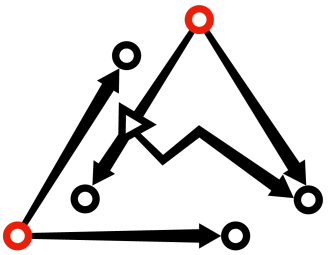

Least Authority

PRIVACY MATTERS

Crypto Suites & Multiparty ECDSA (Incremental Changes) + Encapsulation Layer

Security Audit Report

Safeheron

Updated Final Audit Report: 5 February 2024

Table of Contents

Overview

Background

Safeheron has requested that Least Authority perform a security audit of their Safeheron Crypto Suites CPP, incremental changes to the Crypto Suites and Multiparty ECDSA, along with a review of the Encapsulation Layer for these algorithms.

Project Dates

- **December 4, 2023 - December 15, 2023:** Initial Code Review *(Completed)*
- **December 19, 2023:** Delivery of Initial Audit Report *(Completed)*
- **January 3, 2024):** Verification Review *(Completed)*
- **January 4, 2024:** Delivery of Final Audit Report *(Completed)*
- **January 9, 2024:** Delivery of Updated Final Audit Report - Version 1 *(Completed)*
- **February 5, 2024:** Delivery of Updated Final Audit Report - Version 2 *(Completed)*

Review Team

- Anna Kaplan, Cryptography Researcher and Engineer
- Xenofon Mitakidis, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Safeheron Crypto Suites CPP, incremental changes to the Crypto Suites and Multiparty ECDSA, along with a review of the Encapsulation Layer for these algorithms 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:

- Audit for Safeheron Crypto Suites CPP (except *crypto-zkp-cpp)*
  - https://github.com/Safeheron/safeheron-crypto-suites-cpp
- Incremental audit for crypto-zkp-cpp
  - https://github.com/Safeheron/safeheron-crypto-suites-cpp
- Incremental audit for multi-party-sig-cpp
  - https://github.com/Safeheron/multi-party-sig-cpp
- Incremental audit for multi-flow-cpp
  - https://github.com/Safeheron/multi-party-sig-cpp/tree/main/src/multi-party-sig/mpc-flow
- C++/WASM library (only src)
  - https://github.com/Safeheron/mpc-snap-wasm

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

- crypto-suites-cpp: https://github.com/LeastAuthority/safeheron-crypto-suites-cpp
- multi-party-sig-cpp: https://github.com/LeastAuthority/safeheron-multi-party-sig-cpp

- multi-party-sig/mpc-flow: https://github.com/LeastAuthority/safeheron-multi-party-ecdsa-cpp/tree/main/src/multi-party-sig/mpc-flow
- C++/WASM library: https://github.com/LeastAuthority/safeheron-mpc-wasm

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

- crypto-suites-cpp: `02641725e3448065a02dfa004f40fb266b14a007`
- multi-party-sig-cpp: `c5bc655bb6e6fa5c9fe60775aae3cbac2c3b4b58`
- mpc-snap-wasm: `30d838acb6f6c0e25bb4e1bb7b1f6dc1c0aed0a0`

For the verification, we examined the Git revisions:

- crypto-suites-cpp: `60c5e730926def3c34c0cbb8acdc3173c4f77ede`
- multi-party-sig-cpp: `a52b74ab6d082b7dcd995203477c2ff1bd0fff9f`
- mpc-snap-wasm: `a3871658e0d6ea925a22431be59e45c364534dea`

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 and resources were available to the review team:

- Website: https://www.safeheron.com
- Audit Difference Analysis (Google Doc file) *(shared with Least Authority via email on 29 November 2023)*
- LA_diff.zip *(shared with Least Authority via email on 29 November 2023)*
- 2-3 TSS Scenario Key Recovery Protocol.pdf *(shared with Least Authority via email on 29 November)*
- Updated 2-3 TSS Scenario Key Recovery Protocol with Security Proof.pdf *(shared with Least Authority via Slack on 5 December)*

In addition, this audit report references the following documents:

- D. R. L. Brown, "SEC 1: Elliptic Curve Cryptography." *Certicom Corp.*, 2009, [Brown09]
- 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]
- R. Gennaro and S. Goldfeder, "One Round Threshold ECDSA with Identifiable Abort." *IACR Cryptology ePrint Archive*, 2020, [GG20]
- RFC 9180: https://www.rfc-editor.org/rfc/rfc9180.html
- Update to Current Use and Deprecation of TDEA: https://csrc.nist.gov/news/2017/update-to-current-use-and-deprecation-of-tdea
- NIST Retires SHA-1 Cryptographic Algorithm: https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm

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;
- Whether requests are correctly passed to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the 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

Our team previously conducted an audit on Safeheron's MPC-ECDSA protocol implementation, as described in [CGG+21], and delivered the final audit report on October 19, 2023. In the aforementioned report, we closely investigated how the Safeheron team adjusted the threshold setting, such that `t-out-of-n` parties were sufficient to sign a message. This report is a follow-up of the former report and focuses on the incremental changes implemented to the Crypto Suites and Multiparty ECDSA. In this audit, we reviewed cryptographic primitives, such as BigNums, elliptic curves, cryptographic hash functions and the Paillier cipher as well as Safeheron's implementation of a safe hash function for Fiat Shamir.

The Safeheron team has implemented a custom key recovery algorithm based on Shamir Secret Sharing. We reviewed the corresponding security proof that the team wrote specifically for this custom algorithm and found that it could be improved to adhere to standards for provable security (Suggestion 6).

Additionally, we reviewed the Encapsulation Layer for these algorithms. The Safeheron team implemented wrappers for the key generation, signing, key recovery and auxiliary info, as well as utility functions, such as Elliptic Curve Integrated Encryption Scheme (ECIES) encryption and decryption.

System Design

Our team found that the Safeheron team has prioritized security in the design and implementation of their crypto suites as demonstrated by generally well-implemented advanced cryptography and adherence to best practices.

During our audit, we investigated Safeheron's custom `safe_hash` function. Our team noted that the revised implementation prevents the issue of preimage collisions in case the attacker controls multiple consecutive inputs to the hash function's state. We therefore considered it to be a more secure alternative to the previously used hash functions in settings like the Fiat Shamir transformation. However, we identified a potential overflow issue, which can lead to forged proofs (Issue A).

We examined Safeheron's implementation of BIP39 and compared it to the reference. We also reviewed the tests and confirmed that the Safeheron team has tested their implementation against the reference. Additionally, we compared and tested Safeheron's implementation of standard cryptographic hash functions against the OpenSSL implementations. We could not identify any issues or deviations.

In our review of the cryptographic primitives, we found that the code is generally well-written and organized. However, we identified several issues and areas for improvement (Issue C, Issue D, Suggestion 3, Suggestion 4).

Moreover, we could not identify any issues in the implementation of wrappers. The communication model adheres to the requirements described in [CGG+21]. We investigated potential ways to intercept peer-to-peer (P2P) messages and did not identify any. Moreover, we tested the implementation for data disclosure via memory leaks and did not identify any issues.

Code Quality

We performed a manual review of the repositories in scope and found the codebases to be generally organized and well-written. However, we found that the codebases use the C++ assertion macro and recommend improving the process in which assumption checks are performed by implementing a more stable approach (Issue B).

Tests

The repositories in scope include sufficient test coverage.

Documentation and Code Comments

The project documentation provided for this review offers a sufficient overview of the system and its intended behavior. While the crypto-suites are sufficiently commented, most parts of the codebase have few code comments. In addition, our team found some inaccurate and misleading comments, which we recommend be updated (Suggestion 4).

Scope

The scope of this review was sufficient and included all security-critical components. However, since the Safeheron team updated the key recovery algorithm during the review and stated that they plan to produce a new security proof for it, we recommend that the updated key recovery algorithm and new security proof be comprehensively audited by an independent security firm familiar with the Safeheron codebase once development is complete.

After the initial review, an external security service provider identified a vulnerability in the `crypto-suites/crypto-sss` module and reported the security flaw to the Safeheron team. The Safeheron team then implemented a security patch to remediate the vulnerability, which our team reviewed and verified, as documented in this updated report (Issue E).

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: Length Field in CSafeHash256/512::Write Can Overflow	Resolved
Issue B: Sanity Check Assertions Are Compiled Away in Release Mode	Resolved
Issue C: Incorrect Cofactor Handling in PubKeyRecovery	Resolved

Issue D: Missing Checks in ECDSA Signature Verification	Resolved
Issue E: Missing Check in Feldman's Secret Sharing Allows for Threshold Escalation [Known Issue]	Resolved
Suggestion 1: Remove SHA1 and 3DES From Elliptic Curve Integrated Encryption Scheme	Resolved
Suggestion 2: Replace Elliptic Curve Integrated Encryption Scheme with Hybrid Public Key Encryption	Planned
Suggestion 3: Implement a Different Serialization for the Points at Infinity	Resolved
Suggestion 4: Correct Inaccurate Code Comments	Resolved
Suggestion 5: Improve Exception and Error Handling	Resolved
Suggestion 6: Complete Security Proof Draft	Resolved

Issue A: Length Field in CSafeHash256/512::Write Can Overflow

Location

`crypto-suites/crypto-hash/safe_hash256.cpp#L33`

`crypto-suites/crypto-hash/safe_hash512.cpp#L33`

Synopsis

The length fields that provide injectivity in `CSafeHash` can overflow, resulting in the same hash being produced for different sequences of writes.

Impact

This attack may lead to forged proofs, possibly undermining the security of the MPC-ECDSA signature protocol.

Preconditions

The attacker needs to be able to trick the prover to include a very long value in one of the hashes.

Feasibility

It is unlikely that this attack can be performed successfully outside of a lab setting.

Technical Details

`CSafeHash256` and `CSafeHash512` wrap hash functions and add a 32-bit length to each write. This is a common method for injectively encoding sequences of values that may have variable length. Injectivity means that any two sequences produce different encodings. In the edge case where a very large value (>4GiB) is supplied, the length field wraps around, allowing injectivity to break. Breaking injectivity allows hash collisions to be produced without the hash function itself being attacked.

Remediation

We recommend checking that the length of each write fits in 32-bits.

Status

The Safeheron team has expanded the length representation to 64-bit length, which makes it practically infeasible to overflow the length field.

Verification

Resolved.

Issue B: Sanity Check Assertions Are Compiled Away in Release Mode

Location

Examples (non-exhaustive):

`crypto-suites/crypto-bn/bn.cpp#L125`

`crypto-suites/crypto-bn/rand.cpp#L143`

`crypto-suites/crypto-sss/vsss.cpp#L25`

Synopsis

The C++ assertion macro is implemented for assumption checks throughout the codebase.

Impact

This might lead to unexpected behavior.

Technical Details

In C++, the `assert` macro is used for debugging purposes only, and its behavior depends on the presence of the NDEBUG macro. If NDEBUG is defined, the `assert` macro is compiled away. Developers often utilize the `assert` macro for optimization purposes by eliminating the negative impact assertion checks can have on performance.

In some instances in the codebase, however, the Safeheron team utilizes the `assert` macro to perform assumption checks on parameters in low-level algebraic and cryptographic primitives. These checks are hence missing if the NDEBUG macro is present, which is standard behavior in release builds.

Remediation

We recommend that the Safeheron team either build a custom `assert` macro expressing the same logic, or check the relevant assumption via exception handling.

Status

The Safeheron team has implemented a custom `assert` functionality and replaced the C++ `assert` macro with it.

Verification

Resolved.

Issue C: Incorrect Cofactor Handling in PubKeyRecovery

Location

`crypto-suites/crypto-curve/ecdsa.cpp`

Synopsis

In Safeheron's implementation of the ECDSA `PubKeyRecovery` algorithm, a parameter `j` is used in order to decide which of the four possible public keys from the key recovery process should be considered as valid. However, the implementation uses `j` incorrectly.

Impact

The two cases `j=2` and `j=3` might compute public keys that are invalid for the signature.

Preconditions

A private key would need to have an associated public key `pk=(x,y)`, such that `x` is larger than the modulus of the curve's scalar field.

Feasibility

The attack is straightforward since public keys are distributed uniformly. Hence, by utilizing a brute-force try and error generation of secret keys, it would be feasible to generate an associated public key that satisfies the precondition.

Technical Details

For parameter `2 <= j <= 3`, the Safeheron team computes the `x` coordinate of the curve point R as `x = r + curv->n * j`. However, the algorithm requires `x = r + curv->n`. Since `2*curve->n > curve->p`, the value `curve->n*j` exceeds the curve's base field modulus, which would result in unexpected consequences.

Mitigation

We recommend computing the `x` coordinate of R as `x = r + curve->n in case j=2 or j=3`.

Remediation

As Safeheron's implementation of the ECDSA key recovery algorithm can only handle curves with a cofactor of 1, and utilizes `j` inconsistently, we recommend implementing algorithm 4.1.6, as explained in [Brown09].

Status

The Safeheron team has replaced the parameter `j` with a more descriptive name and updated their elliptic curve representation to include the cofactor. They also updated the recovery algorithm to work as intended for elliptic curves with a cofactor of 1.

Verification

Resolved.

Issue D: Missing Checks in ECDSA Signature Verification

Location

`crypto-suites/crypto-curve/ecdsa.cpp#L122`

Synopsis

In addition to computing the verification equation, an ECDSA verifier must also compute some sanity checks on the input values. While the Safeheron team checks that the coordinates of the public key are reduced and that they satisfy the curve's defining equation, other sanity checks are missing.

Impact

In the absence of such sanity checks, an attacker can forge signatures.

Technical Details

Our team identified the following missing checks on the verifier's input:

- a check verifying that the signer's public key is not the point at infinity;
- a check verifying that the public key is in the large prime order subgroup (note that this is not needed for `secp256k1`, P256 or STARK); and
- a check verifying that the signature values are reduced (i.e., verifying that `r<curve->n` and `s<curve->n`).

Remediation

We recommend implementing the missing checks.

Status

The Safeheron team has implemented the missing checks.

Verification

Resolved.

Issue E: Missing Check in Feldman's Secret Sharing Allows for Threshold Escalation [Known Issue]

Location

`/src/crypto-suites/crypto-sss/vsss.cpp#L32`

Synopsis

During the audit, an external security service provider reported a vulnerability to the Safeheron team. This vulnerability occurs because a test is missing where an honest participant checks that the degree of each random polynomial in Feldman's verifiable secret sharing scheme does not exceed the agreed on threshold `t` of the protocol.

Impact

If a malicious participant generates a commitment to a Feldman random polynomial of degree `T > t` for an agreed on threshold `t`, and the degree is not checked by all other honest participants, it effectively transforms the `t` out of n protocol into a T out of n protocol. As a result, no set of `t` participants will be able to generate a valid signature. In case the attacker choses T>n, it is not possible to generate a valid signature at all.

Feasibility

The attack is straightforward since it is trivial to generate random polynomials of arbitrary degrees.

Remediation

The `VerifyShare` function in Safeheron's implementation of Feldman's verified secret sharing scheme needs to implement a check on the number of coefficient commitments for the shared random polynomial of each participant.

Status

The Safeheron team has implemented the recommended check.

Verification

Resolved.

Suggestions

Suggestion 1: Remove SHA1 and 3DES From Elliptic Curve Integrated Encryption Scheme

Location

`src/crypto-suites/crypto-ecies`

Synopsis

While neither SHA1 nor 3DES is broken in the specific contexts utilized for this implementation, they are not recommended foundations to build on, as they are susceptible to a high number of attacks.

Mitigation

We recommend adhering to best practice and removing support for `SHA1` and `3DES`.

Status

The Safeheron team has removed support for both SHA1 and 3DES, as recommended.

Verification

Resolved.

Suggestion 2: Replace Elliptic Curve Integrated Encryption Scheme with Hybrid Public Key Encryption

Location

`src/crypto-suites/crypto-ecies`

Synopsis

The Elliptic Curve Integrated Encryption Scheme (ECIES) is a family of non-interoperable hybrid encryption standards. In order to improve the encryption scheme used, Hybrid Public Key Encryption (HPKE) was standardized in RFC 9180. In the motivation section of the RFC, the authors explain:

Currently, there are numerous competing and non-interoperable standards and variants for hybrid encryption, mostly variants on the Elliptic Curve Integrated Encryption Scheme (ECIES), including ANSI X9.63 (ECIES) [ANSI], IEEE 1363a [IEEE1363], ISO/IEC 18033-2 [ISO], and SECG SEC 1 [SECG]. See [MAEA10] for a thorough comparison. All these existing schemes have problems, e.g., because they rely on outdated

primitives, lack proofs of indistinguishable (adaptive) chosen-ciphertext attack (IND-CCA2) security, or fail to provide test vectors.

Mitigation

We recommend replacing ECIES with HPKE, as defined in RFC 9180.

Status

The Safeheron team has stated that they plan to implement HPKE in the future.

Verification

Planned.

Suggestion 3: Implement a Different Serialization for the Points at Infinity

Location

crypto-suites/crypto-curve/openssl_curve_wrapper.cpp#L35

Synopsis

The Safeheron team serializes the points at infinity of an elliptic curve to a zero-byte array with the header `0x04` for an uncompressed representation and the header `0x02` for a compressed representation. In the uncompressed case, this is safe only for short Weierstrass elliptic curves that do not have `(x,y)=(0,0)` as a solution and in the compressed case, it is safe only for short Weierstrass curves that do not have `(y,0)` in their solution set.

Mitigation

While neither one of the curves `secp256k1`, P256, and STARK has `(0,0)` or `(y,0)` as curve points, we recommend implementing a different representation for the point at infinity. One solution would be to include two new headers designated to that point, in compressed and uncompressed forms.

Status

The Safeheron team has updated their codebase, such that it now refrains from serializing the point at infinity.

Verification

Resolved.

Suggestion 4: Correct Inaccurate Code Comments

Location

crypto-suites/crypto-paillier/pail.cpp#L86

crypto-suites/crypto-paillier/pail.cpp#L95

crypto-suites/crypto-paillier/pail.cpp#L104

Synopsis

In the aforementioned locations, the code comments note that a Paillier Key Pair should be created with a 1024-bit key size. However, the functions reference varying key sizes (2048-bit, 2072-bit, and 2096-bit).

Mitigation

We recommend correcting the code comment and checking that all comments are accurate and relevant.

Status

The Safeheron team has corrected and improved the code comments.

Verification

Resolved.

Suggestion 5: Improve Exception and Error Handling

Location

crypto-suites/crypto-bn/bn.cpp#L985

crypto-suites/crypto-paillier/pail_pubkey.cpp#L49

Synopsis

While the Safeheron team has implemented extensive error and exception handling in general, our team identified a few instances where edge case behavior was not handled as expected. For example, the square root function does not throw an error but computes the root of negative numbers to be zero, which is unexpected behavior.
In addition, the Paillier encryption algorithm does not check if the input message `m` is in the range `0<=m<n`, which could lead to incorrect signatures.

Mitigation

We recommend improving error and exception handling.

Status

The Safeheron team has improved error and exception handling.

Verification

Resolved.

Suggestion 6: Complete Security Proof Draft

Location

Modified Key Generation and New Key Recovery Protocol; Security Proof

Synopsis

Within the area of provable security, when new cryptographic algorithms are introduced, formal definitions for their respective privacy and security are modeled. According to these definitions, the new algorithms are then proven to be secure. The security definitions are typically derived either through a game-based approach or a simulation-based approach. This was the process for the distributed key generation and threshold signing schemes, as described in [CGG+21]. The distributed key generation and threshold signing schemes in [CGG+21] were proven to be secure in the Universal Composability setting.

The Safeheron team has introduced a new key recovery algorithm based on Shamir Secret Sharing and the Diffie-Hellman key exchange as well as the updates implemented to the key generation algorithm described in [CGG+21] (to match the key recovery algorithm). The team has additionally provided a document with a security proof.

The provided document, however, is missing a security definition for the key recovery functionality in addition to a formal proof. A security definition for the key recovery functionality increases the coverage of the relevant attacks being modeled. This is specifically important in regard to the composability of the subprotocols. Due to the nature of threshold signing sessions and distributed key generation, the concurrent security needs to be taken into account for modeling the protocol. As an example, whether it is possible to recover keys across different signing sessions is unclear in this setting. The proof, therefore, is incomplete and, as a result, does not provide a full picture of the security and privacy of the protocol at this stage.

During our review, the Safeheron team also updated the new key recovery protocol, thus necessitating an additional review in the future. Consequently, a new security proof will also be required in this case.

Mitigation

We recommend adding a security definition and formal proof for the key recovery functionality. This can be done either in the Universal Composability framework as in [CGG+21], or in a standalone manner. Note that if the latter approach is adopted, the composability of the new key recovery functionality with respect to the threshold signing functionality from [CGG+21] is not given.

Status

The Safeheron team has updated the document entailing the changes and improved the security proof draft. Additionally, the Safeheron team has provided a security definition and corresponding proof. However, we were unable to verify its correctness during the verification period, as this would require a detailed analysis of the security requirements of the Shard privacy definition. Therefore, we recommend having the security definition verified by an independent third-party team.

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.