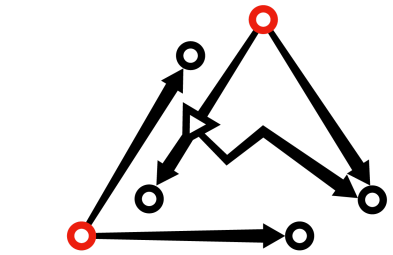

Least Authority

PRIVACY MATTERS

MPC-ECDSA Algorithm

Security Audit Report

Safeheron

Updated Final Audit Report: 19 October 2023

Table of Contents

Overview

Background

Safeheron has requested that Least Authority perform a security audit of their MPC-ECDSA algorithm.

Project Dates

- **August 7, 2023 - August 30, 2023** Initial Code Review *(Completed)*
- **September 1, 2023:** Delivery of Initial Audit Report *(Completed)*
- **September 27:** Verification Review *(Completed)*
- **September 28:** Delivery of Final Audit Report *(Completed)*
- **October 19:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

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

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the MPC-ECDSA algorithm 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:

- Safeheron Multi-Party Signature: https://gitlab.com/safeheron/audit/safeheron-multi-party-signature-cpp/-/tree/main/proto/multi-party-ecdsa-cpp/cmp
- Safeheron Crypto Suites: https://gitlab.com/safeheron/audit/safeheron-crypto-suites-cpp/-/tree/main/src/crypto-zkp-cpp

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

- Safeheron Multi-Party Signature: `b22eb43c5656d906219657add5da3a2751374b86`
- Safeheron Crypto Suites: `08d5e26587a08828d7a628e973b3d4d2b7905c70`

For the verification, we examined the Git revisions:

- Safeheron Multi-Party Signature:
  `39677c79ff0581add5325b8d1717e78b3172d046`
  `4e370d4fdcd0ea799147d211f43bb22d01f2ae9c`
  `dd8212ab096e9dfc3e94637dbbb850706bd42ea6`

- Safeheron Crypto Suites:
  `59cb14b6a0fb120421a6e7caad986e79129e5391`
  `6301a86a419f095e36d05a73d58eafe1057593c8`
  `dac96f252a0d6851889f22dc3258c39b478142b3`
  `6e88eb57c333d92681bf8d00df21b68b68856bc8`
  `8278a4acea347113c53334241b9a34614b7045c7`

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

- Safeheron Multi-Party Signature:
  https://github.com/LeastAuthority/safeheron-mpc-algorithm
- Safeheron Crypto Suites :
  https://github.com/LeastAuthority/Safeheron-Multi-Party-Signature-cpp

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

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

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://www.safeheron.com/en-US
- Safeheron Modification Doc:
  https://github.com/Safeheron/multi-party-ecdsa-cpp/blob/main/Modify-MPC-CMP-as-a-Threshold-Signature-Scheme.pdf

In addition, this audit report references the following documents:

- R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled, "UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts." *IACR Cryptology ePrint Archive*, 2021, [CGG+21]
- Q. Dao, J. Miller, O. Wright, and P. Grubbs, "Weak Fiat-Shamir Attacks on Modern Proof Systems." *IACR Cryptology ePrint Archive*, 2023, [DMW+23]
- N. Makriyannis and U. Peled, "A Note on the Security of GG18." *Fireblocks*, 2021, [MP21]
- R. Gennaro and S. Goldfeder, "Fast Multiparty Threshold ECDSA with Fast Trustless Setup." *IACR Cryptology ePrint Archive*, 2019, [GG18]
- R. Gennaro and S. Goldfeder, "One Round Threshold ECDSA with Identifiable Abort." *IACR Cryptology ePrint Archive*, 2020, [GG20]
- Heartbleed attack:
  https://heartbleed.com
- OpenSSL's BigNum:
  https://www.openssl.org/docs/man1.0.2/man3/bn.html

Areas of Concern

Our investigation focused on the following areas:

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

Findings

General Comments

The Safeheron team implemented an `MPC-ECDSA` protocol, as described in [CGG+21]. This protocol builds on, and updates, former protocols, such as [GG18] and [GG20]. Both of these MPC-ECDSA protocols have a known history of theoretical and implementation issues, as noted in [MP21], whereas no impacts on [CGG+21] are known at the time of writing of this report.

In [CGG+21], the multi-party signing protocol operates in a setting where `n-out-of-n` parties have to sign a message in order to provide a valid signature. As the authors explain, the scheme can be adjusted to a setting where `t-out-of-n` parties are sufficient to sign a message for any threshold `1<t<n` by using a secret sharing scheme (Section 1.2.8). The Safeheron team adjusted the algorithm to the threshold setting and described the necessary modifications in the Safeheron Modifications Doc. Although the security proof explained in [CGG+21] is expected to hold for an adopted protocol in the `t-out-of-n` setting, we recommend documenting the protocol modifications clearly to include a full specification for Safeheron's modified threshold setting and an adjusted security proof (Suggestion 4).

The Safeheron `MPC-ECDSA` protocol is working in a mode of online signing and consists of three phases: the key generation phase, the auxiliary information generation and key refresh phase, and the pre-signing and signing phase. All three phases of Safeheron's `MPC-ECDSA` Protocol are implemented in the Safeheron Multi-Party Signature repository. Additionally, the Safeheron team implemented all needed Non-Interactive Zero-Knowledge Proofs of Knowledge (NIZKs) as well as the needed Paillier cryptosystem in the Safeheron Crypto Suites repository.

Implementation

We performed a manual review of the repositories in scope and found the codebases to be generally well-organized and well-written.

In our review, we compared Safeheron's implementation of the `MPC-ECDSA` algorithm with the underlying [CGG+21] paper. We focused specifically on the modification to the `t-out-of-n` setting described in the Safeheron Modification Doc and identified several instances of unwanted diversions between the implementation and the [CGG+21] paper and the Safeheron Modification Doc (Issue B).

In addition to the areas of concerns listed above, our team also investigated several areas for security vulnerabilities and implementation errors. We analyzed the underlying cryptography needed for the implementation of the [CGG+21] algorithm, such as the implemented Paillier cryptosystem as well as the implementation of all non-interactive zero-knowledge proofs of knowledge (NIZKs) required in the [CGG+21] protocol. We examined the proper usage of the strong Fiat-Shamir transformation, as described in [DMW+23]. In the implementation, we identified multiple locations where weak Fiat-Shamir transformations are used, which could undermine the soundness of the NIZKs, thus making the system highly vulnerable to various attack vectors (Issue A).

We analyzed the creation and handling of the shared randomness implemented for the threshold ECDSA signature scheme. During the signing process, an MPCContext Object is generated, which entails the user's share of the randomness used for signing. This object is released after the signing step and the randomness is destructed.

Our team also identified an instance of an incorrectly used session identifier, which can lead to compromise of the key generation process (Issue C). In addition, we identified an opportunity for the improvement of random sampling from proper types (Suggestion 2).

Tests

The Safeheron team has implemented sufficient tests, which showed high coverage of important logic, such as the Protobuf serialization.

Documentation & Code Comments

The project documentation provided for this review offers a generally sufficient overview of the system and its intended behavior. We found that code comments sufficiently describe the expected behavior of security-critical components and functions.

Scope

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

Dependencies

Our team did not identify any security concerns resulting from an unsafe use of dependencies. Additionally, our team also found that the implementation adheres to recommended best practices as demonstrated, for example, by the use of the OpenSSL's BigNum implementation.

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: Weak Fiat-Shamir Transformation Implemented in Various NIZKs	Resolved
Issue B: Implementation Deviates From Protocol Description	Resolved
Issue C: Session Identifier Used Incorrectly	Resolved
Suggestion 1: Write the Salt at the Beginning of Each Fiat-Shamir Transcript	Resolved
Suggestion 2: Sample Randomness From Proper Types	Resolved
Suggestion 3: Update Typographical Errors	Resolved
Suggestion 4: Document Modification for Threshold Setting Fully and Update Security Proof	Resolved

Issue A: Weak Fiat-Shamir Transformation Implemented in Various NIZKs

Location

`src/crypto-zkp/dlog_elgamal_com_proof.cpp`

`src/crypto-zkp/no_small_factor_proof.cpp#L53`

`src/crypto-zkp/no_small_factor_proof.cpp#L32`

src/crypto-zkp/no_small_factor_proof.cpp#L33

src/crypto-zkp/pail/pail_aff_range_proof.cpp#L32

src/crypto-zkp/pail/pail_blum_modulus_proof.cpp#L58

src/crypto-zkp/pail/pail_dec_modulo_proof.cpp#L25

src/crypto-zkp/dlog_elgamal_com_proof.cpp#L27

src/crypto-zkp/dlog_elgamal_com_proof.cpp#L31

src/crypto-zkp/dlog_equality_proof.cpp#L26

crypto-zkp/pail/pail_dec_modulo_proof.cpp#L26-L28

Synopsis

The MPC-ECDSA algorithm, as specified in [CGG+21], utilizes NIZKs derived from Sigma Protocols using the Fiat-Shamir transformation. As explained in [DMW+23], many NIZKs require an implementation of the *strong* Fiat-Shamir transformation, which has to include every piece of information into the initial transcript of the hash function that is publically available to the verifier after the first round. However, we identified various instances in the code where a weak Fiat-Shamir transformation was used instead, breaking the soundness proof of [CGG+21] and hence making the system vulnerable to unforeseeable attack vectors.

Impact

Critical. The security of the MPC-ECDSA algorithm is critically dependent on the validity of the Sigma protocol relations, as described in section 6 of [CGG+21].

For example, in src/crypto-zkp/dlog_elgamal_com_proof.cpp#L70, the implementation transforms the Sigma Protocol from figure 24 in [CGG+21] into a NIZK. The code includes the prover message `(A, N, B)` of the first round into the Fiat-Shamir transformation, giving the simulated verifier challenge `e = H(A||N||B)`. However, the public inputs L, M, X, Y and the generators g and h are not included in the hash. This breaks the adaptive knowledge soundness of the El-Gamal Commitment NIZK.

To illustrate, consider a malicious prover that can generate a valid NIZK without having a proper witness by choosing invertible field elements z and u and group elements A, N, B, X, such that `e = H(A||N||B)` is invertible. The prover is then able to construct a simulated proof for the public inputs `(L, M, X ,Y)` by defining `L = (g^z/A)^(1/e)`, `Y = (h^u/B)^(1/e)`, and `M = ((g^u·X^z)/N)^(1/e)`.
Moreover, since the generator h is not included into the simulated verifier challenge, it is possible for a malicious prover to transform a valid proof for generator h into a simulated proof for another generator h'.
In the linked locations, weak Fiat-Shamir transformations are applied in multiple locations. Consequently, it is a complex endeavor to reason about every potential attack vector.

Technical Details

In [CGG+21], the transformation of the Sigma Protocols from section 6 is abstracted by the introduction of a ZK-Module M (section 2.3.1), which takes a sigma model as input and computes a NIZK, assuming the existence of a hash function that behaves like a random oracle.
This module describes in detail all data that is necessary to compute the strong Fiat-Shamir transformation of a Sigma Protocol, which, in general, must include a hash over every data that is known to the verifier after the first round of the Sigma protocol. It consists of a description of the underlying cryptographic system, the proof relation, every auxiliary input and the public inputs. However, the

implementation deviates from this in multiple places.
Inclusion of a description of the underlying cryptographic system and the proof relation is possible since the code allows for the inclusion of a 'salt' value that can be initialized with the system's session identifier (SID) or the key-specific sub-session identifier (SSID). However, there are places in the code where this is not done correctly.

For example, the code does not include:

- Initial messages (e.g. `src/crypto-zkp/no_small_factor_proof.cpp#L53`);
- Public inputs (e.g. `src/crypto-zkp/dlog_elgamal_com_proof.cpp#L27` or `crypto-zkp/pail/pail_aff_range_proof.cpp#L32`);
- Statement descriptions (e.g. `src/crypto-zkp/dlog_elgamal_com_proof.cpp#L31` or `src/crypto-zkp/dlog_equality_proof.cpp#L26`); and
- Auxiliary inputs (e.g. `crypto-zkp/pail/pail_dec_modulo_proof.cpp#L26-L28`).

Remediation

The algorithms in [CGG+21] give a detailed description of what data should be included into each Fiat-Shamir transformation.
We recommend updating the code, such that it initializes the salt field of each NIZK with accurate SID, SSID, and AUX values to ensure that the cryptographic system, the proof relation, and any auxiliary values, such as the index, are represented as intended.
In addition, we also recommend that the code include all public inputs, generators, and initial prover messages into the Fiat-Shamir hash.

Status

The Safeheron team has implemented the remediation as recommended.

Verification

Resolved.

Issue B: Implementation Deviates From Protocol Description

Location

`crypto-zkp/pail/pail_dec_modulo_proof.cpp#L96`

`src/crypto-zkp/dln_proof.cpp#L69-L70`

`cmp/aux_info_key_refresh/round3.cpp#L67`

`cmp/aux_info_key_refresh/round0.cpp#L132`

`cmp/aux_info_key_refresh/round0.cpp#L137`

`crypto-zkp/pail/pail_aff_range_proof.cpp#L124`

`crypto-zkp/pail/pail_aff_range_proof.cpp#L125`

`crypto-zkp/pail/pail_blum_modulus_proof.cpp#L224`

Synopsis

During our audit, we found that the implementation deviates from the specification described in [GG18] and [CGG+21].

Impact

High. The deviations that our team identified break the correctness of the MPC-ECDSA scheme and the security proof of [CGG+21].

Technical Details

- The 'Paillier Decryption modulo q' NIZK (Figure 30) requires the computation `w = r * \rho^e mod N_0`. However, the code implements `w = r * \rho^e mod N_0^2` (crypto-zkp/pail/pail_dec_modulo_proof.cpp#L96).
- The 'Ring-Pedersen Parameters' NIZK (Figure 17) requires the parameters h1 and h2 to be elements from `Z_N^*`. However, a check is missing in the verifier that confirms whether h1 and h2 are coprime to N (src/crypto-zkp/dln_proof.cpp#L69).
- The 'Auxiliary Info. & Key Refresh' algorithm (Figure 6, output) requires computing `x_ij = dec(C_ji) mod q` and then checking that `X_ij = g^x_ij`. However, the implementation does not include the modular q operation. As a result, the definition of `\mu = (C^i_j * (1+N_i)^(-x_ij) )^(1/N) mod N^2` results in a different `\mu` (cmp/aux_info_key_refresh/round3.cpp#L67).
- The 'Auxiliary Info. & Key Refresh' algorithm (Figure 6, round 1) requires computing the hash `V_i`, which includes the preimage of the two (uncompressed) curve points `Y_i` and `B_i`. Although both the x-coordinate and y-coordinate of each point need to be hashed, the code mistakenly hashes, twice, the x-coordinate of those points only (cmp/aux_info_key_refresh/round0.cpp#L130-L137).
- The 'Paillier Encryption Range Proof' ([GG18], A.3) requires the verifier to check that `s_1` is a value from `Z_{q^3}` and `t_1` is a value from Z_{q^7}. However, the implementation checks for `s_1` in `[-q^3, q^3]` and `t_1` in `[-q^7, q^7]` (crypto-zkp/pail/pail_aff_range_proof.cpp#L124-L125).
- The 'Paillier-Blum Modulus' NIZK (Figure 16) requires the computation of elements `z_i` from `Z_N`. In the implementation, the verifier checks that `z_i` is neither zero nor one. However, the way those numbers are constructed, the case `z_i = 1` can potentially occur in a correct proof that the verifier would then reject (crypto-zkp/pail/pail_blum_modulus_proof.cpp#L224).

Remediation

We recommend updating the implementation according to the specification.

Status

The Safeheron team has implemented the remediation as recommended (as shown here and here).

Verification

Resolved.

Issue C: Session Identifier Used Incorrectly

Location

cmp/minimal_key_gen/round0.cpp#L89-L90

Synopsis

The sub-session identifier (SSID) is used in the key generation algorithm instead of the session identifier (SID).

Impact

High. This results in a deviation from the paper and also compromises the key generation process since the SSID should only be generated afterwards.

Technical Details

The implementation in the [CGG+21] Protocol is divided into several parts – the key generation process, the auxiliary info and key refresh process, and the signing process. To ensure that these processes are coordinated, several identifiers are instantiated. The overall session identifier (SID, Figure 5, round 1) is set and passed during the key generation process. After that, the sub-session identifier (SSID), which entails the SID, (SSID, Figure 7, round 1) is set and passed to the signing process. Once the signing process is over, the sub-session identifier is erased.

In the `ComputeVerify` function in Round 1 of the Key Generation algorithm, an incorrect session identifier is passed. Here, the SSID is passed instead of the SID, which poses problems for the correct coordination of processes.

Remediation

We recommend updating the `ComputeVerify` function, such that the SID is used instead of the SSID.

Status

The Safeheron team has implemented the remediation as recommended.

Verification

Resolved.

Suggestions

Suggestion 1: Write the Salt at the Beginning of Each Fiat-Shamir Transcript

Location

`crypto-zkp-cpp/src/crypto-zkp`

Synopsis

According to Figure 5 of the [CGG+21] paper, the SID is a string `( ..., G, q, g, P )`, which includes a representation of all involved parties `P= (P_0,P_1,...P_n)`. In such a description, edge cases are possible, where the SID of an `(n+1)`-party protocol is an extension of the SID of an n-party protocol since `(P_0, P_1, ...P_{n+1})` could be implemented on the byte level as `P_0 || P_1 || … || P_n || P_{n+1}`.

Since the code instantiates the Random Oracle abstraction with members from the SHA2 hash function family, and implements the Fiat-Shamir transformation in such a way that the SID is included as the last entry into the Fiat-Shamir hash, this might become an issue because SHA2 is vulnerable to length extension effects.

Mitigation

We recommend including the salt field, and hence the SID description, as the first element of the Fiat-Shamir transcript to prevent potential length extension effects from being exploited.

Status

The Safeheron team has implemented the remediation as recommended.

Verification

Resolved.

Suggestion 2: Sample Randomness From Proper Types

Location

Example (non-exhaustive):

cmp/minimal_key_gen/round0.cpp#L62

Synopsis

The algorithm often requires random samples from scalar fields of elliptic curves. However, the code frequently replaces those samples by sampling 256-bit Big Nums instead. Subsequent modulus operations then include tiny biases into the uniformity assumptions of the samples.

Mitigation

We recommend sampling from the required, specified types.

Status

The Safeheron team has implemented the remediation as recommended.

Verification

Resolved.

Suggestion 3: Update Typographical Errors

Location

- Incorrect comment (SSID instead of SID): cmp/aux_info_key_refresh/round0.cpp#L99
- Incorrect Figure Number (Figure 22 instead of Figure 23): src/crypto-zkp/dlog_equality_proof.h#L24
- Incorrect comment (r instead of gamma): crypto-zkp/pail/pail_dec_modulo_proof.cpp#L155
- Bug without effect (modulo q is in exponent for the generator): crypto-zkp/pail/pail_enc_elgamal_com_range_proof.cpp#L106
- Unnecessary computation (verifier does not need this information): crypto-zkp/pail/pail_aff_group_ele_range_proof_v2.cpp#L159
- Unnecessary computation (delta can be set after the loop): cmp/sign/round2.cpp#L229
- Definition 1.1 of Feldman's Verifiable Secret Sharing using M as a module name. This might be misleading in the protocol description, in Sections 1-3 (M has a special meaning in the [CGG+21] paper, being the ZK module (Section 2.3.1, Figure 2)): blob/main/Modify-MPC-CMP-as-a-Threshold-Signature-Scheme.pdf

Synopsis

We identified a small number of typographical errors, unnecessary computations, and a bug without effect throughout the code – mainly in the code comments and the documentation.

Mitigation

We recommend updating the text and computations in the locations listed above.

Status

The Safeheron team has updated text and computations in all of the locations.

Verification

Partially Resolved.

Suggestion 4: Document Modification for Threshold Setting Fully and Update Security Proof

Location

[CGG+21]

blob/main/Modify-MPC-CMP-as-a-Threshold-Signature-Scheme.pdf

Synopsis

The `MPC-ECDSA` Protocol described in [CGG+21] is being defined for an `n-out-of-n` setting but can be extended to a `t-out-of-n` setting, according to the authors (see Section 1.2.8).
The Safeheron team has adjusted this algorithm for a `t-out-of-n` setting, and the modifications are described in the Modifications Doc. While this document outlines the changes roughly, a modification with an adjusted security proof and a full specification describing this modified setting and the Safeheron implementation is missing.

Mitigation

We recommend updating the documentation and adding:

- A full modification of the `MPC-ECDSA` Protocol, as described in Section 1.2.8 in [CGG+21] for the `t-out-of-n` setting;
- An updated security proof for the `MPC-ECDSA` Protocol, as described in Section 1.2.8 in [CGG+21] for the `t-out-of-n` setting; and
- A specification for the Safeheron `MPC-ECDSA` Protocol, including used constants and their justification, used elliptic curves, used hash functions, Fiat-Shamir transcripts, etc.

Status

The Safeheron team has updated the documentation as suggested.

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, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

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

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

Our Methodology

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

Manual Code Review

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

Vulnerability Analysis

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

Documenting Results

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

Suggested Solutions

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

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

Resolutions & Publishing

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