According to function `group_hash in zcash_primitives/src/group_hash.rs`, the randomness_base generator is computed as:

Params::new()
.hash_length(32)
.personal(VALUE_COMMITMENT_GENERATOR_PERSONALIZATION)
.to_state().
.update(GH_FIRST_BLOCK)
.update(b“r”)
.finalize()

Where GH_FIRST_BLOCK is 64byte and “r” is one byte. If we assume that the first BLAKE2s hash happens to be a point on the Jubjub curve, then an asset_type generator is computed from its associated asset_type identifier as:

Blake2sParams::new()
.hash_length(32)
.personal(VALUE_COMMITMENT_GENERATOR_PERSONALIZATION)
.to_state().
.update(identifier)
.finalize()

From this follows that, since `hash_state.update(A).update(B)` and `hash_state.update(A||B)` result in the same hash, the concatenated string `identifier:=(GH_FIRST_BLOCK||b”r”)` is a valid preimage of the randomness_base generator. However, that string is of size 65 byte, which violates the assumption that an identifier must be of size 32 byte. Since the size of the identifier is checked in the circuit, any identifier of size > 32 byte is invalid.

Remediation

Use the personalization `VALUE_COMMITMENT_RANDOMNESS_PERSONALIZATION` in the definition of `fixed_base_generators[FixedGenerators::ValueCommitmentRandomness]`.

Status

VALUE_COMMITMENT_RANDOMNESS_PERSONALIZATION is now used in the definition of fixed_base_generators[FixedGenerators::ValueCommitmentRandomness].

In addition, following the audit, all personalizations used in the circuits were changed to support domain separation from the original Sapling protocol's personalizations. These changes have not been audited by our team and we suggest they be reviewed in a future audit.

Verification

Resolved.

Issue B: Cofactor Not Cleared in Value Commitment Calculation

Location

https://github.com/joebebel/librustzcash/blob/joe/zcash_proofs/src/sapling/mod.rs#L35:L37

Synopsis

To compute the value balance in order to include it in the circuit, the value is computed in the exponent of the value commitment generator. However, the cofactor is not cleared in function `compute_value_balance`, which is necessary in this construction using the Jubjub curve.

Impact

By not having the cofactor cleared, it is possible that the system is vulnerable to a small subgroup attack, which might leak three bits of information. Additionally, this leads to inconsistency in the codebase, as well as inconsistency with the Sapling implementation, which might result in the inability to generate correct proofs.

Technical Details

Due to the relation of the project to the Zcash Sapling circuit implementation and the use of the Jubjub curve, it is helpful to also clear the cofactors of the generators created.

In modern cryptography, there exists a gap between provable security and practical cryptography. Within cryptographic protocols and their implementations, it is often required to operate with elliptic curves of prime order. In practice, used elliptic curves are not of prime order but have a low-order subgroup and a high-order subgroup and the order of the whole group being of the form of h*p with p a large prime and h a small integer. The integer h is often called the cofactor. This can lead to a small subgroup attack as presented by Lim and Lee.

In such groups, G with subgroup H of prime order p and order of G being h*p with h being a small non-prime (i.e., h=r*s with small prime r), there could exist an element P such that the order of P is r*k with integer k. In <P>, it would then be possible to solve the discrete logarithm problem. To make sure the generator used in this case does not fall into <P> but rather into H, it is recommended to clear the cofactor, meaning scalar multiplying h*P so that all resulting possible elements are actually cleared of having the risk of small order.

This problem can also be identified for the Jubjub curve. The order of the Jubjub curve is 8*p with prime p, as a result, it is necessary to investigate the components for the generator in Jubjub and clear the generator used in the protocol for computations of the value commitment.

Remediation

Clear the cofactor in function `compute_value_balance` for `value_commitment_generator` through scalar multiplication with the cofactor.

It is also possible, for simplification, to have a potential `value_commitment_generator_uncleared` function without having the cofactor cleared and a `value_commitment_generator_cleared` function

with having the cofactor cleared in zcash_primitives/src/primitives.rs to clarify the usage instead of clearing the cofactor manually where needed.

Status

The value balance calculation in zcash_proofs/src/sapling/mod.rs now clears the cofactor of the value commitment generator.

Furthermore, in order to prevent errors in the future, the Metastate team is considering a type system for uncleared and cleared generators, in addition to clearing cofactors as soon as possible, consistent documentation of uncleared and cleared generators in the code, and changes to a consistent notation of asset generator and value balance generator in documentation and code. However, these changes have not been implemented.

Verification

Resolved.

Suggestions

Suggestion 1: Monitor Progress of and Consider Implementing a Gadget for the Hash to Curve RFC

Location

https://github.com/metastatedev/librustzcash/blob/6f216035e7b93db3272285677d69328e3ca7cd36/zcash_primitives/src/primitives.rs#L24-L68
https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/

Synopsis

At the time of writing this report, an ad hoc and iterative approach is taken to hash the asset identifier to a group element (i.e. the `asset_type` generator). The advantage is that it is relatively simple and exists as a well audited gadget. At the same time, an RFC is currently being developed for this kind of use case. Implementing the algorithms proposed as a gadget has the benefits of having to design less custom security critical system parts. Furthermore, standardised protocols and algorithms have often been implemented in several languages by others, such that there is high potential for reusing existing code. However, at this point in time, the RFC is still in draft form and no action needs to be taken.

Status

The Metastate team has considered adopting the techniques described in the RFC and have decided against doing so as they have noted that using a constant-time algorithm is not important in this use case, while maintaining a low complexity of the circuit is a priority. Given that the function needs to be implemented in the circuit, our team acknowledges the tradeoff between complexity and using standardized algorithms and agrees that using constant-time algorithms is not critical in this part of the system.

Verification

Resolved.

Suggestion 2: Remove Unnecessary Fixed Base Generator

Location

https://github.com/joebebel/librustzcash/blob/joe/zcash_primitives/src/jubjub/mod.rs#L313

Synopsis

The fixed `value_base` generator is present in the constants list, which will later be overridden by a supplied generator when used in the circuits. This constant declaration was used when a single asset type was present in the shielded transaction. Now that user defined bases will be supplied by the `asset_type` generator parameter in the circuit exposing the Pedersen commitment in shielded transactions, this constant declaration is no longer necessary.

Mitigation

Remove this constant in favor of a dynamic `asset_type` generator.

Status

The unnecessary constant has been removed. In addition, the remaining fixed base generators have been renumbered. However, the unintended side effects of the renumbering have not been audited as they are out of scope for this review.

Verification

Resolved.

Suggestion 3: Provide Documentation on Asset Type Identifier Hashing and Equality Constraints

Location

Sapling Specification changes document

Synopsis

As noted previously, the Output circuit must perform the final BLAKE2s hash on the `asset_type` identifier to compute the `asset_type` generator in order to ensure that the `asset_type` generator used in the Spend circuit is the same one used in the Output. The documentation notes that this is meant to prevent the case that an attack could supply the negation of the `asset_type` generator in a note commitment. The operation of negation on elliptic curves is simply reflecting the point over the y-axis or negating the y-coordinate point of the (x,y) and with Jubjub curve pair, this sign is on the first coordinate of the pair. With only this requirement, one could imagine limiting the valid identifier space even more by adding a fourth condition that all x-values are positive in the resulting identifier sample. Knowing that all identifiers must hash to `asset_type` generators with positive values could replace the BLAKE2s hash with a simple 1 bit sign check on the first coordinate to ensure negations cannot be present in circuit commitments preventing tokens from being printed unexpectedly. Theoretically, this should double the expected number of Group Hash iterations to four.

However, this would not be enough to ensure that the `asset_type` generator used in the value commitment is exactly the generator used between both the Spend and Output circuits and the notes `asset_type` generators. The circuit must enforce that the `asset_type` generator is derived by being passed through a random oracle such that it is difficult to create generators with a discrete log relationship of any two `asset_type` identifiers.

Mitigation

Provide additional documentation on the hash and equality check, in addition to information on why it is necessary to prevent discrete log relationships beyond the trivial negation case and how hashing provides a guarantee that this is not possible. This would help to add more clarity to this crucial protocol choice.

Status

The Metastate team has acknowledged that the mentioned description and motivation of implementation details should be added to the documentation and that additional source code comments should be added. They have noted that they intend on making these changes in the future.

Verification

Unresolved.

Suggestion 4: Consider More Detailed Definition of Security Model of Asset Type and Adjusted Value Commitments

Location

0.2 Asset Types: Notation and Nomenclature and *0.5 Security* in the Sapling Specification changes document

Synopsis

According to *Section 0.2 Asset Types: Notation and Nomenclature* of the Sapling Specification changes document, asset types are represented in three different ways, being either the asset name, the asset identifier, or the asset generator. These representations and their connections to each other are explained thoroughly, however, as the asset types are being used within shielded transactions, their confidentiality in different representations also needs to be assessed. The security definitions of each representation could be defined more clearly and in detail in the Sapling Specification changes document.

According to *Section 0.5 Security* of the Sapling Specification changes document, the value commitments for all asset types should be next to value hiding and non-forgeable as in Sapling and also be asset hiding and non-exchangeable. These properties are defined in an informal but coherent way. For complex cryptographic protocols, it is helpful to provide formal security definitions through, for example, a game-based or simulation-based security definition. It is recommended to define these security properties in a more formal cryptographic way in order to be able to verify the protocol through a cryptographic security proof, if and when necessary.

Mitigation

Consider adding a formal security definition in terms of provable security for asset types, discussing the asset name, asset identifier, and asset generator, and for the adjusted value commitments.

Status

The Metastate team has responded that a proper formalization of the definitions and model would be a valuable addition, but a potentially open-ended contribution. Furthermore, the documentation of the Metastate Sapling Specification changes document only focuses on the difference from the original Sapling implementation.

Verification

Unresolved.

Suggestion 5: Update Failing and Non-Compiling Tests With Asset Type Upgrades

Location

https://github.com/joebebel/librustzcash/blob/joe/
https://github.com/joebebel/librustzcash/blob/joe/zcash_primitives/src/note_encryption.rs#L1107
https://github.com/joebebel/librustzcash/blob/joe/librustzcash/src/tests/notes.rs#L651-L670

Synopsis

While running the tests on the upgrades made in the branch provided, we found a large number of failing assertions and other tests that failed to compile. The failing tests are due to the addition of the asset generator to the note which has affected other tests throughout the system.

For example,
`note_encryption::tests::compact_decryption_with_incorrect_diversifier` now fails.

Also, some compilation errors were a result from not enough arguments supplied for the function `librustzcash_sapling_compute_cm` in librustzcash/librustzcash/.

Mitigation

Correct the failing and non-compiling librustzcash tests that have been affected by the asset type upgrade.

Status

The Metastate Team has responded that further investigation is needed in order to determine appropriate new external test vectors for each test, which would need to happen prior to making any updates.

Verification

Unresolved.

Recommendations

We recommend that the remaining *Issues* and *Suggestions* stated above be reconsidered and addressed as soon as possible. We also suggest that areas mentioned as unaudited or out of scope in these *Issues* be reviewed as soon as possible for further mitigation of security risks.

We recommend that the Metastate Sapling Specification changes document more clearly specify the security definition of the interconnection of the asset types name, identifier and generator. We also suggest incorporating the Sapling Specification changes document into the overall project documentation so that coverage is comprehensive.

Lastly, we recommend that test coverage be sufficiently adapted and compatible with the asset type upgrades, allowing all tests to compile and pass successfully.

We appreciate that the Metastate team has utilized well known and trusted gadgets, cryptographic primitives and implementation. We commend them for taking further precautions by making only the necessary changes while implementing the updates to the Sapling specification, thus reducing the potential attack surface.

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 C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. 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. Although we are a small team, we believe that 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 included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site 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. While 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, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, 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 though we have not yet 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 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 report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, 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 resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.