Least Authority
PRIVACY MATTERS

Trusted Setup: Phase 1

Security Audit Report

Aleo

Final Report Version: 30 July 2021

Table of Contents

Overview

Background

Aleo has requested that Least Authority perform a security audit of the Aleo Trusted Setup. An Aleo setup consists of coordinator and participant roles. The coordinator initializes the setup, verifies contributions, and moves the protocol forward while the participants download the previous contribution, contribute, and upload their contributions.

The following components are considered in-scope for the review:

- Three Aleo setups including Groth16 and Universal setups.
    - **1)** Universal setup for Marlin on BLS12-377 of size 2^30
        - Universal setup, run by both the coordinator and participants.
    - **2)** Groth16 setup on BLS12-377 of size 2^19 and **3)** Groth16 setup on BW6 of size 2^20 (TBD)
        - Powers of Tau, run by both the coordinator and participants.

Project Dates

- **September 10 - September 25**: Code review *(Completed)*
- **September 30**: Delivery of Initial Audit Report *(Completed)*
- **July 28 - 29:** Verification Review *(Completed)*
- **July 30:** Delivery of Final Audit Report *(Completed)*

Review Team

- Mirco Richter, 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 Trusted Setup 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:

- https://github.com/AleoHQ/aleo-setup/tree/feat/opt-pipeline/phase1
- https://github.com/AleoHQ/aleo-setup/tree/feat/opt-pipeline/phase1-cli
- https://github.com/AleoHQ/aleo-setup/tree/feat/opt-pipeline/phase1-wasm

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

```
313a938e0549f36007c69a4a6cd0955988beee94
```

For the verification, we examined the Git revision:

```
cd27cfb25cfa6569830e7074f9e7e0c951a8b36f
```

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

https://github.com/LeastAuthority/aleo-setup

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:

- Marlin with SonicKZG10: https://hackmd.io/@kobigurk/B1H2lEV7D
- Optimistic pipelining for Groth16: https://hackmd.io/@kobigurk/Bk6nkkdNw
- BGM17: https://eprint.iacr.org/2017/1050.pdf
- Groth16: https://eprint.iacr.org/2016/260.pdf
- Marlin: https://eprint.iacr.org/2019/1047.pdf
- KZG10: https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors in the setup code;
- Value commitment integrity and value base integrity checks;
- Data privacy, data leaking, and information integrity;
- Resistance to 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 completed a security audit of Phase 1 of the Aleo Common Reference String Multi-Party Computation (CRS-MPC) to catch potential errors in advance of the ceremony. Once the computation is complete, errors in the trusted setup are very difficult and expensive to correct, including requiring another trusted setup ceremony to be run.

Our team conducted a line-by-line manual code review of the in-scope repositories. We found the scope of the Phase 1 audit to be sufficient, with the exceptions of the /setup-utils/src/helpers.rs file, which we recommend be included in the scope of the Phase 2 review, given that the code contains a considerable amount of the cryptographically relevant helper functions.

Code Quality + Documentation

Our team found the code to be well organized and logically structured, along with including sufficient test coverage. However, while some code comments are present, large parts of the implementation would benefit from readable and verifiable math in the comments. Given that this information is already present in the project documentation, increasing and expanding the code comments should be relatively easy to achieve (Suggestion 2).

More detailed information in the code comments describing how to securely run a ceremony would benefit future users and prevent errors that occur due to incorrect handling. Although an example of how to execute a CRS-MPC ceremony can be found in /phase1-cli/scripts, we recommend including a clear description of how (with what parameters) and in which order the branches in function `execute_cmd` in file /src/bin/phase1.rs, l. 22, have to be called. This is especially important in “chunk-mode” as the ratio check in the Powers of Tau must be done across chunk borders (Suggestion 2).

Additionally, all of the optimizations that are used and that deviate from BGM17 should be explained in detail, allowing future users to better understand the safety measures in place (Suggestion 1). While Marlin with SonicKZG10 describes the specification of the Marlin CRS-MPC protocol, it lacks clear reasoning of whether or not the deviations from BGM17 have negative implications on security.

System Design

The Aleo system implements BGM17 for a Groth16 CRS-MPC and for a Marlin CRS-MPC. It is clear that security was a primary consideration by the Aleo development team. However, the cryptography library in use is not sufficiently maintained, so we recommend using an up-to-date and regularly maintained cryptography library (Issue C). Further system design improvements such as defining chunk sizes or using vectors in Rust (Suggestion 3) and using a customized hash value (Suggestion 4) are also recommended.

Assumptions

Our team assumes the following to be true:

- Function `E::G1Projective::rand(rng)` produces cryptographically strong pseudorandomness in the base field associated with the prime order group of the curve, provided that `rng` is cryptographically strong.
- The generators g_1 and g_2 from G1 and G2 satisfy the requirements from BGM17 (i.e. they are sampled uniformly at random, no discrete log relations are known, and they are indeed generators of large order prime groups).
- The randomness beacon is external to the codebase and its properties are largely unspecified as a result. We assume that a beacon will be chosen that satisfies the requirements of Section 3.2 in BGM17.
- We assume that BGM17 has a few typos in section 7.1. For the Groth16 initialization of the first round, BGM17 states `[αx_i]^0 := g_1, for all i from [0..n - 1]`, while according to the notation (section 3.1) of that same reference, it should state `[αx_i]_1^0 := g_1, for all i from [0..n - 1]`. Similarly, BGM17 wrote `[x_i]^0 := g_1, for all i from [n..2n - 2]` as well as `[βx_i]^0 := g_1, for all i from [0..n - 1]`, while it should have written `[x_i]_1^0 := g_1, for all i from [n..2n - 2]` as well as `[βx_i]_1^0 := g_1, for all i from [0..n - 1]`. This typo then continues into the computation and verification parts of the first round. Additionally, typos could be spotted in Marlin with SonicKZG10, where in the notation section, "Alpha Tau G1" was incorrectly noted as "Alpha Tau G2". The Aleo development team is aware of those typos and implemented the correct versions in code.
- Batch sizes are set correctly to being batch size > 3 + 3*(total size in log 2), as needed for the CRS-MPC contribution in the Marlin protocol.
- The parameters are generated in such a way that the program does not read/write beyond the bounds of the u8-slices.

If in the case that any of these noted assumptions are untrue, we suggest further investigation to assess the impact on the security of the implementation.

Deviations from BGM17

We compared the implementation to the Groth16 CRS-MPC as described in section 6 of BGM17 and to the Marlin CRS-MPC as described in Marlin with SonicKZG10. In particular, we carefully considered the deviation from BGM17 and the optimizations that have been made. We found that the code implements the Multi-Party Computation (MPC) process as provided, with the exception of the following deviations and optimizations:

1. In BGM17, all ratio checks are done in the exponent of the same generators `g_1` and `g_2` of `G1` and G2, respectively. However, instead of those generators, Aleo uses group elements `g_1^s` and `g_2^s` (for random but non zero field elements `s`) as bases for the ratio checks (e.g. see function `key_generation` in /phase1/src/key_generation.rs, or function `compute_g2_s_key` in /phase1/src/helpers/accumulator.rs. While this is technically a deviation from BGM17, the security proof is not impacted since all ratios are the same in both cases, which follows from:

   `B(g_1^s,f^x) = B(g_1^(sx),f)` if and only if `B(g_1,f^x) = B(g_1^x,f)`

   for all non zero field elements s, field elements x, G2 elements f and pairing B(.,.).

2. In BGM17, the Powers of Tau are computed and verified in a single sequential computation. However, Aleo deviates from this by an approach called optimistic pipelining (see Optimistic pipelining for Groth16). In this approach, participants only receive and process consecutive parts (chunks) of the Powers of Tau. This splits the long sequential computation into parts, opening it to some degree of parallelism.

   During our analysis, we came to the conclusion that this does not impact the security of the system, as long as the verification ensures that power ratios are verified across the boundaries of chunks (i.e. the last element of the previous chunk must have a proper ratio with the first element of the current chunk, and so on). Furthermore, we found that the Aleo development team is aware of this and has provided the following sequence of verification steps:
   a. Execute the `VerifyAndTransformPokAndCorrectness` command on each chunk;
   b. Once all chunks are obtained, execute the `Combine` command to aggregate all chunks together; and
   c. Execute the `VerifyAndTransformRatios` command on the combined chunks to finalize the verification.

3. Function `aggregate_verification` in file /phase1/src/verification.rs executes the ratio checks between consecutive Powers of Tau separately. However, the Aleo development team implemented an approach that deviates from BGM17. Instead of verifying the individual ratios,

   `B(g_1^tau^(j-1), g_2^tau) = B(g_1^tau^(j), g_2)`

   for all `j` in a range `[0..N]`, the identity

   ```
   B(g_1^(tau^0*c_0)*...*g_1^(tau^(N-1)*c_(N-1))}, g_2^tau) =
         B(g_1^(tau*c_0)*...*g_1^(tau^N*c_(N-1))}, g_2)
   ```

   is checked (for random field elements `c_j`) instead as an optimization to reduce the number of computationally expensive pairings. However, this optimization was used without a proof that it is equivalent to the original algorithm. As a result, we recommend that the Aleo development team write such a proof (Suggestion 1).

4. The Groth16 CRS-MPC specification was derived in section 6 of BGM17, including a correctness proof. However, a similar derivation for the Marlin CRS-MPC is not part of BGM17, but is a central contribution of the Aleo code base instead. This can be seen as deviation from BGM17 and a proper derivation of the specifications as given in Marlin with SonicKZG10 from BGM17 is required. We recommend adding documentation on this deviation (Suggestion 1).

Non-Issues

The Aleo codebase uses Rust crate mmap, which is labeled as unsafe, since a compromised machine might eventually manipulate the challenge or response file while the rust code assumes this file to be immutable. This is a general problem with mmap as discussed by the Rust community.

Based on our analysis, a problem that may arise in such a scenario is that some contributions are invalid. This can be detected in the verification step and, as a result, the most harm that can be inflicted is the unnecessary use of computational power and time for the compromised and consecutive Powers of Tau contributions. Thus, we consider this to be a non-issue.

In addition, we found that the crate Crossbeam Channel Version 0.4.3 is labeled as yanked. While this does not appear to have security implications since all existing dependencies continue to be working, we suggest it be further investigated by the Aleo development team.

Specific Issues

We list the issues 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: A Compromised Machine Might Read or Generate False Secret Randomness	Unresolved
Issue B: Hashing to G2 Exposes a Discrete Log Relation to the Generator	Resolved
Issue C: RUSTSEC-2016-0005 Rust-Crypto is Unmaintained	Unresolved
Suggestion 1: Include Proofs for all Deviations and Optimizations from BGM17	Partially Resolved
Suggestion 2: Increase and Expand Code Comments	Unresolved
Suggestion 3: Consider Deserialization of Byte Fields into Structs & Arrays for Better Readability Using Constant Generics	Unresolved
Suggestion 4: Use a Personalized Hash in the Initial Accumulator	Unresolved

Issue A: A Compromised Machine Might Read or Generate False Secret Randomness

Location

Example of how to generate the randomness:
https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/phase1-cli/scripts/phase1_chunked.sh

Actual use of randomness:
https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/phase1-cli/src/contribute.rs

Synopsis

During previous CRS-MPC ceremonies, such as the Zcash Sprout ceremony in 2016, considerable efforts were made by the Electric Coin Company team to secure the underlying hardware used to compute any contribution and reduce the likelihood that it could be compromised. However, if decentralized private computations become widespread, circuit specific CRS-MPCs like those required in Groth16 might become more common and occur frequently, particularly as circuits increase in size. As a result, it cannot be assumed that all ceremony participants will be able to put forth such considerable effort to ensure the integrity of the hardware and software will remain the same as it did for previous ceremonies.

Given this knowledge, we suggest in threat modeling, to assume that the machines involved in ceremonies may be compromised in some way. As a result, the Aleo codebase would benefit from taking additional precautions to protect the random seed and the private key against a compromised system, the latter of which might be able to read or generate false secret randomness.

Impact

The overall impact is low, since the entire CRS-MPC is secure as long as at least one private key among all contributions remains a secret. However, in the unlikely event that an attacker is able to accumulate the secret randomness of all participants, it can compute false proofs in any system that is based on the derived CRS.

Preconditions

In order to read the seed or the private key, an attacker must compromise the executing machine in such a way that they are able to read or write from or to the process memory (e.g. RAM/SWAP). A second precondition would be a random number generator attack, in which the system's randomness generator is compromised in order to generate fake randomness.

Technical Details

In function `key_generation` in file /phase1/src/key_generation.rs, the private key (usually called the toxic waste or the secret field elements) is deterministically derived from a seed that has to be provided as a parameter during program start. The seed and `private_key` then exist during the execution time of the function `contribute` in the file /phase1-cli/src/contribute.rs. However, depending on the circuit size, that function might run for an extended period, giving an attacker enough time to complete their attack. After termination, the associated memory is deallocated but not actively overwritten. Furthermore, no precautions have been taken to prohibit the compiler from allocating various copies of that secret in different parts of the memory, including less volatile parts (e.g. a swap partition).

Remediation

As long as any contribution is executed in a single chunk, there is no reason for the seed randomness to be generated outside of the function `contribute`. In this scenario, Rust's crate `secret` can be used as protected-access memory for the seed. This guarantees that the memory is both freed and overwritten after termination.

Unfortunately, if the contribution is provided across more than one chunk, the previously mentioned approach will not sufficiently address the issue since the secret randomness needs to be shared across different computations. As a result, we recommend a pre-contribution phase that generates the randomness inside Rust's crate `secret` and encrypts it using strong symmetric encryption and a user defined password, the latter of which is, for example, stored in an OS keyring or memorized by the user. The encrypted toxic waste can then be stored safely and used in the various chunk computations. In each computation, it is decrypted and then handled as we suggested under the single chunk computations.

Status

The Aleo team has responded that, since one non-compromised participant is sufficient for security, they do not plan to resolve this issue. Given that the impact of this issue is indeed minimal since an attacker needs to compromise all ceremony participants to gain knowledge over the hidden entropy used, we consider the decision to leave the issue unresolved at the time of verification to be acceptable.

Verification

Unresolved.

Issue B: Hashing to G2 Exposes a Discrete Log Relation to the Generator

Location

Function `hash_to_g2` in file:
https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/setup-utils/src/helpers.rs

Synopsis

As described in BGM17, CRS-MPCs require access to an oracle, whose outputs are uniform and independent elements of the group G2. This implies that no discrete logarithm relation between different oracle queries should be known to any user of the system. However, we found that the approach taken by the Aleo development team trivially exposes a discrete logarithm relation between any output and the generator.

Impact

Having knowledge of discrete logarithm relations between any output and the generator imply discrete logarithm relations between any two outputs, which in turn violates the assumptions of BGM17 on the independence of outputs. As a consequence, the security proof no longer holds true.

Feasibility

This is very feasible as the discrete logarithm relation is easy to compute.

Technical Details

Function `hash_to_g2` calls function `sample` from file
algebra-core/src/curves/models/short_weierstrass_jacobian.rs, which computes the hash to curve as follows:

Let g_2 be a generator of G2 and x be the byte string that needs to be hashed to G2. Then the curve hash is computed as,

```
        H_G2(x) := g_2^Hash(x),
```

where `Hash(.)` is a cryptographically secure hash function that hashes to the base field. The discrete log relation between `H_G2(x)` and `g_2` is then trivially seen as `Hash(x)`.

Remediation

A secure approach would be to hash separately to the x and the y coordinates and check if the result is actually a point in the required large order prime group. If it is not in the prime order subgroup, changing the initial byte string is necessary. Depending on the approach, proper cofactor clearing may also be necessary.

Status

The Aleo team rewrote the `hash_to_G2` function and implemented a try-and-increment approach for hashing into G1 of the twist in the usual way. This was done by first hashing into the base field, trying to find a curve point, and then doing a cofactor multiplication to project into G1 of the twist. Since the twist isomorphism identifies G1 of the twist with G2 of the original curve over the extension field `F_{q^k}` with `k` being the embedding degree, this implements a way to hash into G2. We analyzed the implementation and were not able to identify any new issues. Thus, we consider the original issue to be resolved.

Verification

Resolved.

Issue C: RUSTSEC-2016-0005 Rust-Crypto is Unmaintained

Location

https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/setup-utils/Cargo.toml

https://rustsec.org/advisories/RUSTSEC-2016-0005.html

Synopsis

The Aleo codebase uses the Rust-Crypto crate in version 0.2. However, according to the Rust advisory database, RUSTSEC-2016-0005, that particular crate has not seen a release or GitHub commit since 2016 and its author is unresponsive.

Impact

While vulnerabilities have not been identified in this crate to date, the Rust advisory database recommends to change that dependency to an actively maintained crate.

Remediation

We recommend switching to a maintained cryptography library as advised in RUSTSEC-2016-0005.

Status

The Aleo team has stated that the Rust-Crypto crate in version 0.2 is only used for SHA2 hashing in Beacon code and is therefore unnecessary to remediate for security, from their perspective. We verified that the Rust-Crypto crate in version 0.2 is only used in this specific setting which then, as an example, generates randomness for the beacon in Phase 1. Even if the crate is only used in one place in the code, we nevertheless recommend switching to a maintained cryptography library in the future.

Verification

Unresolved.

Suggestions

Suggestion 1: Include Proofs for all Deviations and Optimizations from BGM17

Location

https://github.com/AleoHQ/aleo-setup

BGM17: https://eprint.iacr.org/2017/1050.pdf

Marlin: https://eprint.iacr.org/2019/1047.pdf

Synopsis

The in-scope repository contains an implementation of the MPC of BGM17. As described in detail in the section Deviations from BGM17 above, the codebase deviates from the CRS-MPC algorithm as described in BGM17.

For example, the codebase extends the work of BGM17 for functionality with the Groth16 zk-SNARK to functionality with Marlin. In BGM17, a player-exchangeable MPC for the generation of the CRS for zk-SNARKs is presented and security for Groth16 is proven. However, the security of the player-exchangeable MPC is not yet shown for Marlin in the work of BGM17.

Mitigation

We recommend documenting proper security proofs and reasoning for those deviations, demonstrating that the security proof of BGM17 is maintained, and that other new constructions as listed in the section Deviations from BGM17 can be proven secure, as well.

Status

The Aleo team has provided recent literature (i.e. 2021) by Kohlweiss et al.. The proposal by Kohlweiss et al. (also here) provides a new security framework for non-interactive zero-knowledge protocols with ceremony arguments, which generalizes the notion of updatable reference strings as needed for Marlin. However, we did not verify the new security framework and the provided security proofs in the work by Kohlweiss et al., as this is out of scope for the verification. As such, this suggestion is partially resolved.

Verification

Partially Resolved.

Suggestion 2: Increase and Expand Code Comments

Location

https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/phase1/

https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/phase1-cli/

https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/phase1-wasm/

Synopsis

The implementation would benefit from additional code comments due to the complex nature of the code base, as well as new methods of calculation (i.e. through optimistic pipelining for Groth16 and the extension for Marlin in the style of Sonic and KZG10 batching).

Mitigation

Given that information is already summarised in Marlin with SonicKZG10 and considering the number of calculations being executed in the code, readable and verifiable math in the comments would assist in future debugging and auditing of the code base. As a result, we recommend increasing and expanding on the code comments, also as noted in the section Code Quality + Documentation above.

Status

The Aleo team has stated that they are committed to increasing code comments in the future. We encourage the Aleo team to expand code comments due to the mathematical nature of the code base.

Verification

Unresolved.

Suggestion 3: Consider Deserialization of Byte Fields into Structs & Arrays for Better Readability Using Constant Generics

Location

https://github.com/AleoHQ/aleo-setup

Synopsis

In the Aleo code base, basic data structures, like the accumulator, are mostly handled as raw byte sequences and if those data structures have to be passed as function arguments, slices are used. However, dealing with raw byte fields makes reviewing the implementation more difficult, as the logical structure of the byte field is not easily deducible from the visual appearance in source code. Furthermore, the capabilities of the compiler may not be fully exploited.

Rust is currently developing the const_generic feature, which might make the code more readable. Using the example of the `challenge` byte field and the aforementioned feature, a type such as the following is more easily readable from the auditors perspective in contrast to `Challenge: &[u8]`:

```
#![feature(const_generics)]

Struct Challenge<const N1: usize, const N2: usize> {
      previous_hash: [u8,HASH_LENGTH],
      tau_g1: [G1; N1],
      tau_g2: [G2; N2],
      alpha_tau_g1: [G1; N2],
      beta_tau_g1: [G1, N2],
      beta_g2: G2,
      public_key: Option<PublicKey>,
}
```

For example, reading something like `Challenge.tau_g2[0]` is more intuitive than `Challenge[HASH_LENGTH+(2n-2)*G1.size_in_byte()]`.

Mitigation

We recommend using more descriptive types whenever possible to make the code more easily accessible in terms of reasoning and readability. However, constant generics are currently not available in stable Rust.

Status

The Aleo team has responded they will not address this suggestion as they find it unnecessary. However, it is worth noting that the problem described in this suggestion led the Aleo team to discover a bug shortly after the completion of our security audit, where sizes of contribution files were calculated incorrectly in the Phase1 Parameters struct. This bug was subsequently and immediately addressed by the Aleo team.

To our understanding, this issue resulted from handling values not through comprehensive data structures but byte sequences and slices. While it is not trivial to incorporate descriptive types, we still encourage the Aleo team to review their basic data structures and make full use of the capabilities of the Rust compiler.

Verification

Unresolved.

Suggestion 4: Use a Personalized Hash in the Initial Accumulator

Location

Function `new_challange` in file
https://github.com/AleoHQ/aleo-setup/blob/313a938e0549f36007c69a4a6cd0955988beee94/phase1-cli/src/new_challenge.rs

Synopsis

Over time, users may want to have more then one CRS-MPC for the same circuit size in the event that, for example, a particular ceremony is not trusted anymore for various reasons. However due to the use of function `blank_hash` and a static generator in function `initialization`, the initial accumulator is exactly the same for all ceremonies that have the same circuit size. This may confuse and mix associated contributions.

Mitigation

We recommend an approach for ceremony separation where a personalized initial hash is used instead of simply hashing the empty set. For example, a hash of the agreed on generator together with a date or a version number may be used.

Status

The Aleo team has responded they will not address this suggestion as they find it unnecessary. However, we encourage and recommend using a personalized hash in the initial accumulator in order to have the possibility for various contributions for different ceremonies.

Verification

Unresolved.

About Least Authority

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

The Least Authority team has skills for reviewing code in 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.