Least Authority
PRIVACY MATTERS

FROST Demo
Security Audit Report

ZCG

Final Audit Report: 29 April 2025

Table of Contents

Overview

Background

ZCG has requested that Least Authority perform a security audit of the FROST server and client components. The project implements a demo that illustrates how to use the FROST protocol based on the `frost-crate`. The FROST server helps participants and coordinators communicate with each other, and The FROST client is a CLI tool that demonstrates how to interact with the server and the `frost-crate`.

Project Dates

- **February 3, 2025 - February 12, 2025:** Initial Code Review *(Completed)*
- **February 13, 2025:** Delivery of Initial Audit Report *(Completed)*
- **April 29, 2025:** Verification Review *(Completed)*
- **April 29, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Jasper Hepp, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor and Writer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the FROST server and client components 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:

- Frost-server:
  https://github.com/ZcashFoundation/frost-zcash-demo/tree/main/frostd
- Frost-client:
  https://github.com/ZcashFoundation/frost-zcash-demo/tree/main/frost-client
- Dependencies:
  - https://github.com/ZcashFoundation/frost-zcash-demo/tree/main/coordinator
  - https://github.com/ZcashFoundation/frost-zcash-demo/tree/main/dkg
  - https://github.com/ZcashFoundation/frost-zcash-demo/tree/main/participant
  - https://github.com/ZcashFoundation/frost-zcash-demo/tree/main/trusted-dealer

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

- `548a8a7329c6eed8180464662f430d12cd71dfcc`

For the verification, we examined the Git revision:

- `548a8a7329c6eed8180464662f430d12cd71dfcc`

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

- frost-zcash-demo:
  https://github.com/LeastAuthority/frost-zcash-demo

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:

- Zcash Foundation GitHub:
  https://github.com/ZcashFoundation/frost-zcash-demo
- FROST Documentation:
  https://frost.zfnd.org

In addition, this audit report references the following documents:

- C. P. L. Gouvêa and C. Komlo, "Re-Randomized FROST." *IACR Cryptology ePrint Archive*, 2024, [GK24]
- C. Komlo and I. Goldberg, "FROST: Flexible Round-Optimized Schnorr Threshold Signatures." *IACR Cryptology ePrint Archive*, 2020, [KG20]
- RFC 9591:
  https://datatracker.ietf.org/doc/rfc9591
- Memory Zeroization:
  https://docs.rs/zeroize/latest/zeroize/index.html#
- `frost` crate:
  https://github.com/ZcashFoundation/frost

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 passed correctly 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

The code in scope serves as a demonstration of how to use the `frost-crate`. The `frost-crate` is an implementation for a threshold Schnorr signatures scheme called FROST (RFC 9591, [KG20]). The `frost-demo` allows a user to locally mimic a key generation setup via a trusted dealer or a distributed key generation protocol (DKG) as well as the FROST signing protocol. A server allows secure and authenticated communication between the parties using HTTP, TLS (optional), and encryption of messages via the noise protocol.

System Design

Server

We reviewed the server and its functionality and found that the `send` and `receive` functions do not properly validate messages (Issue D). Additionally, we identified that HTTP error handling can be improved (Suggestion 7) and further recommend abstracting and unifying the entire HTTP communication struct, including its encryption and decryption functions (Suggestion 2).

Trusted Dealer

We reviewed the trusted dealer against the specification in RFC 9591 (Appendix C) and found that it does not check whether all participants received the same VSS commitment (Issue C).

Distributed Key Generation

We reviewed the DKG implementation against the specification in [KG20] to evaluate its proper usage of the `frost-crate`. We did not find any issues within this context.

FROST Signing Protocol

We reviewed the coordinator and the participants in the context of the signing protocol. In particular, we reviewed the implementation against the specification in RFC 9591 to evaluate its proper usage of the `frost-crate`.

We found that a user can overwrite contacts when importing a new contact, which might be exploitable in a spoofing attack (Issue E). We also found that the participants trust the coordinator to use the expected message in the signing process (Issue F). Additionally, we suggest introducing a trait that requires users to implement protocol-specific message verification (Suggestion 1).

Our team reviewed the rerandomization extending RFC 9591. This demo uses the `rerandomize-crate` based on [GK24]. We note that it implements the version in which a centralized party selects the randomization factor, rather than the alternative design described in Section `5.1` [GK24]. While we did not identify any issues with the overall usage of rerandomized FROST, we still suggest improving the robustness of the `rerandomize` argument (Suggestion 4).

We observed that `share refreshing` and `identifiable abort` have not yet been implemented. In addition, we found that the demo currently handles message vectors that contain a single message only and hence does not allow signing several messages in parallel within one invocation of the signing protocol. Furthermore, RFC 9591 allows for a Single Coordinator or Coordinator-Less Deployments, while the code only implements the Single Coordinator version.

Handling of Secure Data

The implementation does not properly manage secret data stored in memory. We recommend zeroizing all secret data that has been used (Issue B). In addition, we noted that the secret key of the user is stored in an unencrypted local `config` file. Although we do not consider this to be an issue since the implementation is for demo purposes only, we still recommend encrypting it (Suggestion 6).

Some functions, such as `print_signing_package`, log secret data (e.g., commitments). While we do not consider this to be an issue since the implementation is for demo purposes only, our team advises against logging secret data in production.

Dependencies

We examined the dependencies implemented in the codebase and identified two vulnerabilities. We recommend upgrading the two dependencies to their recommended replacements (Issue A).

Code Quality

We performed a manual review of the repositories in scope and found the code to be well-organized and in adherence to Rust best practices. However, we recommend renaming certain functions in the participant and coordinator components to improve clarity (Suggestion 3).

Tests

The `frost-crate` includes test coverage. Note that our team did not assess whether test coverage was sufficient, as the tests were out of the scope of this audit. However, we ran the demo and tested its functionality based on the documentation.

Documentation and Code Comments

The project documentation provided for this review offers a sufficient overview of the system and its intended behavior. In particular, `frost-demo` is documented extensively with comprehensive READMEs and relevant descriptions that facilitate understanding the code. Additionally, the codebase is well-commented with accurate inline comments, which was helpful in understanding the intended functionality of most of the components.

Scope

Given that the scope of this review was limited to the HTTP trait implementations, our team considered it sufficient. Note that the CLI and socket versions were excluded from the scope of this audit, as agreed upon with the FROST demo team. Additionally, our team assumed that the `frost-crate` dependencies behave as intended and comply with the FROST specification.

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: Usage of Vulnerable Dependencies	Resolved
Issue B: No Zeroization of Secret Data	Resolved

Issue C: Missing VSS Commitment Verification by Participants in Trusted Dealer	Resolved
Issue D: Missing Checks in Send and Receive Functions of the Server	Resolved
Issue E: Import Allows Overwrite of Contacts	Resolved
Issue F: Participants Act as Signing Oracles	Resolved
Suggestion 1: Introduce Protocol-Specific Message Verification	Implemented
Suggestion 2: Abstract and Unify the Encrypt / Decrypt Functions for All HTTP Files	Implemented
Suggestion 3: Rename Functions and Variables in Participants and Coordinator for Improved Clarity	Implemented
Suggestion 4: Add Randomizer Sanity Check To Improve Robustness	Implemented
Suggestion 5: Improve Handling of Excessively Large Messages During Encryption / Decryption	Implemented
Suggestion 6: Only Save Encrypted Secrets to File	Partially Implemented
Suggestion 7: Improve HTTP Error Handling	Implemented

Issue A: Usage of Vulnerable Dependencies

Location

`Cargo.toml`

`Cargo.lock`

Synopsis

Analyzing the codebase with `cargo audit` for dependency versions shows two vulnerabilities.

Impact

Using unmaintained dependencies or packages with known vulnerabilities may lead to critical security vulnerabilities in the codebase.

Technical Details

The following dependency vulnerabilities were identified:

```
Crate:    idna
Version:  0.5.0
Title:    `idna` accepts Punycode labels that do not produce any non-ASCII when decoded
Date:     2024-12-09
ID:       RUSTSEC-2024-0421
```

URL:	https://rustsec.org/advisories/RUSTSEC-2024-0421
Solution:	Upgrade to `>=1.0.0`

Crate:	`openssl`
Version:	`0.10.66`
Title:	`ssl::select_next_proto` use after free
Date:	`2025-02-02`
ID:	RUSTSEC-`2025-0004`
URL:	https://rustsec.org/advisories/RUSTSEC-2025-0004
Solution:	Upgrade to `>=0.10.70`

Remediation

We recommend upgrading the two dependencies.

Status

The FROST demo team has resolved the issue and removed the `OpenSSL` dependency.

Verification

Resolved.

Issue B: No Zeroization of Secret Data

Location

Examples (non-exhaustive):

`participant/src/cli.rs#L47`

`frost-client/src/trusted_dealer.rs#L64`

`dkg/src/cli.rs#L72`

Synopsis

Secret data, such as the nonce or the secret key, is not erased from memory and could be leaked.

Impact

This issue could potentially result in the full disclosure of secrets.

Preconditions

Root access to the machine is required, enabling the reading of process memory from other processes.

Severity

Low.

Technical Details

As highlighted in RFC 9591 and other related documentation, secret data, such as the nonce, must be deleted after it has been used. More specifically:

- For the nonce and commitment in RFC 9591, Section `5.2` states: "*Each participant **MUST** delete the nonce and corresponding commitment after completing sign.*"
- For the trusted dealer key generation in RFC 9591, Appendix C states: "*delete secret values after distributing shares to each participant*" and "*The trusted dealer **MUST** delete the `secret_key` and `secret_key_shares` upon completion.*"

- For the DKG protocol described in [KG20], in Figure 1, the paper states that the proofs of knowledge `sigma_i` from Round 1, Step 5, as well as the polynomial evaluations from Round 2, Step 3, should be deleted.

Currently, secret data is stored in memory, while it is likely overwritten during normal code execution. This is neither a best practice nor a secure method for deleting secret data. Instead, zeroization should be implemented.

Remediation

We recommend utilizing memory zeroization of all sensitive values.

Status

The FROST demo team has addressed this issue using Rust's `zeroize` crate but noted that a more robust solution will require improvements to `frost-core`, which they intend to implement in the future.

Verification

Resolved.

Issue C: Missing VSS Commitment Verification by Participants in Trusted Dealer

Location

`frost-client/src/trusted_dealer.rs#L64`

Synopsis

The FROST demo implementation deviates from RFC 9591 by omitting the verification of the VSS (Verifiable Secret Sharing) commitment. According to RFC 9591 (Appendix C), after receiving shares from the trusted dealer, participants must verify that they received the same VSS commitment.

Impact

Without the check, a compromised dealer could introduce inconsistent secret shares. This could undermine the integrity of the generated keys and, by extension, the security of the signing protocol.

While the demo's context does not pose any risks, a production-level adaptation relying on this approach would be vulnerable to subtle but critical misconfigurations and attacks.

Severity

Medium.

Technical Details

The FROST specification in RFC 9591 (see Appendix C) requires that each participant, upon receiving their share, verify the accompanying VSS commitment. In the current implementation, the trusted dealer does not send individual messages but rather embeds the share and commitment data in the configuration files of each participant. This is acceptable in a demo setting. Nevertheless, the participant's check should be included for demonstration purposes.

Remediation

We recommend implementing a check to verify that each participant receives the same VSS commitment in the function `trusted_dealer_for_ciphersuite`, after line L65. In addition, we recommend adding a code comment highlighting that this check should be performed by each participant in a production-ready application.

Status

The FROST demo team argued convincingly that the suggested check would not be meaningful in its current form because commitment verification only occurs after the `config` files are received, and this process takes place outside the tool; furthermore, the necessary commitments are not included in the `config` file. The team also noted that adding the check for reference would be impractical, as it relies on a broadcast channel, which is complex to configure. Instead, the FROST demo team chose to include warnings in the command-line help and within the source code.

Our team agrees with the development team's response and thus considers this issue resolved.

Verification

Resolved.

Issue D: Missing Checks in Send and Receive Function of the Server

Location

`frostd/src/functions.rs`

Synopsis

The FROST demo server lacks proper validation in its message handling functions. Specifically, the `send` and `receive` functions do not verify whether the provided `user.pubkey` is a member of the current session. Additionally, the `send` function does not check whether the intended recipients are included in the session's list of authorized public keys. Together, these oversights allow any user—even those not part of the session—to send and receive messages as if they were legitimate session participants.

Impact

An attacker can exploit the missing checks to inject unauthorized messages. However, the encryption of messages via the noise protocol does not allow an attacker to send messages to the participants of the session since the decryption would fail.

Preconditions

An attacker would need to know the session ID.

Severity

Low.

Remediation

We suggest implementing a session membership check in the `send` and `receive` functions. We suggest also checking whether the recipients are part of the session.

Status

The FROST demo team has resolved this issue.

Verification

Resolved.

Issue E: Import Allows Overwrite of Contacts

Location

`frost-client/src/contact.rs#L58`

Synopsis

The function `import` allows overwriting existing contacts.

Impact

This issue increases the risk of spoofing attacks.

Severity

Medium.

Technical Details

Contacts are stored in a map. Adding a new entry with an existing key overwrites the previous value. For contacts, the key is the name and the value is the public key.

An attacker could use this to overwrite an existing contact of an honest participant using a public key that they control. The attacker could then trick the honest participant to participate in a key generation and signing protocol with a forged public key, thereby executing a type of spoofing attack.

Remediation

We recommend adding a check during import to prevent overwrites. The `remove` functionality could be used if a user needs to change an existing contact.

Status

The FROST demo team has resolved this issue and added a check to prevent multiple contacts from using the same public key.

Verification

Resolved.

Issue F: Participants Act as Signing Oracles

Location

`src/comms/http.rs#L308`

Synopsis

The participants trust the coordinator to use the expected message in the signing process, as there is currently no way for participants to validate the expected content of a message. The FROST demo team has already identified a similar instance here.

Impact

The extent of the impact depends on the specific application.

Remediation

One potential approach to resolving this issue is to tie `session_ids` to `message-hashes`, such that each participant can verify that the message hash is related to the session ID. We also suggest referring to the recommendations outlined in RFC 9591 (Section 7.6) for hashing the message.

Status

The FROST demo team has implemented a prompt that displays the message to be signed, allowing the user to explicitly consent. The team additionally noted that the proposed remediation is not applicable, as the tool must remain compatible with the specific protocol the user is interacting with, which requires signing the exact bytes supplied by the user.

Verification

Resolved.

Suggestions

Suggestion 1: Introduce Protocol-Specific Message Verification

Location

`src/comms/http.rs#L308`

Synopsis

In Round 2, the coordinator sends the message to the participant, but the participant does not verify its protocol-specific structure. This deviates from the recommendations documented in RFC 9591 (Section 7.7) and the `frost-crate` (here).

Mitigation

We recommend defining message verification as a trait requiring users to implement protocol-specific message verification. We also suggest message hashing, as recommended in RFC 9591 (Section 7.6).

Status

The FROST demo team stated that, because the tool lacks awareness of the specific protocol invoking it, it is unable to perform message verification itself. However, they emphasized the importance of displaying the message to the user and prompting them to confirm their intention to sign the message. To address this, they introduced such a prompt along with a trait method similar to the one proposed by our team. In the context of Zcash, this will later be extended to display the transaction plan of the transaction being signed. At that stage, the FROST demo team intends to introduce a Content-Type mechanism to help users understand the nature of the data being signed.

Verification

Implemented.

Suggestion 2: Abstract and Unify the Encrypt / Decrypt Functions for All HTTP Files

Location

`src/comms/http.rs#L117`

src/comms/http.rs#L370

src/comms/http.rs#L276

Synopsis

The DKG component, the participant, and the coordinator each implement the same struct, `HTTPComms`, which includes an encryption and decryption function for messages received via the server.

Mitigation

Since the code is the same across the three implementations, we recommend unifying this code for better abstraction and improved error resilience.

Status

The FROST demo team has implemented the mitigation in [PR#495] and [PR#496].

Verification

Implemented.

Suggestion 3: Rename Functions and Variables in Participants and Coordinator for Improved Clarity

Location

participant/src/cli.rs#L27

Synopsis

Certain functions and variable names are not intuitive and decrease the readability of the code. This includes, in particular, the functions responsible for the signing protocol on both the participant and coordinator side. The participant side progresses in rounds (1 and 2), while the coordinator progresses in steps (1, 2, and 3). However, the connection between the two communicating parties is not immediately clear. In addition, the function `get_signature_shares` called by the coordinator first sends signing packages and then receives signature shares in the second step.

Mitigation

We suggest unifying the rounds and steps into one consistent setup for the coordinator and participant. We additionally recommend renaming the function `get_signature_shares` to `send_signing_packages_and_get_signature_shares` or a similar alternative.

Status

The FROST demo team has implemented the mitigation as recommended.

Verification

Implemented.

Suggestion 4: Add Randomizer Sanity Check To Improve Robustness

Location

frost-client/src/args.rs#L173

Synopsis

The randomizer as an argument input to the command of the coordinator is implemented as a vector of strings. This vector should have the same length as the message vector, but no check is implemented in the code to verify this. However, since the code does not support signing multiple messages simultaneously, this does not result in a security-relevant issue.

Mitigation

We recommend changing the type to `Option<Vec<String>>` and adding a check to verify that if the option is `SOME`, the length of the randomizer matches the length of the messages passed in as an argument.

Status

The FROST demo team has implemented the mitigation as recommended.

Verification

Implemented.

Suggestion 5: Improve Handling of Excessively Large Messages During Encryption / Decryption

Location

`src/comms/http.rs#L306`

`src/comms/http.rs#L460`

Synopsis

In order to prevent denial-of-service (DoS) attacks, a message from the server must have a size of less than `65535` bytes in the functions `encrypt` and `decrypt` for the coordinator, participant, and DKG protocol. If the message exceeds this bound, the code of the coordinator (for example) would abort in L460 of the function `recv` when the coordinator receives a message from the participant. Hence, a malicious participant can end the coordinator process by sending a message that is too large to decrypt.

Mitigation

We recommend implementing a different approach for handling messages exceeding a certain size. Instead of throwing an error, the code should be modified to ignore large messages.

Status

The FROST demo team stated that FROST is not a robust protocol—participants can always prevent a signing session from succeeding. Accordingly, they do not attempt to prevent all possible forms of session abortion, as participants can disrupt sessions in various ways, such as by sending invalid encrypted messages. Nonetheless, they implemented a message limit check on the server to help mitigate memory exhaustion, as message length would otherwise only be validated upon decryption. They also introduced session cleanup mechanisms for the coordinator and DKG starter roles to prevent aborted sessions from persisting in the event of protocol errors.

Verification

Implemented.

Suggestion 6: Only Save Encrypted Secrets to File

Location

frost-client/src/config.rs

Synopsis

The FROST client currently reads sensitive data from configurations, including secrets from a `config` file, which is not encrypted.

Mitigation

While unencrypted secrets might be acceptable for a demo version, we still recommend refraining from storing secrets in plaintext. A more appropriate approach would be to encrypt this data and prompt the user for a password during login.

Status

The FROST demo team acknowledged the importance of this suggestion but stated that addressing it will require careful design to maintain interoperability with other applications. They have scheduled the planned improvements for a future update and, in the meantime, will include warnings about the issue in the documentation.

Verification

Partially Implemented.

Suggestion 7: Improve HTTP Error Handling

Location

src/comms/http.rs#L496

Synopsis

In the following lines of code, `_r` is not checked. Consequently, if the server responds with HTTP `400` or `500`, the code will not detect it:

```
let _r = self.client
          .post(format!("{}/send", self.host_port))
          .json(&frostd::SendArgs { ... })
          .send().await?;
```

Mitigation

One possible solution could be to implement the following:

```
let resp = self.client.post(...).send().await?;
if !resp.status().is_success() {
    return Err(eyre!("send failed: {}", resp.status()).into());
}
```

Status

The FROST demo team has implemented the mitigation as recommended.

Verification

Implemented.

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.