

Aave Family Accounts

Security Assessment and Fix Review

November 12, 2026

Prepared for:
Adam Schoeman
Aave

Prepared by: **Fredrik Dahlgren and Graham Sutherland**

Table of Contents

Table of Contents **1**
Project Summary **3**
Executive Summary **4**
Project Goals **7**
Project Targets **9**
Project Coverage **10**
Automated Testing **11**
Summary of Findings **13**
Detailed Findings **15**
1. Refresh and access tokens are not invalidated when credentials change 15
2. Web socket unconditionally skips device metadata checks 20
3. Unauthenticated JSON-RPC proxy for Alchemy has no rate limiting or RPC method restrictions 23
4. Passkey replay protection can be strengthened 26
5. Unauthenticated CoinCover public keys allow server-side account recovery 29
6. Unauthenticated client public keys allow server-side account recovery 31
7. A malicious Shamir secret-share holder can choose the recombined secret key 34
8. Swift SDK persists JWTs without enforced data protection 36
9. Two-factor disable attempts are not rate-limited 38
10. Race condition in credential removal database query could lead to permanent account lockout 40
11. Race condition in password addition could lead to permanent account lockout 43
12. SDK release workflow executes dependency lifecycle scripts while holding GitHub Packages publish token 47
13. Production CI runner installs unpinned Vercel package 50
14. Self-hosted CI runner shared across PR-triggered and production-deploy workflows 52
15. IDOR in editWalletMetadata allows cross-user edits to wallet metadata 55
16. Third-party GitHub actions not pinned to commit hash 58
17. Minimum element size clickjacking guard can be bypassed with CSS transforms 59
18. Password normalization is different across platforms 61
19. Offline recovery master key derivation is different across platforms 63
A. Vulnerability Categories **65**

B. Automated Analysis Tool Configuration **67**
C. FCA SCA Gap Analysis **70**
D. Code Quality Issues **72**
E. Fix Review Results **73**
Detailed Fix Review Results 75
F. Fix Review Status Categories **79**
About Trail of Bits **80**
Notices and Remarks **81**

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Application Security and Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Graham Sutherland, Consultant
graham.sutherland@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 1, 2026	Pre-project kickoff call
June 9, 2026	Status update meeting #1
June 15, 2026	Status update meeting #2
June 25, 2026	Delivery of report draft
June 25, 2026	Report readout meeting
July 1, 2026	Fix review
July 8, 2026	Delivery of final comprehensive report
October 31, 2026	Follow-up fix review
November 12, 2026	Delivery of final comprehensive report with fix review

Executive Summary

Engagement Overview

Aave engaged Trail of Bits to review the security of Aave Family Accounts. A team of two consultants conducted the review from June 1 to June 18, 2026, for a total of four engineer-weeks of effort.

Our testing efforts focused on JWT handling, passkey and two-factor authentication flows, CoinCover recovery, cryptographic key derivation and Shamir secret-sharing, dApp and iframe interaction boundaries, transaction approval protections, Financial Conduct Authority strong customer authentication (FCA SCA) alignment, and SDLC controls. We performed manual source-code review, dependency and workflow analysis, targeted testing, and automated analysis using CodeQL, Jest, npm audit, Semgrep, and Zizmor.

From June 31 to July 1 and on October 31, 2026, Trail of Bits reviewed fixes for the issues reported during the review. The results of this fix review are described in appendix E.

Observations and Impact

Aave Family Accounts includes several security-oriented controls, including multiple authentication and account recovery mechanisms. The findings show that several of these controls need stronger enforcement at account recovery, session management, dApp interaction, and deployment trust boundaries.

The highest-risk findings affect session persistence and account recovery. Refresh tokens and access tokens remain valid after credential changes, allowing an attacker with an existing refresh token to retain access even after a user changes passwords, removes credentials, or enables two-factor authentication (TOB-AAVEFAM-1). The CoinCover recovery flow also does not authenticate the public keys used during setup and recovery, which means that a compromised Family Accounts server could substitute attacker-controlled keys, obtain the recovery share, and recover account master key material (TOB-AAVEFAM-5, TOB-AAVEFAM-6).

The remaining findings essentially constitute defense-in-depth gaps and opportunities for operational hardening. Web socket requests bypass device metadata checks (TOB-AAVEFAM-2), two-factor disable attempts are not rate-limited (TOB-AAVEFAM-9), credential change database operations contain race conditions that can lock users out of their accounts (TOB-AAVEFAM-10, TOB-AAVEFAM-11), and `editWalletMetadata` allows cross-user wallet metadata edits when an attacker knows a wallet ID (TOB-AAVEFAM-15). The JSON-RPC proxy exposes Aave's Alchemy usage to unauthenticated abuse (TOB-AAVEFAM-3), and the integrated clickjacking guard can be bypassed with CSS transforms (TOB-AAVEFAM-17).

The CI/CD findings flagged by Zizmor identify ways that dependency lifecycle scripts, unpinned packages, shared self-hosted runners, and mutable third-party action references could expose package publishing or production deployment credentials (TOB-AAVEFAM-12, TOB-AAVEFAM-13, TOB-AAVEFAM-14, TOB-AAVEFAM-16). The FCA SCA gap analysis also identified technical gaps around mandatory two-factor authentication, inactivity timeouts, compromised-password checks, and strong authentication for trusted beneficiary changes.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Aave take the following steps:

- **Remediate the findings disclosed in this report.** Prioritize the findings that affect account compromise, recovery-key protection, and credential changes. These findings should be addressed through direct fixes or broader refactoring efforts.
- **Strengthen session and credential lifecycle controls.** Invalidate sessions after credential and two-factor authentication changes, add a user-visible mechanism for revoking all sessions and trusted devices, enforce device checks consistently across HTTP and web socket requests, and require reauthentication for sensitive account and recovery operations.
- **Clarify and enforce the recovery trust model.** Authenticate CoinCover and client public keys before using them for recovery-share encryption, document the trust assumptions for server-side and third-party recovery flows, and add regression tests for key-substitution and share-manipulation scenarios.
- **Harden externally reachable and dApp-controlled surfaces.** Add authentication, rate limits, method restrictions, and weighted request limits to the JSON-RPC proxy. Move approval-required dApp flows to a popup context or another browser context that the dApp cannot visually transform or obscure.
- **Improve database consistency for credential changes.** Add database-level uniqueness constraints and account-level locking for credential add and remove operations so concurrent requests cannot violate authentication invariants or leave accounts unusable.
- **Separate and constrain CI/CD trust zones.** Isolate PR-triggered and production-deployment workflows onto separate ephemeral runner pools, pin GitHub Actions to commit hashes, pin deploy tooling or run it from digest-pinned images, disable dependency lifecycle scripts during privileged installs, and scope publish and deploy credentials to the jobs that require them.
- **Expand cross-platform regression test coverage.** Add shared test vectors for password normalization, recovery master key derivation, Shamir share validation,

and SDK storage behavior. Increase focused unit coverage for security-sensitive error paths and edge cases that end-to-end tests may not exercise.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	7
Low	8
Informational	1
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Authentication	2
Configuration	4
Cryptography	3
Data Exposure	1
Data Validation	1
Denial of Service	3
Session Management	2
Timing	2

Project Goals

The engagement was scoped to provide a security assessment of the Aave Family Accounts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the JWT service reject JWTs with unexpected algorithm types?
- Is passkey authentication implemented correctly?
- Does the CoinCover integration conform to CoinCovers API documentation?
- Can a malicious dApp exploit the well-known transaction flow to bypass user interaction?
- Are server-side well-known transaction checks sufficiently strict, and does the implementation fall back to require user approval for unhandled cases?
- Are cryptographic keys and IVs generated securely across the implementation?
- Is HKDF used correctly to derive child keys?
- Are the 2-of-2 Shamir recovery shares protected against account compromise if either share is exposed to an attacker?
- Are Shamir share integrity checks in the threshold recovery flow sufficient to detect tampering?
- Does the codebase rely on any insecure or outdated cryptographic dependencies?
- Does the codebase conform to cryptographic engineering best practices?
- Can appOrigin be spoofed during auth or RPC popup initialization to redirect the AccountMasterKey or session data to an attacker-controlled window?
- Can a malicious dApp trigger `InjectedMasterController`, `AuthPopupController`, or `RpcPopupController` to return sensitive session material to an unauthorized recipient?
- Can `requireMasterIframe`'s same-origin frame enumeration be exploited, for instance by a dApp injecting a same-origin FA iframe, to intercept session data proxied to the master iframe?
- Can `sessionRetrieveListener` or `sessionDataListener` be reached from an unauthorized origin, bypassing the `FAMILY_ACCOUNTS_ORIGIN` and `appOrigin` checks?

- Can the `IntersectionObserver` v2 availability check be bypassed to enable the integrated signing modal without click-jacking protection active?
- Can the `ClickjackGuard` component be defeated, through element overlay, resize, or visibility-state manipulation, to trick users into approving transactions they cannot see?
- Are there improvements to the CI/CD or SDLC?
- Does the implementation comply with the FCA SCA specification?

Project Targets

The engagement involved reviewing and testing the following target.

Aave Accounts

Repository	https://github.com/family/family-account
Version	`f925d49d8219c8d362be4357faebc36b43529ecb`
Type	TypeScript, Swift
Platform	Web

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Manual review.** We reviewed the TypeScript and Swift source code for the Aave Family Accounts implementation. This review focused on authentication and account recovery flows, account and wallet access controls, server-side state management, dApp-origin validation, popup and iframe message routing, well-known transaction approval logic, cryptographic key generation and derivation, and CI/CD workflows.
- **Static analysis.** We built CodeQL databases for the TypeScript SDK, server, and Swift SDK, and analyzed them with public and Trail of Bits' private query suites. We also ran Semgrep with public and internal rule sets.
- **Test coverage.** We used Jest to review unit test coverage for the TypeScript SDK and server.
- **Dependency analysis.** We ran `npm audit` to identify JavaScript and TypeScript dependencies with known vulnerabilities. We also performed a best-effort review of key dependencies like the JWT library used to authenticate client devices, and the Shamir secret sharing implementation used in the CoinCover account recovery flow.
- **Workflow and SDLC analysis.** We ran Zizmor against the repository's GitHub Actions workflows and reviewed workflow trust boundaries, credential scope, dependency-install behavior, action pinning, runner isolation, package publication, and production deployment paths.
- **Specification and integration review.** We compared relevant authentication and recovery behavior against CoinCover's public ReCOVER documentation and performed a best-effort review of the implementation against the technical requirements specified by the Financial Conduct Authority's Strong Customer Authentication (FCA SCA) guidelines.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The FCA SCA gap analysis focused on clear deviations from the referenced specifications, and should not be treated as a full FCA SCA compliance assessment.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix B.1
Jest	A JavaScript and TypeScript test framework	Appendix B.2
`npm-audit`	An npm subcommand that checks installed packages for known security vulnerabilities	Appendix B.3
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix B.4
Zizmor	A static-analysis tool that identifies known vulnerabilities in GitHub actions	Appendix B.5

Areas of Focus

Our automated testing and verification work focused on the following questions:

- Does the codebase contain commonly known vulnerabilities or weaknesses, as identified by CodeQL and Semgrep?
- Do any of the GitHub workflows defined by the repository contain known vulnerable code patterns?
- Does the codebase have any dependencies with known vulnerabilities?
- Is the unit and integration test coverage across the codebase sufficient?

Test Results

The results of this focused testing are detailed below.

CodeQL
We built CodeQL databases for the TypeScript SDK and server, and the Swift SDK. We then analyzed these using public and Trail of Bits' private query suites. This analysis flagged that REST routes are registered without Express-level rate-limiting (see TOB-AAVEFAM-3 and TOB-AAVEFAM-9).

Jest
We used Jest to collect test coverage reports for the TypeScript SDK and server. The test suite passes, but unit test coverage across the repository is low (at 12%). However, many of the web services are exercised by the Playwright end-to-end tests, which means that this number does not correctly reflect the project's overall test coverage. That being said, we still recommend the Aave team to invest in more unit test coverage, as end-to-end testing typically tends to focus on the happy path and may miss implementation edge cases.

npm audit
We ran `npm audit` to identify JavaScript and TypeScript dependencies with known vulnerabilities. This analysis flagged an issue (CVE-2026-47429) in Vitest which allows for remote arbitrary file reads under some configurations. Vitest is only a development dependency but we still recommend the team to update Vitest to the latest version.

Semgrep
We ran Semgrep with a number of public rulesets and Trail of Bits' internal rulesets to identify vulnerable code patterns and unidiomatic code constructs. This analysis did not identify any issues in the codebase.

Zizmor
We ran the static-analysis tool Zizmor to audit the GitHub action workflows used in the repository. This identified a number of security issues in the project's CI/CD pipeline (TOB-AAVEFAM-12, TOB-AAVEFAM-13, TOB-AAVEFAM-14, and TOB-AAVEFAM-16).

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Refresh and access tokens are not invalidated when credentials change	Session Management	High
2	Web socket unconditionally skips device metadata checks	Session Management	Medium
3	Unauthenticated JSON-RPC proxy for Alchemy has no rate limiting or RPC method restrictions	Denial of Service	Medium
4	Passkey replay protection can be strengthened	Authentication	Informational
5	Unauthenticated CoinCover public keys allow server-side account recovery	Cryptography	High
6	Unauthenticated client public keys allow server-side account recovery	Cryptography	High
7	A malicious Shamir secret-share holder can choose the recombined secret key	Cryptography	Low
8	Swift SDK persists JWTs without enforced data protection	Data Exposure	Low
9	Two-factor disable attempts are not rate-limited	Authentication	Medium
10	Race condition in credential removal database query could lead to permanent account lockout	Timing	Medium
11	Race condition in password addition could lead to permanent account lockout	Timing	Medium

12	SDK release workflow executes dependency lifecycle scripts while holding GitHub Packages publish token	Configuration	**Low**
13	Production CI runner installs unpinned Vercel package	Configuration	**Low**
14	Self-hosted CI runner shared across PR-triggered and production-deploy workflows	Configuration	**Low**
15	IDOR in editWalletMetadata allows cross-user edits to wallet metadata	Access Controls	**Medium**
16	Third-party GitHub actions not pinned to commit hash	Configuration	**Low**
17	Minimum element size clickjacking guard can be bypassed with CSS transforms	Data Validation	**Medium**
18	Password normalization is different across platforms	Denial of Service	**Low**
19	Offline recovery master key derivation is different across platforms	Denial of Service	**Low**

Detailed Findings

1. Refresh and access tokens are not invalidated when credentials change

Severity: **High**	Difficulty: **Medium**
Type: Session Management	Finding ID: TOB-AAVEFAM-1
Target: `family-account/apps/server/src/services/jwt.service.ts` `family-account/apps/server/src/services/credentials.service.ts`	

Description

When a user authenticates to Family, the server returns a refresh token with an expiry time of either 14 or 182 days for web and mobile agents respectively. The refresh token can be presented to the `refresh` GraphQL mutation in exchange for an updated refresh token whose expiry time is reset back to the full 14 or 182 days, as well as an access token with a 30 minute expiry time. The access token is used as a short-term credential for authenticating the user's API calls. Password changes, credential removal, credential addition, and two-factor authentication enable / disable events do not invalidate these tokens. Any refresh token generated by an attacker after an account compromise can be perpetually renewed even if the compromised credential (password or passkey) is changed or removed.

The `refresh` mutation handler calls into `refreshAuthentication`, which verifies the current refresh token and then calls either `createDashboardJwtInfo` or `createInjectedJwtInfo` depending on the role specified in the JWT, although the behavior for both is essentially equivalent for the purposes of this issue. The dashboard case is implemented as shown in figure 1.1:

```
case JwtRole.REFRESH_DASHBOARD:
  return {
    jwtInfo: createDashboardJwtInfo({
      userAccountId: verifyResult.id,
      deviceId,
      deviceInfo,
    }),
    userAccountId: verifyResult.id,
    deviceId,
    appOrigin: null,
  };
```

Figure 1.1: Refresh operation for dashboard refresh tokens in `refreshAuthentication` (`family-account/apps/server/src/services/jwt.service.ts:432–442`)

The `jwtInfo` object returned by `refreshAuthentication` is returned in the GraphQL response. Its contents are the return value of the `createDashboardJwtInfo` function, which is implemented as shown in figure 1.2:

```
export const createDashboardJwtInfo = ({
  userAccountId,
  deviceId,
  deviceInfo,
}: DashboardJwtInfoParams): JwtInfo => {
  return {
    accessToken: createJwtAccessToken({
      role: JwtRole.ACCESS_DASHBOARD,
      deviceIdHash: DeviceIdHash.fromDeviceId(deviceId),
      id: userAccountId,
    }),
    refreshToken: createJwtRefreshToken(
      {
        role: JwtRole.REFRESH_DASHBOARD,
        deviceIdHash: DeviceIdHash.fromDeviceId(deviceId),
        id: userAccountId,
      },
      deviceInfo,
    ),
    idToken: createIdToken(userAccountId, ROOT_ORIGIN),
  };
};
```

Figure 1.2: `createDashboardJwtInfo` function implementation (`family-account/apps/server/src/services/jwt.service.ts:214–235`)

The code unconditionally calls `createJwtAccessToken` and `createJwtRefreshToken`, issuing a new access token and refresh token. These functions are implemented as shown in figure 1.3:

```
const createJwtAccessToken = (
    payload: JwtRolePayload): Jwt =>
{
  return new Jwt(
    jwt.sign(toJSON(payload), JWT_SECRET, {
      expiresIn: JWT_AT_EXP_TIME,
    }),
  );
};
const createJwtRefreshToken = (
    payload: JwtRolePayload,
    deviceInfo: DeviceInfo): Jwt =>
{
```

```
  return new Jwt(
    jwt.sign(toJSON(payload), JWT_SECRET, {
      expiresIn: deviceInfo.userAgent.parse().browser
        ? JWT_RT_WEB_EXP_TIME
        : JWT_RT_MOBILE_EXP_TIME,
    }),
  );
};
```

Figure 1.3: Implementation of `createJwtAccessToken` and `createJwtRefreshToken` (`family-account/apps/server/src/services/jwt.service.ts:95–111`)

Both of these functions generate new tokens with renewed expiry times. In the case of the refresh token the expiry time is selected based on whether or not the `User-Agent` header in the request is recognized as a web browser: if so, the expiry is set to `JWT_RT_WEB_EXP_TIME` (14 days); if not, the expiry is set to `JWT_RT_MOBILE_EXP_TIME` (182 days). The same functionality exists for injected tokens and is not meaningfully different.

Authentication checks are gated by the `verify` function. A simplified version of the implementation is shown in figure 1.4:

```
export const verify = async ({
  jwtToken,
  roles,
  deviceInfo,
  deviceId,
  omitDeviceCheck,
}: VerifyParams): Promise<JwtPayload | null> => {
  try {
    // cryptographically verify the JWT authenticity
    const result = jwt.verify(jwtToken.value, JWT_SECRET) as JwtPayloadRaw;
    // check that the device ID in the JWT matches the one in the request
    const headerDeviceIdHash = DeviceIdHash.fromDeviceId(deviceId);
    const jwtDeviceIdHash = new DeviceIdHash(result.deviceIdHash);
    if (!headerDeviceIdHash.equals(jwtDeviceIdHash)) {
      throw new AuthenticationError('Invalid device ID');
    }
    // extract the account ID
    const userAccountId = new UserAccountId(result.id);
    // unless told otherwise, verify device metadata matches an approved device
    const deviceValid = omitDeviceCheck
      ? await validateUserDeviceExists(userAccountId, deviceId)
      : await validateUserDevice(userAccountId, deviceId, deviceInfo);
    if (!deviceValid) {
      throw new AuthenticationError('Device was revoked');
    }
    // verified - return result
    switch (result.role) {
      case JwtRole.ACCESS_DASHBOARD:
```

```
    case JwtRole.REFRESH_DASHBOARD:
      return {
        id: userAccountId,
        iat: result.iat,
        exp: result.exp,
        role: result.role,
        deviceId,
        appOrigin: ROOT_ORIGIN,
      };
```

Figure 1.4: `verify` implementation (`family-account/apps/server/src/services/jwt.service.ts:245–218`)

The `verify` function checks that the JWT is authentic and not expired, that the device ID (if present in the token) matches the one presented in the request, and that the device metadata (browser vendor and version, system vendor and version, IP geolocation) matches a non-revoked device. It does not perform any revocation checks on the token itself. Once a token is minted it remains valid until expiry. Tokens are not invalidated by changing passwords, updating passkey info, removing a passkey, disabling 2FA, or enabling 2FA, even if the credential used to start the session is removed.

While the device ID and device metadata checks prevent the most trivial class of session theft, they do not offer protection in the case where an attacker steals session tokens using infostealer malware. In such a case, the attacker knows the victim's IP address and can attain the device ID (stored client-side by the browser) and the browser's user-agent string. The IP geolocation check considers any IP address geolocated to within 100km of the origin IP to be acceptable, which an attacker can bypass using a residential proxy service. The checks also do not protect against cases where the attacker is able to log into Family with a new device, such as when credentials and OTP are phished. We also found that requests made through the web socket set the `omitDeviceCheck` argument to true, bypassing device checks entirely - see TOB-AAVEFAM-2 for more information.

Exploit Scenario

An attacker compromises a victim's account and gains valid refresh tokens and access tokens. The victim notices that something is awry and defensively changes their password, enables two-factor authentication, and removes secondary account credentials. These operations do not invalidate the attacker's refresh tokens or access tokens, so the attacker maintains access to the account. As long as the attacker uses the same user-agent string as the victim, proxies their traffic to a location within the geofencing area of the session, and uses their compromised refresh token with the `refresh` GraphQL mutation every 14 or 182 days (depending on whether the token was for web or mobile), they can generate new refresh tokens and can maintain perpetual access to the victim's account.

This same scenario applies if an attacker compromises a victim's passkey. The victim defensively removes the passkey from their account, but the attacker's refresh token

remains valid and can be perpetually refreshed. This invalidates one of the key intended security properties of passkeys.

Recommendations

Short term, invalidate refresh tokens and auth tokens when any credential change operation is performed: password change, 2FA enable/disable, credential removal, and optionally credential addition. We suggest one of the two following implementation approaches:

1. Record a `last_credential_change` timestamp on the server, updated whenever a credential change occurs. Compare the JWT's `iat` (issued at) timestamp to the last credential change timestamp for the user, and reject the request if the JWT was issued prior to the change.
2. Record a `credential_version` GUID in the user account model on the server, and update it to a new random GUID whenever a credential change occurs. Store the current `credential_version` GUID in each issued JWT. Compare the JWT's GUID to the current one in the user model using a constant-time comparison, and reject the request if the GUID has changed.

Both of these approaches have similar security properties (achieving robust mass invalidation of prior tokens) with minor tradeoffs: the timestamp approach alleviates the need to store any additional fields in the JWT, whereas the GUID approach avoids storing the timestamp of the last credential change (a potentially sensitive piece of information) in the user model.

Long term, require the user to periodically re-authenticate once their session reaches a certain age, measured from the time of login rather than the last refresh cycle, to prevent perpetual usage of stolen session tokens. Consider decrementing this re-authentication timer every time the user's device metadata changes, so that changes in location shorten the time before a re-authentication prompt will occur. Additionally, introduce a "panic button" in the frontend that allows a user to manually invalidate all sessions and remove all trusted devices at once, and validate that this feature and related ones continue to be usable even if an attacker sets up hundreds of devices on the victim's account. Consider giving the user the option to revoke other potential avenues of compromise, such as connected third party integrations, at the same time.

2. Web socket unconditionally skips device metadata checks

Severity: **Medium**	Difficulty: **Medium**
Type: Session Management	Finding ID: TOB-AAVEFAM-2
Target: `family-account/apps/server/src/services/jwt.service.ts` `family-account/apps/server/src/api-graph.ts`	

Description

Family implements device metadata checks as a defense in depth protection against session theft. For HTTP requests, user agent and IP address are stored when the user first logs in. This metadata is compared against future requests in that session, and the session is invalidated if the user agent substantially changes or the geolocation of the IP address moves more than 100km away from that of the session originator. However, web service requests disable this check entirely.

Authentication checks are gated by the `verify` function. A simplified version of the implementation is shown in figure 2.1:

```
export const verify = async ({
  jwtToken,
  roles,
  deviceInfo,
  deviceId,
  omitDeviceCheck,
}: VerifyParams): Promise<JwtPayload | null> => {
  try {
    // cryptographically verify the JWT authenticity
    const result = jwt.verify(jwtToken.value, JWT_SECRET) as JwtPayloadRaw;
    // check that the device ID in the JWT matches the one in the request
    const headerDeviceIdHash = DeviceIdHash.fromDeviceId(deviceId);
    const jwtDeviceIdHash = new DeviceIdHash(result.deviceIdHash);
    if (!headerDeviceIdHash.equals(jwtDeviceIdHash)) {
      throw new AuthenticationError('Invalid device ID');
    }
    // extract the account ID
    const userAccountId = new UserAccountId(result.id);
    // unless told otherwise, verify device metadata matches an approved device
    const deviceValid = omitDeviceCheck
      ? await validateUserDeviceExists(userAccountId, deviceId)
      : await validateUserDevice(userAccountId, deviceId, deviceInfo);
    if (!deviceValid) {
      throw new AuthenticationError('Device was revoked');
    }
```

```
    // verified - return result
    switch (result.role) {
      case JwtRole.ACCESS_DASHBOARD:
      case JwtRole.REFRESH_DASHBOARD:
        return {
          id: userAccountId,
          iat: result.iat,
          exp: result.exp,
          role: result.role,
          deviceId,
          appOrigin: ROOT_ORIGIN,
        };
```

Figure 2.1: `verify` implementation (`family-account/apps/server/src/services/jwt.service.ts:245–218`)

When `omitDeviceCheck` is not set, the code calls `validateUserDevice` and passes in `deviceInfo`, which contains the device metadata mentioned above. This function checks whether the device ID (a GUID passed by the client) and device metadata match those stored in the database. When `omitDeviceCheck` is set, however, the code instead calls `validateUserDeviceExists`, which only checks the device ID and not any of the device metadata.

The HTTP server does not set `omitDeviceCheck`, but the WS server does, as shown in figure 2.2:

```
const serverCleanup = useServer(
  {
    schema,
    context: async (ctx) => {
      const headers = {
        ...ctx.connectionParams.headers,
        ...ctx.extra.request.headers,
      };
      return extractRequestContext(
        {
          url: nonNullable(ctx.extra.request.url, '...'),
          headers,
          socket: {
            remoteAddress: ctx.extra.request.socket.remoteAddress,
          },
        },
        {
          /* device check (type/geolocation) is omitted for web socket
          as they usually (especially on mobile) have no headers with
          device information */
          omitDeviceCheck: true,
        },
        // ...
```

Figure 2.2: Web socket server sets `omitDeviceCheck` to true (`family-account/apps/server/src/api-graph.ts:112–141`)

The web socket server provides access to all GraphQL operations, including the `refresh` mutation to gain an updated refresh token, without any device check protections. Although the comment within the code indicates that this was intentional, the implementation skips device checks even if the client can provide this metadata, and skips IP geolocation checks even though the server knows the incoming IP address.

No distinction is made between sessions initiated through the HTTP API and sessions initiated through the web socket. A refresh token or auth token that is created on the HTTP API, where device checks are enforced, can be used on the WS API without the checks.

Exploit Scenario

An attacker steals a refresh token and device ID from a victim's browser session. Instead of using the HTTP API, the attacker uses the web socket to access the victim's account, bypassing agent metadata and IP geolocation restrictions.

Recommendations

Short term, enforce device checks on web socket requests whenever device metadata is known by the server.

Long term, consider performing IP geolocation checks for websocket requests whenever the user's IP address is known to the server, even if device metadata is not available.

3. Unauthenticated JSON-RPC proxy for Alchemy has no rate limiting or RPC method restrictions

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-AAVEFAM-3

Target: `https://api.app.family.co/proxy/rpc/1`
`family-account/apps/server/src/rest/proxy/call-rpc.ts`
`family-account/apps/server/src/api.ts`
`family-account/apps/server/src/api-rest.ts`

Description

The Family API provides a JSON-RPC proxy that enables the frontend web application code to query EVM blockchains via Alchemy, a third party node provider, without revealing Aave's paid API keys for Alchemy to users. This proxy is unauthenticated, has no rate limiting, and does not restrict the types of request that can be proxied. Some RPC methods supported by Alchemy can be expensive and time consuming to execute, leading to rate limiting by Alchemy and high billing.

The proxy is implemented as shown in figure 3.1:

```
export const callRpc = async (req: Request, res: Response) => {
  const chainIDResult = z.coerce.number().safeParse(req.params.chainId);
  if (!chainIDResult.success) {
    res.status(422).json({ error: 'Invalid chain id provided, must be a number' });
    return;
  }
  const chainId = chainIDResult.data;
  const url = isInEnum(SupportedChain, chainId) ?
    getAlchemyRpcUrl(chainId) : undefined;
  if (!url) {
    res.status(422).json({ error: `Chain id ${chainId} is not supported` });
    return;
  }
  const validationResult = rpcSchema.safeParse(req.body);
  if (!validationResult.success) {
    res.status(422).json({ error: 'Invalid rpc params' });
    return;
  }
  const rpcRequest = validationResult.data;
  const response = await fetch(url.value, {
    body: JSON.stringify(rpcRequest),
    method: 'POST',
```

```
  });
  if (response.status >= 500) {
    res.status(500).json({ error: 'Error calling rpc' });
    return;
  }
  const contentType = response.headers.get('content-type');
  if (!contentType || contentType !== 'application/json') {
    res.status(500).json({ error: 'Error calling rpc' });
    return;
  }
  res.setHeader('content-type', contentType);
  Readable.fromWeb(response.body).pipe(res.status(response.status));
  return;
};
```

Figure 3.1: Alchemy JSON-RPC proxy implementation (family-account/apps/server/src/rest/proxy/call-rpc.ts:24–88)

The proxy takes in a chain ID from the URL, looks up the appropriate Alchemy RPC URL for that chain, validates that the request body meets the RPC schema, and proxies the request using the `fetch` API. The Alchemy RPC URL returned by `getAlchemyRpcUrl` contains the Alchemy API key; this is how proxied requests are hydrated with credentials. No code within this handler implements any rate limiting or restrictions on the types of RPC methods that can be called via the proxy.

The JSON-RPC proxy is part of the API server, which is constructed as shown in figure 3.2:

```
server.use(
  requestContextMiddleware(),
  cors<cors.CorsRequest>((_, callback) => {
    try {
      callback(null, {
        origin: ALLOWED_ORIGINS.map((origin) => origin.value),
      });
    } catch (error) {
      assertError(error);
      logger.error('Error in cors middleware', error);
      callback(error);
    }
  }),
  express.json({ limit: '5mb' }),
);
```

Figure 3.2: API server initialization code (family-account/apps/server/src/api.ts:30–46)

The request context middleware extracts parameters and authentication state from the request, but does not actively enforce any authentication or authorization in this case. The CORS middleware restricts the origins from which web browsers may make requests to the

proxy, but this does not prevent uncontrolled access from non-browser sources such as `curl`.

The route for the JSON-RPC proxy is configured as shown in figure 3.3:

```
app.post('/proxy/rpc/:chainId', callRpc);
```

Figure 3.3: JSON-RPC proxy route pointing to `callRpc`, with no `authorizedOnly` requirement or other security controls. (`family-account/apps/server/src/api-rest.ts:28`)

This HTTP POST route is available to unauthenticated users.

Exploit Scenario

An attacker crafts an `eth_call` request for a contract that contains an unbounded loop and repeatedly sends this call to the JSON-RPC proxy with the gas parameter set to a large value. For each request, Alchemy simulates the contract's EVM bytecode until the specified gas amount runs out. The attacker's crafted requests are designed to run for as long as possible, triggering Alchemy's rate limiting for Family's API key and degrading service for all operations.

An attacker may also make requests that require walking the entire chain, for example by requesting `eth_getLogs` for blocks 0 to latest; these take time and are billable operations under Alchemy's current pricing model. An attacker who sends a large number of requests from multiple IPs could run up an Alchemy bill in the order of thousands of dollars per day.

Recommendations

Short term, add a per-IP usage limits to the JSON-RPC proxy, for example a bucket of tokens that can be consumed by each requesting IP address before limits are applied. Additionally, restrict the RPC methods allowed by the proxy only to those that are necessary in the current authentication context; for example, if `eth_call` is only ever called by the application when a user is logged in, that method should only be available to authenticated users.

Long term, consider adding weighted gas limits to requests depending on the type and request parameters.

4. Passkey replay protection can be strengthened

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Authentication	Finding ID: TOB-AAVEFAM-4
Target: `apps/server/src/services/passkeys/passkeys.service.ts`	

Description

The `verifyLoginPasskey` function is used to verify passkey-based credentials. The implementation largely follows the W3 passkey specification, but could be strengthened against replay attacks in two ways.

Expire cached challenges on successful login. The implementation checks the passkey signature against the public key stored in the user's `UserCredentialInfo` on the server, and then verifies the challenge against the server-side cached challenge stored in the Redis cache. However, the stored challenge is not evicted from the cache on successful login. This means that the same credential could be replayed until the cache entry expires automatically. The cache is configured to expire stored challenges after five to ten minutes, but unconditionally deleting the challenge once the credential has been verified would make the challenge single use and would prevent the risk of replays completely.

```
export const verifyLoginPasskey = async (
  userId: UserAccountId,
  credentialsInfo: PasskeyCredentialInfo,
  signatureInfo: PasskeySignatureInfo,
): Promise<VerifyLoginPasskeyResult> => {
  const isValidSignature = await verifyPasskeySignature(credentialsInfo,
signatureInfo);

  if (!isValidSignature) {
    return { isValid: false, reason: 'Invalid signature' };
  }

  const challengeInfo = await passkeysChallengeCache.get(signatureInfo.id);

  if (!challengeInfo) {
    return { isValid: false, reason: 'Challenge not found' };
  }

  if (!challengeInfo.userId.equals(userId)) {
    return { isValid: false, reason: 'Invalid challenge user id' };
  }
```

```
  const clientInfo = parseClient(signatureInfo.clientData);

  if (clientInfo.type !== 'webauthn.get') {
    return { isValid: false, reason: 'Invalid client info type' };
  }

  if (clientInfo.crossOrigin) {
    return { isValid: false, reason: 'Cross origin passkey not allowed' };
  }

  const actualChallenge = new PasskeyChallenge(base64ToHex(clientInfo.challenge));
  if (!challengeInfo.challenge.equals(actualChallenge)) {
    return { isValid: false, reason: 'Invalid challenge' };
  }

  const authenticatorInfo =
    parseAssertionAuthenticator(signatureInfo.authenticatorData);

  if (!verifyRequiredAuthenticatorFlags(authenticatorInfo)) {
    return { isValid: false, reason: 'Invalid authenticator flags' };
  }

  const originResult = await verifyReadOrigin(clientInfo, authenticatorInfo);
  if (!originResult.isValid) {
    return { isValid: false, reason: 'Invalid origin' };
  }

  return { isValid: true };
};
```

Figure 4.1: The `verifyLoginPasskey` function does not delete the stored challenge from the challenge cache on success

family-account/apps/server/src/services/passkeys/passkeys.service.ts#L136–L184

Validate authenticator data signature counts. The implementation currently parses the signature count from the authenticator data, but does not check it against a server-side expected count. To further improve replay protection and protect against cloned passkeys, consider adding server-side counter checks before accepting the credential. Since passkeys can be synchronized across devices, care must be taken to avoid false positives. We recommend the following flow for checking the client-provided signature counter against a server-side stored counter.

```
if requestCounter == 0 and storedCounter == 0:
    // The authenticator does not use the signature counter.
    accept
if the credential is backup-eligeble or synchronized:
    // The authenticator's counter may not be reliable.
    accept
if the credential is single-device and either counter is non-zero:
```

```
require requestCounter > storedCounter
otherwise fall back to alternative authentication methods
```

Figure 4.2: Recommended flow for validating the authenticator's signature counter

The `be` (authenticator is backup-eligeble / multi-device) and `bs` (current backup-state) flags on the parsed authenticator data from the client can be used to determine the type of authenticator used.

Recommendations

Short term, delete the stored challenge once the passkey credential has been verified in `verifyLoginPasskey`.

Long term, consider validating the signature counter to detect replay attacks and cloned single-device passkeys.

5. Unauthenticated CoinCover public keys allow server-side account recovery

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-AAVEFAM-5
Target: `apps/server/src/services/coincover.service.ts`	

Description

The client SDK's implementation of the CoinCover recovery setup flow does not (and cannot) authenticate the CoinCover public key before using it to encrypt the provider Shamir share. A malicious or compromised Family Accounts server can replace the CoinCover public key with an attacker-controlled public key, decrypt the provider share, combine it with the server share, and recover the account master key.

```
const GenerateKeyResponseSchema = z
  .object({
    key_id: z.string(),
    public_key: z.string(),
    signature: z.string(),
  })
  .transform((data) => ({
    keyId: data.key_id,
    publicKey: data.public_key,
    signature: data.signature,
  }));
```

Figure 5.1: The server's CoinCover API client parses the public key signature, but the signature is not forwarded to the client SDK (family-account/apps/server/src/services/external-providers/coincover/coincover-api.client.ts#L50–L60)

```
const keyResult = await coincoverApiClient.generateKey({ userAccountId,
verificationId });

if (keyResult.isFailure()) {
  logger.error('Failed to generate CoinCover key', {
    userAccountId: userAccountId.value,
    verificationId: verificationId.value,
    error: keyResult.error.message,
  });
  return failure(new CoincoverGenerateKeyError());
}

const publicKeyPem = convertBinaryPublicKeyToPem(
```

```
  arrayify(`0x${keyResult.value.publicKey}`).buffer,
);

return success({ verificationId, publicKey: publicKeyPem });
```

Figure 5.2: The server CoinCover recovery setup flow returns the CoinCover public key, but omits the signature (family-account/apps/server/src/services/coincover.service.ts#L244–L259)

CoinCover's public ReCOVER documentation states that generated public keys are signed by CoinCover, that the response includes both a `publicKey` and a `signature`, and that the signature should be verified with a CoinCover-provided verifying key. The server's setup implementation in `startCoincoverSetup` parses the signature field from CoinCover's key-generation response, but the function returns only the public key to the client SDK.

The SDK accepts the server-provided public key without authentication, encrypts the provider share with it, and sends the encrypted provider share together with the API share to the Family Accounts server. A malicious server can therefore provide an attacker-controlled key to the client, decrypt the provider key, and then combine both keys to recover the recovery master key.

The same issue exists during recovery re-enrollment. After CoinCover verification, the server generates and returns a new public key that the SDK uses for the next provider share without receiving or verifying CoinCover's signature.

Exploit Scenario

An attacker compromises the Family Accounts server. When a user enables CoinCover recovery, the attacker substitutes CoinCover's public key with an RSA public key controlled by the attacker. The client SDK encrypts the provider share with this key and sends the API share to the server as part of the setup request. The attacker decrypts the provider share, combines both Shamir shares to derive the recovery master

key, decrypts the recovery private key, and decrypts the account master key. The attacker then gains access to the user's account material without completing CoinCover identity verification.

Recommendations

Short term, return CoinCover's key signature to the client SDK and verify the public key against a pinned CoinCover-provided verifying key before encrypting any recovery share during setup and re-enrollment.

Long term, treat all third-party recovery key material as authenticated data with explicit issuer verification, key-purpose binding, and add unit tests ensuring that the implementation is robust against key-substitution attacks.

6. Unauthenticated client public keys allow server-side account recovery

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-AAVEFAM-6
Target: `apps/server/src/services/coincover.service.ts`	

Description

During CoinCover recovery, the client SDK generates an ephemeral RSA key pair and sends the public key to the Family Accounts API. The client's public key is forwarded to CoinCover by the server, and is used by CoinCover to encrypt the provider share. Since the public key lacks any form of authentication, a compromised server can simply replace the public key obtained from the client with an attacker-controlled key, allowing the attacker to decrypt the provider share returned from CoinCover, and combine the resulting plaintext share with its own key share to obtain the recovery master key.

```
const recoverResult = await coincoverApiClient.recoverBackup({
  userAccountId,
  verificationId,
  publicKey,
});

if (recoverResult.isFailure()) {
  logger.error('Failed to recover backup from CoinCover', {
    userAccountId: userAccountId.value,
    verificationId: verificationId.value,
    error: recoverResult.error.message,
  });
  return failure(new CoincoverBackupRecoveryError());
}

const coincoverShareBackup = recoverResult.value.backup.find(
  (backup) => backup.itemKey === COINCOVER_SHARE_KEY,
);

if (!coincoverShareBackup) {
  return failure(new CoincoverShareNotFoundError());
}

if (coincoverShareBackup.backupId !== keyInfo.backupId.value) {
  logger.error('CoinCover backup ID mismatch', {
    userAccountId: userAccountId.value,
    verificationId: verificationId.value,
    storedBackupId: keyInfo.backupId.value,
```

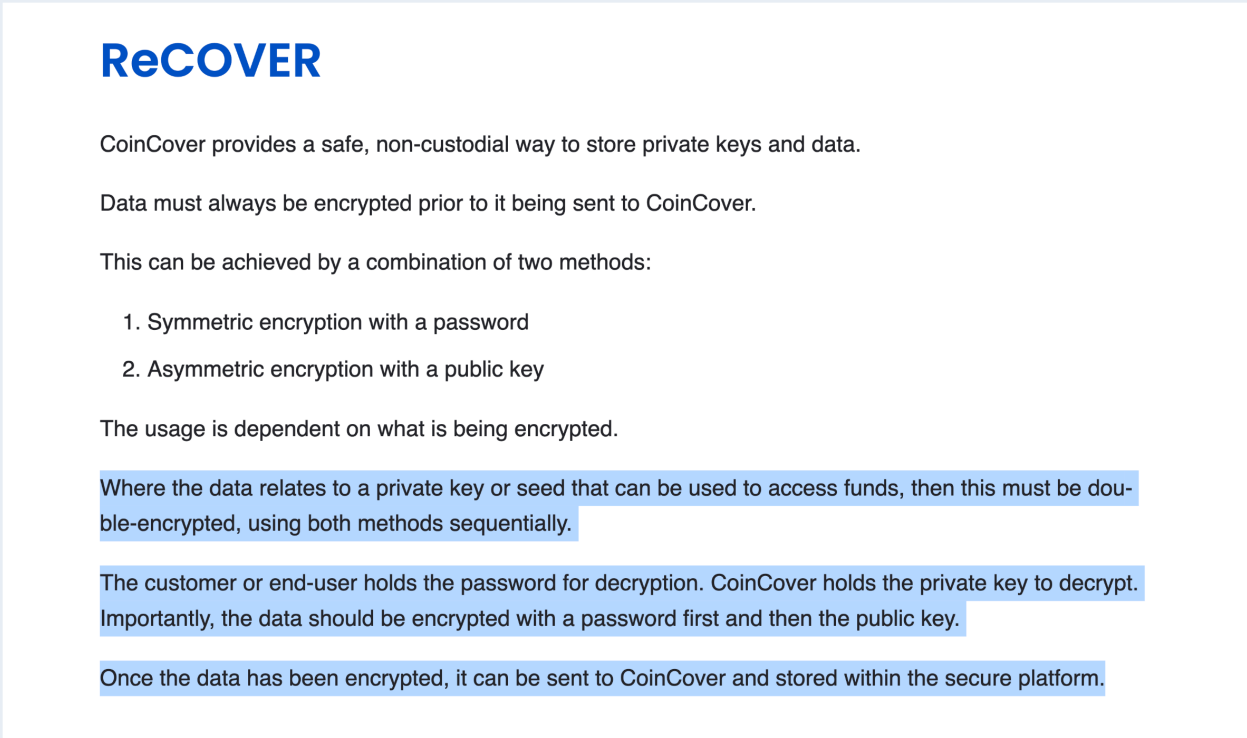
```
    recoveredBackupId: coincoverShareBackup.backupId,
  });
  return failure(new CoincoverBackupIdMismatchError());
}

const accountInfo = await requireUserAccountByUserId(userAccountId);

return success({
  coincoverKeyHash: keyInfo.coincoverKeyHash,
  serverKey: keyInfo.serverKey,
  encryptedCoincoverKey: new EncryptedCoincoverKey(coincoverShareBackup.itemValue),
  recoveryEncryptedAccountMasterKey: keyInfo.recoveryEncryptedAccountMasterKey,
  recoveryEncryptedPrivateKey: keyInfo.recoveryEncryptedPrivateKey,
  recoveryEncryptedPrivateKeyIv: keyInfo.recoveryEncryptedPrivateKeyIv,
  encryptedAccountInfo: accountInfo.encryptedAccountInfo,
});
```

Figure 6.1: A compromised server could replace the client-provided public key in the recovery request to CoinCover (family-account/apps/server/src/services/coincover.service.ts#L522–L565)

The CoinCover documentation explicitly states that customer-provided secrets like seeds or private keys must be double encrypted using both symmetric and asymmetric encryption to remain secure.



ReCOVER

CoinCover provides a safe, non-custodial way to store private keys and data.

Data must always be encrypted prior to it being sent to CoinCover.

This can be achieved by a combination of two methods:

1. Symmetric encryption with a password
2. Asymmetric encryption with a public key

The usage is dependent on what is being encrypted.

Where the data relates to a private key or seed that can be used to access funds, then this must be double-encrypted, using both methods sequentially.

The customer or end-user holds the password for decryption. CoinCover holds the private key to decrypt. Importantly, the data should be encrypted with a password first and then the public key.

Once the data has been encrypted, it can be sent to CoinCover and stored within the secure platform.

Figure 6.2: The CoinCover ReCOVER documentation requires customer-provided secrets to be double encrypted to protect against a compromised server

That being said, reviewing Fordefi's documentation describing their CoinCover recovery flow, CoinCover appears to provide undocumented support for user-provided GPG keys delivered out of band during recovery.

Exploit Scenario

Suppose that the Family Accounts server has been compromised by an attacker.

1. A user completes CoinCover recovery identity verification.
2. The user's client generates an ephemeral RSA key pair and sends the public key to the server.
3. The attacker replaces the user's public key with an attacker-controlled public key in the server-side `RecoverBackupRequest` request to CoinCover.
4. CoinCover returns the provider recovery share encrypted under the attacker-controlled key.
5. The attacker decrypts the provider share and combines it with the server-held key share.
6. The attacker reconstructs the recovery master key, decrypts the recovery private key, and uses it to decrypt the account master key.

Recommendations

Short term, reach out to CoinCover to better understand the options available to mitigate this issue.

Long term, provide public documentation that clearly describes the trust assumptions around the CoinCover recovery flow. In particular, if the security of the feature is based on the assumption that the server is not actively malicious, this should be documented.

7. A malicious Shamir secret-share holder can choose the recombined secret key

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-AAVEFAM-7
Target: `node_modules/.pnpm/shamir-secret-sharing@0.0.4/node_modules/shamir-secret-sharing/src/index.ts`	

Description

The `shamir-secret-sharing` module's `combine` function fails to check that all shares have a x-coordinate strictly greater than zero when the shares are recombined. If a malicious shareholder returns a secret share with x-coordinate equal to zero they can completely control the output from the share recovery process. In Aave's case, this means that if either CoinCover or the Family Accounts server is compromised, the attacker can choose the recovered master key.

```
export async function combine(shares: Uint8Array[]): Promise<Uint8Array> {
  // Share type and length validation ...

  // This will be our reconstructed secret
  const secretLength = shareLength - 1;
  const secret = new Uint8Array(secretLength);

  const xSamples = new Uint8Array(sharesLength);
  const ySamples = new Uint8Array(sharesLength);

  const samples: Set<number> = new Set();
  for (let i = 0; i < sharesLength; i++) {
    const share = shares[i]!;
    const sample = share[shareLength - 1]!;

    // The last byte of each share should be a unique value between 1-255 inclusive.
    if (samples.has(sample)) {
      throw new Error(
        'shares must contain unique values but a duplicate was found');
    }

    samples.add(sample);
    xSamples[i] = sample;
  }

  // Reconstruct each byte
```

```
  for (let i = 0; i < secretLength; i++) {
    // Set the y value for each sample
    for (let j = 0; j < sharesLength; ++j) {
      ySamples[j] = shares[j]![i]!;
    }

    // Interpolate the polynomial and compute the value at 0
    secret[i] = interpolatePolynomial(xSamples, ySamples, 0);
  }

  return secret;
}
```

Figure 7.1: The `combine` function fails to ensure that none of the x-coordinates are zero (`shamir-secret-sharing/src/index.ts#L273–L332`)

Exploit Scenario

A compromised Family Accounts server returns a Shamir secret share with x-coordinate zero, which means that the recovery master key will be equal to the secret share provided by the server. A compromised Family Accounts server could provide matching encrypted account data to the client, making the recovery succeed with attacker chosen account data.

Recommendations

Short term, update the `shamirCombineShares` function to check that all sharing indices are nonzero before calling into the `shamir-secret-sharing` module's `combine` function to recombine the provided shares.

Long term, consider performing internal code reviews of critical third-party dependencies to identify security vulnerabilities, and report any issues found to the maintainers.

8. Swift SDK persists JWTs without enforced data protection

Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-AAVEFAM-8
Target: `packages/sdk-ios/Sources/Services/Family/FamilyJWTSessionStorage.swift`	

Description

The iOS SDK stores the account access token, refresh token, and ID token as a JSON file without pinning an iOS data-protection class or excluding the file from backups. Because the SDK does not enforce a protection class, the confidentiality of these bearer tokens depends on the host application's data-protection entitlement and on the user having set a device passcode, neither of which the SDK controls. The file could also be included in iTunes and iCloud backups. An actor who obtains the app container, either through a backup or through forensic methods, can recover the refresh token and reuse the session.

The default family session configuration selects `FamilyJWTSessionFileStorage` for JWTs. The same branch uses the file-backed device ID store, while device recovery key material is persisted in a dedicated Keychain-backed store.

```
switch storage {
    case .ephemeral:
        jwtService = FamilyJWTSessionEphemeralStorage()
        deviceStorage = FamilyDeviceInfoEphemeralStorage()
        deviceRecoveryKeyStorage = FamilyDeviceRecoveryKeyEphemeralStorage()
    case .default:
        jwtService = FamilyJWTSessionFileStorage(...)
        deviceStorage = FamilyDeviceInfoStorage(...)
        deviceRecoveryKeyStorage = FamilyDeviceRecoveryKeyStorage(...)
}
```

Figure 8.1: Default iOS sessions use file-backed JWT storage while recovery keys use a Keychain-backed provider (family-account/packages/sdk-ios/Sources/PocketSDK.swift#L256–L269)

The `FamilyJWTSessionFileStorage.save` method copies each token string into a `JWTSessionCache` object, JSON-encodes it, and writes the bytes to the `jwt.data` file in the application container using `Data.write`. Because no `options` argument is supplied to `Data.write`, the SDK does not apply an explicit file-protection class or backup-exclusion attribute to the file.

```
func save(_ info: JWTSessionInfo) throws {
    let cachedInfo = JWTSessionCache(
        accessToken: info.accessToken.value,
        refreshToken: info.refreshToken.value,
        idToken: info.idToken.value
    )

    let encodedInfo = try JSONEncoder().encode(cachedInfo)
    let fileURL = try self.makeJWTSessionFileURL()

    if FileManager.default.fileExists(atPath: fileURL.path) {
        try FileManager.default.removeItem(at: fileURL)
    }

    try encodedInfo.write(to: fileURL)
}
```

Figure 8.2: The save method does not specify a file-protection class or a backup-exclusion attribute for the stored JWTs (family-account/packages/sdk-ios/Sources/Services/Family/FamilyJWTSessionStorage.swift#L17–L32)

iOS assigns a default per-file data-protection class only when the host application enables the Data Protection capability, and that protection takes effect only when the device has a passcode. A file written without an explicit class therefore inherits whatever the host application defaults to, which may be `NSFileProtectionNone`.

The SDK binds requests to the device with an `x-device-id` header and a consume-once `x-device-pop` proof whose signing key resides in the Keychain, so replaying a recovered token may still require additional device-bound key material.

Exploit Scenario

A user signs in with the iOS SDK, and the SDK writes the dashboard access token, refresh token, and ID token to the file system. The user later creates an unencrypted local backup of the device, or an attacker obtains the app container through a logical extraction or another filesystem weakness. The attacker reads the JSON file, extracts the refresh token, and replays it to obtain access to the user's account.

Recommendations

Short term, persist JWT session material through the existing `PocketKeychainStorage` wrapper with the `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` accessibility class. The after-first-unlock class allows the background session-refresh service to read the tokens while the device is locked, and the this-device-only class keeps them off other devices and out of backups.

Long term, centralize iOS secret storage behind a single provider with documented security properties for bearer tokens, recovery keys, session keys, and account metadata.

9. Two-factor disable attempts are not rated-limited

Severity: **Medium**	Difficulty: **Medium**
Type: Authentication	Finding ID: TOB-AAVEFAM-9
Target: `apps/server/src/services/two-factor/two-factor-authenticator.service.ts`	

Description

The dashboard's `DisableTwoFactorAuthCodeRequest` handler checks whether 2FA is enabled, validates the TOTP, returns `INVALID_CODE` on failure, and disables 2FA on success. However, the endpoint does not enforce any lockout or rate-limiting. This means that an attacker with a stolen dashboard session can make unlimited TOTP guesses to disable two-factor authentication for the compromised account.

```
export const disableTwoFactorAuth = async (
  userAccountId: UserAccountId,
  code: TwoFactorAuthCode,
): Promise<TwoFactorAuthStatus> => {
  if (!(await isTwoFactorAuthEnabled(userAccountId))) {
    throw new ValidationError("Two factor auth is not enabled");
  }

  const result = await validateTwoFactorAuthCode(userAccountId, code);
  if (!result) {
    return TwoFactorAuthStatus.INVALID_CODE;
  }

  await disableTwoFactorAuthDb(userAccountId);
  void isTwoFactorAuthEnabledCacheStorage.set({
    userAccountId,
    isEnabled: false,
  });

  return TwoFactorAuthStatus.SUCCESS;
};
```

Figure 9.1: The two-factor disable API handler validates TOTP codes and disables 2FA without updating the user-lock state (family-account/apps/server/src/services/two-factor/two-factor-authenticator.service.ts#L98–L115)

Exploit Scenario

An attacker obtains a victim's active dashboard access token through a browser compromise. The attacker calls the `DisableTwoFactorAuthCodeRequest` handler repeatedly with random TOTP values. Each incorrect guess returns `INVALID_CODE` but does not trigger any lockout behavior. The attacker continues guessing until a valid code is accepted, and the server disables two-factor authentication for the compromised account. The attacker can then use the lowered account protection to make future credential, recovery or session abuse easier.

Recommendations

Short term, reuse the same user-lock mechanism that is used in the login- and recovery-2FA flows in the disable-2FA flow.

Long term, centralize TOTP and recovery-code validation to a single helper that enforces lockout, and allows for auditing and alerts. Add regression tests for security sensitive endpoints to ensure that lockout is enforced and that the user-lock state is updated on failed attempts.

10. Race condition in credential removal database query could lead to permanent account lockout

Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-AAVEFAM-10
Target: `family-account/apps/server/src/database/credentials/write-operations/remove-credential.db.ts`	

Description

The authentication model for Family allows a user to have multiple credentials (e.g. a password and a passkey) assigned to their account. These credentials can be removed using the `removeCredential` GraphQL mutation. The code attempts to prevent a user from removing all of their credentials, since doing so would render their account inoperable. This check is vulnerable to a race condition.

The deletion is implemented in a SQL query used by the `removeCredentialDb` function:

```
DO $$
DECLARE
  remaining_credentials_count int;
BEGIN
  IF $<credentialType> = 'PASSWORD' THEN
    DELETE FROM auth_credentials
    WHERE user_account_id = $<userAccountId>
      AND credential_type = 'PASSWORD';
  END IF;

  IF $<credentialType> = 'PASSKEY' THEN
    DELETE FROM auth_credentials
    WHERE user_account_id = $<userAccountId>
      AND credential_type = 'PASSKEY'
      AND passkey_credential_id = $<passkeyCredentialId> ;
  END IF;

  SELECT COUNT(*) INTO remaining_credentials_count
  FROM auth_credentials WHERE user_account_id = $<userAccountId> ;

  IF remaining_credentials_count = 0 THEN
    RAISE EXCEPTION 'User needs to have at least one credential.';
  END IF;
END $$;
```

Figure 10.1: SQL query for credential removal (family-account/apps/server/src/database/credentials/write-operations/remove-credential.db.ts#L18–L37)

The apparent intent behind the query is to use the database transaction to ensure consistency and thus avoid a race condition. However, the default isolation in PostgreSQL is `READ COMMITTED`, which means that `SELECT` operations within the transaction operate on the snapshot taken at the start of the transaction - changes made within the transaction are visible to subsequent reads, but changes made outside of the transaction are not. PostgreSQL does serialize concurrent transactions when they modify or delete the same row, since the row-level lock taken by the modification or deletion alerts the transaction manager to a potential consistency issue, but it does not serialize concurrent transactions when they delete entirely distinct rows within the same table.

This leads to the following race condition:

1. Two calls to `removeCredentialDb` occur simultaneously. One call asks to delete the user's `PASSWORD` credential, the other asks to delete the user's `PASSKEY` credential.
2. Two SQL queries are executed concurrently: TX1 and TX2. PostgreSQL takes snapshots at the beginning of the queries. Both of these snapshots contain the `PASSWORD` and `PASSKEY` credential rows in the `auth_credentials` table.
3. TX1 enters the first `IF` block and performs a `DELETE` on the `PASSWORD` row within the TX1 snapshot. A row-level lock is placed on the `auth_credentials` row for the user's `PASSWORD` credential.
4. TX2 concurrently enters the second `IF` block and performs a `DELETE` on the `PASSKEY` row within the TX2 snapshot. A row-level lock is placed on the `auth_credentials` row for the user's `PASSKEY` credential.
5. The two row-level locks taken by the transactions are for distinct rows, so no serialization occurs.
6. TX1 reaches the `SELECT` statement. TX1's view of the data is based on the snapshot taken at the start of the transaction, which contained both the `PASSWORD` and `PASSKEY` rows. The earlier `DELETE` operation, in step 3, removed the `PASSWORD` row from the table in this transaction, but the `PASSKEY` row is still present because TX2 has not yet been committed. This `SELECT` returns 1.
7. TX2 reaches the `SELECT` statement. TX2's view of the data is also based on the snapshot taken at the start of the transaction, which contained both the `PASSWORD` and `PASSKEY` rows. The `DELETE` operation in step 4 removed the `PASSKEY` row from the table in this transaction, but the `PASSWORD` row is still present because TX1 has not yet been committed. This `SELECT` also returns 1.
8. TX1 reaches the final `IF` block. Its prior `SELECT` returned 1, so the check passes and no exception is raised.

9. TX2 reaches the final `IF` block. Its prior `SELECT` also returned 1, so the check passes again.
10. Both queries end and reach the commit stage. The PostgreSQL transaction manager sees that the two transactions modified distinct rows, which does not trigger rollback and reevaluation.
11. Both the `PASSWORD` and `PASSKEY` rows for the user are deleted in the committed state. The user account is left with no credentials and all future login attempts fail.

Exploit Scenario

An attacker compromises a user's account, then intentionally deletes both user credentials simultaneously to achieve a denial-of-service attack against their account, rendering it unusable.

This issue could theoretically be triggered by accident, e.g. by a user pressing both buttons in rapid succession, but it is very unlikely.

Recommendations

Short term, initiate a row-level lock on all `auth_credential` rows for the specified user at the start of the transaction. For example:

```
DO $$
DECLARE
    remaining_credentials_count int;
BEGIN
    PERFORM 1 FROM auth_credentials
              WHERE user_account_id = $<userAccountId> FOR UPDATE;
```

Figure 10.2: Suggested row-level lock implementation.

This prevents the race condition at step 5 of the flow described above. Each transaction applies update locks to all credential rows for the user, not just the one to be deleted, which forces serialization of the transactions.

Long term, consider taking a lock on the `user_account` row during credential change transactions. For example:

```
PERFORM 1 FROM user_account
          WHERE user_account_id = $<userAccountId> FOR UPDATE;
```

Figure 10.3: Suggested account-level lock implementation.

This locks a single central entity and prevents other account modifications from occurring concurrently.

References

- PostgreSQL docs for `READCOMMITTED`

11. Race condition in password addition could lead to permanent account lockout

Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-AAVEFAM-11
Target: `family-account/apps/server/src/services/credentials.service.ts` `family-account/apps/server/src/database/credentials/write-operations/add-credential.db.ts`	

Description

The authentication model for Family allows a user to have multiple credentials (e.g. a password and a passkey) assigned to their account. These credentials can be added using the `addCredential` GraphQL mutation. A user is only supposed to have either zero or one password credentials assigned to them at any point - never two or more. The code attempts to enforce this, but a pair of race conditions allow the checks to be bypassed. The login path enforces the number of password credential records for a user as a security invariant and throws an error if a user has two password credentials, preventing login.

The `addCredential` GraphQL mutation handler performs an authentication check and then calls into `addCredential` in `credentials.service.ts` to perform the credential addition operation. The password addition portion of the code is shown in figure 11.1:

```
switch (credentialType) {
  case CredentialType.PASSWORD: {
    const credentials = await requireUserCredentialsInfo(userAccountId);

    if (credentials.some((info) => info.credentialType === CredentialType.PASSWORD))
    {
      return failure(new PasswordAlreadySetError());
    }

    await addCredentialDb(common);
    break;
  }
```

Figure 11.1: Password credential addition code in `addCredential` (`family-account/apps/server/src/services/credentials.service.ts#L430-L440`)

The code first calls `requireUserCredentialsInfo` for the logged-in account ID, which returns the credentials associated with the user. The code then checks whether a password

credential record already exists for the user. If a password credential exists, it returns a failure, otherwise it calls `addCredentialDb` to perform the database update. This check is vulnerable to a TOCTOU race condition when two concurrent calls to `addCredential` are executed in parallel - if the second request's `requireUserCredentialsInfo` call completes before the first request's `addCredentialDb` call is reached, both requests pass the check and `addCredentialDb` is called twice.

The `addCredentialDb` function performs the following SQL query to update the database:

```
DO $$
DECLARE
    passwords_count int;
BEGIN
    IF $<credentialType> = 'PASSWORD' THEN
        SELECT COUNT(*) INTO passwords_count
          FROM auth_credentials
          WHERE user_account_id = $<userAccountId>
            AND credential_type = 'PASSWORD';

        IF passwords_count = 1 THEN
            RAISE EXCEPTION 'User already have password setup.';
        END IF;
    END IF;

    INSERT INTO auth_credentials(user_account_id, credential_type, ...)
    VALUES ($<userAccountId>, $<credentialType>, ...);
END $$;
```

Figure 11.2: Credential addition SQL query (family-account/apps/server/src/database/credentials/write-operations/add-credential.db.ts#L38–L52)

This query contains a secondary check to try to prevent the addition of two password credentials. It counts the number of rows in the `auth_credentials` table that match the user ID and the `PASSWORD` credential type, and raises an exception if the count is equal to 1.

This check is also vulnerable to a race condition. The default isolation in PostgreSQL is `READ COMMITTED`, which means that `SELECT` operations within the transaction operate on the snapshot taken at the start of the transaction - changes made within the transaction are visible to subsequent reads, but changes made outside of the transaction are not. Concurrent transactions that update or delete the same rows trigger a rollback and reevaluation at commit-time to ensure consistency, as do concurrent inserts that would cause a constraint violation, but concurrent inserts that produce distinct rows do not trigger this type of serialization.

This leads to the following race condition:

1. A user only has one PASSKEY credential and zero PASSWORD credentials. Two calls to `addCredential` are made simultaneously, both asking to add a PASSWORD.
2. The `addCredential` function runs concurrently for the two requests, calling `requireUserCredentialsInfo` in parallel. At this point the user only has a PASSKEY credential in the database, so the `credentials.some` check finds no PASSWORD credentials and passes.
3. The `addCredentialDb` function is called for both requests, executing the SQL query twice concurrently: TX1 and TX2. PostgreSQL takes snapshots at the beginning of the queries. Both of these snapshots only the PASSKEY credential row in the `auth_credentials` table.
4. TX1 enters the first `IF` block and performs a SELECT on the TX1 snapshot. The TX1 snapshot only contains the PASSKEY row, so `passwords_count` is set to 0. It reaches the `IF passwords_count = 1` branch, which is not taken.
5. TX2 concurrently enters the first `IF` block and performs a SELECT on the TX2 snapshot. The TX2 snapshot also only contains the PASSKEY row, so `passwords_count` is set to 0. It reaches the `IF passwords_count = 1` branch, which is not taken.
6. TX1 reaches the `INSERT` statement, inserting a new credential row into the TX1 snapshot.
7. TX2 reaches the `INSERT` statement, inserting a new credential row into the TX2 snapshot.
8. Both queries end and reach the commit stage. The PostgreSQL transaction manager sees that the two transactions both inserted a row. Since the addition of both rows would not violate a constraint, they are committed, leaving the user with two PASSWORD rows.

The `extractPasswordCredentials` function is called in all login paths. It includes an invariant check which fails when a user has more than one password credential, as shown in figure 11.3:

```
invariant(
  credentials.length <= 1,
  `Either none or one password credential should be present, found
${credentials.length}`,
);
```

Figure 11.3: Invariant requiring 0 or 1 password credentials during login (family-account/apps/server/src/services/credentials.service.ts:201–204)

As such, a user account with 2 or more password credentials assigned to it is prevented from logging in.

Exploit Scenario

An attacker compromises a user's account, then intentionally adds two password credentials simultaneously to achieve a denial-of-service attack against their account, rendering it unusable.

This issue could theoretically be triggered by accident by a user, but it is very unlikely.

Recommendations

Short term, add a partial unique index to the `user_account_id` columns of the `auth_credentials` table, constrained for password records. For example:

```
CREATE UNIQUE INDEX
  ON auth_credentials(user_account_id)
  WHERE credential_type = 'PASSWORD'
```

Figure 11.4: Suggested partial unique index on password credentials.

This ensures that only one `PASSWORD` row may exist for any user, and raises a constraint violation if a transaction attempts to create more than one `PASSWORD` row. Consider also adding this partial unique index for `PASSKEY` credentials.

Long term, consider taking a lock on the `user_account` row during credential change transactions. For example:

```
PERFORM 1 FROM user_account WHERE user_account_id = $<userAccountId> FOR UPDATE;
```

Figure 11.5: Suggested account-level lock implementation.

This locks a single central entity and prevents other account modifications from occurring concurrently.

References

- PostgreSQL docs for `READCOMMITTED`

12. SDK release workflow executes dependency lifecycle scripts while holding GitHub Packages publish token

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-AAVEFAM-12
Target: `.github/workflows/release-sdk.yml` `package.json`	

Description

The `release-sdk` CI job publishes the `@aave-account/sdk-react-native` package to GitHub Packages on every push to `master` that touches `packages/sdk-react-native/**`, `packages/sdk-common/**`, or `.changeset/**`. The job is granted `packages: write`, `contents: write`, `pull-requests: write`, and `id-token: write`, and runs `pnpm install --frozen-lockfile` without `--ignore-scripts`. A supply-chain compromise of any direct or transitive dependency of any workspace package, including development dependencies, can execute arbitrary code on the runner with these permissions in scope, allowing the attacker to publish a malicious SDK build to mobile wallets and dApp integrators consuming the package and compromise downstream customers.

The workflow declares the following permissions and install step:

```
permissions:
  contents: write
  id-token: write
  pull-requests: write
  packages: write

jobs:
  release-sdk:
    runs-on: family-latest
    steps:
      - uses: actions/checkout@v4
      - uses: pnpm/action-setup@v4
        with: { version: '10.28.2' }
      - uses: actions/setup-node@v4
        with:
          node-version: '22'
          registry-url: 'https://npm.pkg.github.com'
          cache: 'pnpm'
      - run: pnpm install --frozen-lockfile
      - name: Build SDK
```

```
        run: pnpm run -C packages/sdk-react-native build:force-bundle
      - name: Create release PR or publish
        uses: changesets/action@v1
        with:
          publish: pnpm changeset publish
          version: pnpm changeset version
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          NODE_AUTH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Figure 12.1: SDK release workflow permissions and install step (family-account/.github/workflows/release-sdk.yml#L18–L55)

The `pnpm install` step on line 41 does not pass `--ignore-scripts`. Because the root `package.json` declares `"postinstall": "pnpm run build"` at line 20, the install step automatically runs `pnpm run build` across the workspace after dependency resolution, which in turn invokes each workspace package's build script. Any `preinstall`, `install`, or `postinstall` script declared by a dependency anywhere in the resolved tree of any workspace package executes during this step.

Actions default to `persist-credentials: true`, which results in the `GITHUB_TOKEN` being persisted in `.git/config`. A malicious `postinstall` script can read this file and exfiltrate the token, or simply modify the build output before the changeset is signed and published.

The published SDK is consumed by the iOS and Android mobile wallets and by dApp integrators using the `apps/example-integration` template. A malicious SDK build would execute attacker-controlled JavaScript inside the wallet at runtime, with access to wallet state, signing flows, and credential storage.

Exploit Scenario

An attacker compromises a transitive dependency of any of the 24 workspace packages and publishes a new minor version. On the next push to `master` matching the SDK paths, the `release-sdk` workflow runs `pnpm install --frozen-lockfile` and executes the dependency's `postinstall` script on the runner. The script reads `.git/config`, exfiltrates the workflow's `GITHUB_TOKEN`, and patches the SDK build output before `pnpm changeset publish` runs. The patched SDK is signed and shipped to GitHub Packages under the legitimate maintainer token. Mobile wallet builds and dApp integrators install the SDK and execute the attacker's code, exposing wallet state and signing flows to runtime compromise on user devices.

Recent precedents for this npm post-install attack pattern include the `eslint-config-prettier` compromise (CVE-2025-54313, July 2025), in which a phished maintainer published versions carrying a Windows post-install trojan to 30M+ weekly downloads, and the Shai-Hulud npm worm (September 2025), which scrapes runner secrets during post-install and chains to publish further packages.

Recommendations

Short term, pass `--ignore-scripts` to the install step:

```
- run: pnpm install --frozen-lockfile --ignore-scripts
```

Figure 12.2: Suggested install step with lifecycle script execution disabled.

Set `persist-credentials: false` on `actions/checkout` to remove the `.git/config` exfiltration channel:

```
- uses: actions/checkout@v4
  with:
    persist-credentials: false
```

Figure 12.3: Suggested checkout step with credential persistence disabled.

Long term, add `--provenance` to `pnpm changeset publish` so the SDK is published with an SLSA provenance attestation tied to the workflow's OIDC identity, and require consumers to verify the attestation at install time. Gate the publish step behind a separate `environment: package-release` with required reviewers so a single compromised PR cannot ship to GitHub Packages without a second pair of eyes. Move the SDK publish into a dedicated job with `permissions: { packages: write, id-token: write }` only, so the install and build steps do not run with the publish token in scope.

13. Production CI runner installs unpinned Vercel package

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-AAVEFAM-13
Target: `.github/workflows/vercel-production.yml`	

Description

The `deploy-production` CI job installs the latest version of the `vercel` package on the runner, then uses it to perform a production deployment. The runner is provided with production credentials. A supply-chain attack against the `vercel` package or any of its dependencies would impact this CI job, giving an attacker control over the production environment.

The CI job installs `vercel@latest` globally using npm:

```
- name: Install Vercel CLI
  run: |
    npm cache clean --force
    npm install --global vercel@latest
    vercel --version
```

Figure 13.1: Vercel CLI installation steps for the `deploy-production` CI job (`family-account/.github/workflows/vercel-production.yml#L48–L52`)

This installs the latest version of the `vercel` npm package, as evaluated at the time of the job execution, along with its dependencies, on the runner. It also executes the pre-install, install, and post-install scripts of `vercel` and its dependencies.

The `VERCEL_TOKEN`, `VERCEL_ORG_ID`, and `VERCEL_PROJECT_ID_WEB` secrets are not loaded into env for this command. However, a malicious post-install script can gain persistence on the runner and extract these secrets from future steps.

The `vercel` package has 829 total dependencies from 567 unique organizations. These packages have around 600 contributors.

Exploit Scenario

An attacker compromises any of the 829 dependencies of Vercel. They introduce a malicious post-install script in the dependency that sends information about the environment and future running processes back to the attacker. The malicious code provides the attacker with the Vercel access token. The attacker uses this token to publish

malicious code to the production environment, compromising user accounts and stealing funds.

Recommendations
Short term, install with `npm ci --ignore-scripts` and mitigate the unpinned dependency in one of three ways:

1. Install the Vercel CLI package with an exact version, commit a `package.json` dev-dependency so the lockfile pins the entire tree.
2. Run the deploy inside a digest-pinned container image with `vercel` baked in.
3. Use cooldown periods with a private repository such as Artifactory to ensure a delay between new version releases and adoption of that new release.

In any case, update the Vercel package periodically and introduce security alerts to notify you when the package needs a more urgent upgrade.

Long term, add `--ignore-scripts` to every `npm install` / `pnpm install` in workflows that handle production secrets.

14. Self-hosted CI runner shared across PR-triggered and production-deploy workflows

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-AAVEFAM-14

Target: `.github/workflows/pr-validation.yml`
`.github/workflows/dependency-review.yml`
`.github/workflows/e2e-tests.yml`
`.github/workflows/main.yml`
`.github/workflows/hotfix.yml`
`.github/workflows/release-sdk.yml`
`.github/workflows/vercel-production.yml`
`.github/workflows/prepare-release.yml`
`.github/workflows/prepare-hotfix.yml`

Description

Every CI workflow in the repository except `spm-tag.yml` runs on self-hosted runners labelled `family-latest`. The same runner pool is used for workflows triggered by attacker-influenced events (`pull_request`, `repository_dispatch`) and workflows that hold production deploy credentials (`push` to `master`, tag pushes matching `v[0-9]+.[0-9]+.[0-9]+`, `workflow_dispatch` from privileged actors). GitHub's documentation states that self-hosted runners "do not have guarantees around running in ephemeral clean virtual machines, and can be persistently compromised by untrusted code in a workflow." If the runners in the `family-latest` pool are not ephemeral, a contributor who can open a PR can gain persistence on a runner and have it inherited by the next production-deploy job that lands on the same host.

The `runs-on: family-latest` label or `runner_labels: "family-latest"` parameter appears in the following sites:

```
e2e-tests.yml:20           runs-on: [family-latest]       # repository_dispatch,
workflow_dispatch
e2e-tests.yml:63           runs-on: [family-latest]
dependency-review.yml:10   runs-on: family-latest         # pull_request
release-sdk.yml:27         runs-on: family-latest         # push:master (SDK paths)
vercel-production.yml:18   runs-on: family-latest         # push:tags v*.*.*
pr-validation.yml:28       runner_labels: 'family-latest' # pull_request
main.yml:35                runner_labels: "family-latest" # push:master
hotfix.yml:33              runner_labels: "family-latest" # push:hotfix/*
prepare-release.yml:30     runner_labels: "family-latest" # workflow_dispatch
```

```
prepare-hotfix.yml:30      runner_labels: "family-latest" # workflow_dispatch
```

Figure 14.1: Workflows using the `family-latest` self-hosted runner pool.

The PR-triggered and dispatch-triggered workflows accept input from anyone who can open a PR or send a `repository_dispatch` event.

There is no documentation in the repository confirming that `family-latest` runners are ephemeral. If the runner host persists between jobs, a malicious or compromised contributor can land a workspace package change that exploits the lack of `--ignore-scripts` to gain persistence on the runner.

Exploit Scenario

A contributor with PR-open access opens a PR adding a malicious `postinstall` script to a workspace package's dependency tree, or compromises a single such contributor's workstation or GitHub session. The PR triggers `pr-validation.yml` on a `family-latest` runner. The post-install script writes a shim `pnpm` binary into `~/.local/bin` that wraps the real binary, then exits cleanly. The PR is closed before review.

Later, a legitimate merge to `master` schedules `main.yml` to a runner in the `family-latest` pool, and the scheduler picks the same host. The shim `pnpm` runs during the build step, captures the `PRODUCTION_AWS_ASSUME_ROLE` OIDC-minted credentials and the `ARGOCD_PROVISIONER_TOKEN` from the process environment, and exfiltrates them. The attacker uses these credentials to deploy a malicious image to the `pocket-production` ArgoCD project, compromising the wallet API serving custodial keys to users.

Recommendations

Short term, separate the runner pool by trust level. Use disjoint labels for PR-triggered and dispatch-triggered workflows versus push-triggered and tag-triggered privileged workflows:

- `family-pr` for `pr-validation.yml`, `dependency-review.yml`, `e2e-tests.yml`
- `family-prod` for `main.yml`, `hotfix.yml`, `release-sdk.yml`, `vercel-production.yml`, `prepare-release.yml`, `prepare-hotfix.yml`

The two pools must run on disjoint systems, with no shared filesystem or network reachability between them. Configure each runner registration to only accept jobs from its assigned label, and verify in the GitHub repository settings that the runner repository-allowlist disallows runners being picked up by PRs from outside the org.

Long term, migrate the `family-latest` pool to ephemeral runners using the GitHub Actions Runner Controller (ARC) just-in-time pattern: a fresh VM or container is provisioned per job and destroyed when the job ends, with no persistent filesystem or shell state between jobs. Document the ephemerality guarantee and add a CI check that fails if a workflow attempts to use a non-ephemeral label for a privileged event. Combine with

runtime detection on the runner hosts for the persistence patterns described above (process tree monitoring, anomaly detection on outbound network traffic, integrity monitoring of `~/.bashrc`, `~/.profile`, `~/.local/bin`, and cron tables).

Remove `cancel-in-progress: true` from `e2e-tests.yml` so a malicious dispatch cannot interrupt a benign run mid-cleanup.

15. IDOR in editWalletMetadata allows cross-user edits to wallet metadata

Severity: **Medium**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-AAVEFAM-15
Target: `apps/server/src/graphql/resolvers/wallet/index.ts` `apps/server/src/services/wallet.service.ts` `apps/server/src/database/wallet/write-operations/edit-wallet-index-metadata.db.ts`	

Description

The `editWalletMetadata` GraphQL mutation is used to edit the metadata of a user's wallet. This metadata includes the picture, emoji, icon, color, and label. The code does not properly check that the wallet whose metadata is being edited belongs to the currently logged-in user.

The `editWalletMetadata` handler takes in a wallet ID as part of the request. It then calls `editWalletIndexMetadata` without first verifying ownership of the wallet. This calls `editWalletIndexMetadataDb`, which performs the database update.

The database query is as follows:

```
UPDATE wallet_metadata
SET picture = $<picture>,
    emoji = $<emoji>,
    icon = $<icon>,
    color = $<color>,
    label = $<label>,
    date_updated_on = NOW()
WHERE wallet_id = $<walletId>;
```

Figure 15.1: SQL query for updating the wallet metadata.
(family-account/apps/server/src/database/wallet/write-operations/edit-wallet-index-metadata.db.ts#L18–L20)

This query only constrains the update to the `wallet_id`, and does not check the user ID associated with the wallet.

The `editWalletIndexMetadataDb` function performs a pair of SELECT queries in the same transaction, retrieving information about HD and private key wallets, scoped to the user ID, but these do not cause the transaction to abort. The code after the transaction completes includes an invariant which fails because both SELECT queries return no rows,

causing the request to return an error status, but the database has already been updated by this point.

```
const [, hdWallet, privateKeyWallet] = await db.tx((t) => {
  const update = t.none(
    outputSql(query, {
      userAccountId,
      walletId,
      ...walletMetadataToDbParams(walletMetadata),
    }),
  );
  const hdWalletSelect = t.oneOrNone<ToJSON<HdWalletDbResponse>>(
    getHdWalletSql(userAccountId, walletId),
  );
  const privateKeyWalletSelect = t.oneOrNone<ToJSON<PrivateKeyWalletDbResponse>>(
    getPrivateKeyWalletSql(userAccountId, walletId),
  );

  return t.batch([update, hdWalletSelect, privateKeyWalletSelect]);
});

const wallet = hdWallet ?? privateKeyWallet;

invariant(wallet, 'Failed to edit wallet index metadata');
```

Figure 15.2: Wallet ownership is checked after the update is performed (family-account/apps/server/src/database/wallet/write-operations/edit-wallet-index-metadata.db.ts#L28–L48)

Wallet IDs are UUIDs, but they are revealed to approved dApps and may be leaked through support or logging channels.

Exploit Scenario

An attacker who knows another user's wallet ID calls the `editWalletMetadata` mutation using that ID. An attacker could use this for vandalism, harassment, or to trick a user into thinking that their account was compromised (e.g. for social engineering purposes).

The wallet picture URL is restricted to the S3 upload path configured within the Family application, so it cannot be an arbitrary URL. However, since any other user can upload an image for their own wallet, they can still set essentially-arbitrary images as part of the request.

Recommendations

Short term, mirror the `removeWallet` pattern and pre-check ownership via `getWalletById(userAccountId, walletId)` before calling the DB layer.

Long term, consider adding defense in depth by updating the SQL query to constrain the update to wallets owned by the current user. For example:

```
UPDATE wallet_metadata wm
SET picture = $<picture>, …
FROM wallet w
WHERE w.wallet_id = wm.wallet_id
  AND w.wallet_id = $<walletId>
  AND w.user_account_id = $<userAccountId>;
```

Figure 15.3: Example of fixed SQL query.

16. Third-party GitHub actions not pinned to commit hash

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-AAVEFAM-16

Target: `e2e-tests.yml`
`release-sdk.yml`
`spm-tag.yml`
`dependency-review.yml`
`vercel-production.yml`	

Description
None of the third-party actions configured for the `family-account` repo were pinned to a commit hash. While some were configured to point to a major version tag, these tags are mutable and can be modified to point to new commits.

The following third-party actions were affected:

- `actions/checkout`
- `actions/dependency-review-action`
- `actions/setup-node`
- `aws-actions/configure-aws-credentials`
- `changesets/action`
- `pnpm/action-setup`
- `mikepenz/action-junit-report`

Exploit Scenario
An attacker compromises a third-party action, for example by phishing a maintainer, and pushes an update to the action containing a malicious payload. The payload steals credentials and other sensitive data, giving the attacker access to the CI and other production infrastructure.

Recommendations
Short term, pin the actions to commit hashes instead of using `@v*`.

Long term, enable organization-level enforcement of pinning in GitHub.

17. Minimum element size clickjacking guard can be bypassed with CSS transforms

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AAVEFAM-17
Target: `family-account/apps/web/components/common/ClickjackGuard/index.tsx`	

Description

The ClickjackGuard feature attempts to prevent clickjacking within dApps. It utilizes client-side JavaScript to measure the size of a container `div` element within the `iframe` and raises an alarm when the `div` element is found to be below the specified minimums. The measurement does not consider CSS transforms (e.g. scale), which can bypass this check.

The size check is implemented as shown in figure 17.1:

```
const MINIMUM_ALLOWED_WIDTH = 350;
const MINIMUM_ALLOWED_HEIGHT = 500;

export const ClickjackGuard = ({ children, onPotentialClickjack }:
ClickjackGuardProps) => {
  const onResize = useCallback(
    (_: HTMLDivElement, { contentRect }: ResizeObserverEntry) => {
      if (
        contentRect.width < MINIMUM_ALLOWED_WIDTH ||
        contentRect.height < MINIMUM_ALLOWED_HEIGHT
      ) {
        onPotentialClickjack(new ClickjackDetectedError('SIZE_BELOW_MINIMUM'));
      }
    },
    [onPotentialClickjack],
  );
```

Figure 17.1: Element size checking in ClickjackGuard (family-account/apps/web/components/common/ClickjackGuard/index.tsx#L106–120)

The check measures the `contentRect` width and height for the container `div` element. If a malicious page resizes the outer `iframe` through traditional means (i.e. actually resizing the element) then its `clientWidth` and `clientHeight` properties will shrink, propagating

through to the inner `contentRect` of the `div` being measured, leading to a clickjacking alert.

However, CSS transformations such as `transform: scale(0.5)` applied to the `iframe` do not affect its `clientWidth` and `clientHeight` properties, and therefore do not propagate to reductions in the `contentRect` size for the `div` element. This bypasses the ClickjackGuard feature.

The `isVisible` protection included elsewhere in the ClickjackGuard feature is not a guarantee - the WICG IntersectionObserver v2 codifies a best-effort attempt to identify whether or not an element is occluded or visible, but is not designed to be robust in a malicious context. Browser implementations differ in behavior and contain known gaps that can be abused to bypass the visibility check.

Exploit Scenario

A malicious dApp inserts the `iframe` via the SDK as designed, but then applies `transform: scale(0.1); transform-origin: top left;` to the `iframe`'s style immediately after `connect()` resolves. The inner `div` element measures above the minimum allowed values, so ClickjackGuard does not report any anomalies. The dApp utilizes a known bypass for visibility tracking, such as applying 99% opacity to an overlay element, to further obscure the element and conduct a clickjacking attack.

Recommendations

Short term, use a popup window for approval-required flows instead of the integrated modal. There is no robust way for JS running within the `iframe` to accurately measure its own visual size or reason about the state of elements outside of the `iframe`. Moving approval-required flows to a popup ensures that the page is hosted from a browser context that a malicious dApp cannot interfere with. Make sure to reject approval-required methods on the master-iframe channel to prevent a malicious dApp from bypassing the SDK's intended routing.

Long term, consider adding JS to the dApp-side code that posts the results of `getBoundingClientRect()` and `getComputedStyle()` on the `iframe` inward, failing closed if the data is not provided and alerting Aave engineers when an alert is tripped. Trigger alerts if the bounding client rectangle goes below the minimum allowed size threshold, or if the computed style information contains non-default `filter`, `opacity`, `mix-blend-mode`, `clip-path`, or `transform` styles. This information can be spoofed by a malicious dApp, but may still provide notice that a malicious dApp developer is experimenting with a clickjacking attack.

18. Password normalization is different across platforms

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-AAVEFAM-18
Target: `packages/custom-types/src/password.type.ts`, `packages/sdk-ios/Sources/Types/UserCredential.swift`	

Description

The TypeScript SDK normalizes the password according to NFKC before it is passed to scrypt. The same normalization is not applied by the Swift SDK. This means that a user who creates or updates their password on one platform could be unable to use the same password to authenticate themselves on another platform.

```
export class Password extends StringType {
  constructor(password: string, dangerBypassValidation = false) {
    // there may be multiple UTF-8 representations of a given string
    password = password.normalize('NFKC');
    super(password, dangerBypassValidation);
  }
```

Figure 18.1: The TypeScript SDK normalizes UTF-8 passwords before passing them to scrypt (family-account/packages/custom-types/src/password.type.ts#L32–L37)

```
struct Password {
    public let value: String

    public var data: Data {
        return Data(self.value.utf8)
    }

    public init(value: String) {
        self.value = value
    }
}
```

Figure 18.2: The Swift SDK performs no normalization of the password (family-account/packages/sdk-ios/Sources/Types/UserCredential.swift#L11–L21)

Exploit Scenario

A user creates an account in the web SDK with the password `Taro ﾊﾟｽﾜｰﾄﾞ2026!`, using a Japanese input method that emits half-width katakana. The web SDK normalizes the

password to `Taroパスワード2026!` before deriving and registering the credential authentication token.

Later, the user installs the iOS application and enters the same password. The iOS SDK hashes the raw password bytes without normalization, derives a different credential master key, and submits a different credential authentication token. The server rejects the login attempt, and the user must reset their credential using one of the supported recovery paths to access their account on iOS.

Recommendations

Short term, apply the same NFKC normalization before password validation and password-based key derivation on each platform.

Long term, add cross-platform regression tests and test vectors to ensure that password-derived keys agree across all supported platforms.

19. Offline recovery master key derivation is different across platforms

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-AAVEFAM-19
Target: `packages/custom-types/src/recovery-master-key.type.ts`, `packages/sdk-ios/Sources/Types/CredentialMasterKey.swift`	

Description

Offline recovery master keys are derived differently depending on which SDK processes the same offline recovery code. This means that a recovery code created on one platform will fail during recovery on another platform, preventing users from recovering accounts after device or credential loss.

The TypeScript implementation derives the recovery master key from the recovery code by passing the hexadecimal string representation of the 16-byte offline recovery code to scrypt. The Swift implementation derives the same key by passing the raw 16-byte offline recovery code to scrypt. It follows that the two platforms will derive different keys from the same recovery code.

```
public static async generate(
  id: UserAccountId,
  offlineRecoveryCode: RecoveryOfflineCode,
  scryptMethod: ScryptFunction,
): Promise<RecoveryMasterKey> {
  const value = await scryptMethod(id.value, offlineRecoveryCode.toHexString());
  return new RecoveryMasterKey(value);
}
```

Figure 19.1: The TypeScript SDK converts the recovery code to hexadecimal before passing it to scrypt (family-account/packages/custom-types/src/recovery-master-key.type.ts#L15–L22)

```
struct RecoveryMasterKey: CredentialMasterKeyGenerating {
    public let value: Data

    static func generateFrom(id: UserAccountID,
        offlineRecoveryCode: RecoveryOfflineCode) throws -> RecoveryMasterKey
{
        let userAccountIDData = Data(id.value.utf8)
        let scrypt: Data = try self.generateFrom(salt: userAccountIDData,
            password: offlineRecoveryCode.value)
```

```
        return RecoveryMasterKey(value: scrypt)
    }
}
```

Figure 19.2: The Swift SDK passes the raw recover code to scrypt without encoding it (family-account/packages/sdk-ios/Sources/Types/CredentialMasterKey.swift#L66–L75)

During setup, the SDK sends a recovery authentication token derived from the master key to the server, and the server stores a hash of the token together with the encrypted recovery data. During recovery, the SDK rederives the master key from the recovery code, computes the recovery authentication token and sends it to the server. If the token is different from the token used during setup, the server returns an error and recovery fails.

Exploit Scenario

A user generates an offline recovery code for their account using the web SDK and stores the displayed code in a secure location.

Later, the user loses access to their password, forcing her to recover her account using her offline recovery code. She enters the correct offline recovery code on her iPhone, but the iOS SDK derives a different recovery master key from the key used during setup. The server rejects the corresponding recovery authentication token, causing the account recovery process to fail.

Recommendations

Short term, remove the call to `RecoveryOfflineCode.toHexString` to consistently pass raw byte arrays to scrypt across all supported platforms.

Long term, add regression tests and corresponding test vectors to ensure that the same offline recovery codes result in the same recovery master keys and authentication tokens across all supported platforms.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Automated Analysis Tool Configuration

As part of this assessment, we used the tools described below to perform automated testing of the codebase.

B.1. CodeQL

We used CodeQL to detect known vulnerabilities in the TypeScript SDK and server, and Swift SDK implementations.

```
# Build database
codeql database create ../audit/static-analysis/codeql-db \
    --language=javascript-typescript \
    --source-root=. \
    --overwrite

# GitHub security-extended suite
codeql database analyze ../audit/static-analysis/codeql-db \
    codeql/javascript-queries:codeql-suites/javascript-security-extended.qls \
    --format=sarif-latest \
    --output=../audit/static-analysis/codeql-security-extended.sarif

# Trail of Bits internal suite
codeql database analyze codeql-db \
    path/to/trailofbits/internal.qls \
    --format=sarif-latest \
    --output=../audit/static-analysis/codeql-trailofbits.sarif
```

Figure B.1: The commands used to run CodeQL on the codebase

During the engagement, we used the following query suites to analyze the codebase:

- GitHub's `javascript-security-extended.qls` query suite
- Trail of Bits' internal queries

B.2. Jest

We used Jest to review the test coverage across the TypeScript SDK and server.

```
npx jest --forceExit \
    --coverage \
    --coverageReporters=text \
    --coverageReporters=json-summary \
    --coverageReporters=lcov \
    --coverageDirectory=coverage \
    > jest.txt 2>&1
```

Figure B.2: Command used to generate test coverage data for the TypeScript code in the repository

B.3. npm audit

We used the `npm audit` subcommand to review the project's TypeScript and JavaScript dependencies for known vulnerabilities.

```
npm audit
```

Figure B.3: Command used to review the TypeScript dependencies for known vulnerabilities

B.4. Semgrep

We ran the static analyzer Semgrep with the rulesets shown in figure B.5 to identify low-complexity vulnerabilities in the codebase.

```
brew install semgrep

semgrep --metrics=off --sarif --config=<RULE_SET> > semgrep-<RULE_SET>.sarif
```

Figure B.4: The command and rulesets used to run Semgrep on the codebase

We ran Semgrep with the following public rule sets as well as a number of Trail of Bits' internal rule sets.

- `p/javascript`
- `p/typescript`
- `p/react`
- `p/nodejs`
- `p/nodejsscan`
- `p/owasp-top-ten`
- `p/jwt`
- `p/secrets`
- `p/dockerfile`
- `p/github-actions`
- `p/ci`

B.5. Zizmor

We ran the static analyzer Zizmor to identify vulnerable configurations in the project's GitHub workflows.

```
brew install zizmor

zizmor --persona=auditor --format sarif .github/workflows/ > zizmor-results.sarif
```

Figure B.5: The commands used to run Zizmor on the GitHub workflows defined in the repository

To review the results of these analyses, we recommend outputting the result as SARIF, and then using Trail of Bits' SARIF Explorer VS Code extension.

C. FCA SCA Gap Analysis

This appendix contains a gap analysis to help identify areas where behaviors of the Aave Family system are likely to be incompatible with the FCA SCA specifications and its associated or referenced material.

This gap analysis focuses solely on the technical measures defined within the specification and does not include any analysis of organizational policy and procedure. Due to time constraints we focused our analysis on areas where we could identify clear deviations from the specifications, rather than attempting to exhaustively identify all potential deviations. This analysis is informative in nature and does not represent a compliance audit or accreditation for the purposes of FCA SCA.

A specific question we were asked by Aave is whether mobile passkeys unlocked by a biometric factor are compatible with SCA requirements. Based on our experience of UK banking applications (which we presume are required to be SCA compliant) and our reading of the FCA SCA specifications, we expect that the passkey and biometric factors would be considered independent possession and inheritance factors and would therefore be compatible as long as both factors are re-affirmed at each SCA-gated operation.

2FA should be mandatory

The FCA SCA defines strong customer authentication as:

> *"strong customer authentication" means authentication based on the use of two or more elements that are independent, in that the breach of one element does not compromise the reliability of any other element, and designed in such a way as to protect the confidentiality of the authentication data, with the elements falling into two or more of the following categories: knowledge, possession, inherence.*

FCA SCA Reg 100(1) states:

> *A payment service provider must* apply strong customer authentication *where: a payment service user accesses its payment account online, whether directly or through an account information service provider; initiates an electronic payment transaction; or carries out any action through a remote channel which may imply a risk of payment fraud or other abuses.*

Family does not currently mandate 2FA.

Sessions should be invalidated after 5 minutes of inactivity

SCA RTS Article 4(3)(d) states:

> *the maximum time without activity by the payer after being authenticated for accessing its payment account online shall not exceed 5 minutes*

Family currently uses a 30 minute expiry for access tokens, and a 14 or 182 day expiry for refresh tokens. There is also currently no mechanism for invalidating existing tokens.

Passwords should be checked for compromise

SCA RTS Article 2 states, in part:

> *Payment service providers shall ensure that the transaction monitoring mechanisms take into account, at a minimum, [...] lists of compromised or stolen authentication elements*

Family does not currently have any mechanism that checks whether user passwords have been compromised, for example by querying Have I Been Pwned or a similar service.

Strong authentication not applied when adding trusted beneficiaries

SCA RTS Article 13(1) states:

> *Payment service providers shall apply strong customer authentication where a payer creates or amends a list of trusted beneficiaries through the payer's account servicing payment service provider*

We expect that address book changes and dApp approvals would be considered in-scope for this requirement. Family does not require re-authentication with a second factor for these operations.

D. Code Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **The accepted algorithms are implicit when JWTs are verified.** The implementation uses `jsonwebtoken.verify` to verify client-provided JWTs. However, it does not specify the algorithm used. This is not a security issue since a symmetric secret is provided, but it means that any of the algorithms HS256, HS384, and HS512 are accepted. We recommend specifying the accepted algorithm explicitly to make the code easier to review.

```
const result = jwt.verify(jwtToken.value, JWT_SECRET) as JwtPayloadRaw;
```

Figure D.1: family-account/apps/server/src/services/jwt.service.ts#L253–L253

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From June 29 to July 01, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the Aave team for the issues identified in this report. On October 31, 2026, we performed a follow-up review of the mitigations for TOB-AAVEFAM-5 and TOB-AAVEFAM-7. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 19 issues described in this report, Aave has resolved 18 issues, and has not resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Refresh and access tokens are not invalidated when credentials change	High	Resolved
2	Web socket unconditionally skips device metadata checks	Medium	Resolved
3	Unauthenticated JSON-RPC proxy for Alchemy has no rate limiting or RPC method restrictions	Medium	Resolved
4	Passkey replay protection can be strengthened	Informational	Resolved
5	Unauthenticated CoinCover public keys allow server-side account recovery	High	Resolved
6	Unauthenticated client public keys allow server-side account recovery	High	Unresolved
7	A malicious Shamir secret-share holder can choose the recombined secret key	Low	Resolved
8	Swift SDK persists JWTs without enforced data protection	Low	Resolved

9	Two-factor disable attempts are not rate-limited	Medium	Resolved
10	Race condition in credential removal database query could lead to permanent account lockout	Medium	Resolved
11	Race condition in password addition could lead to permanent account lockout	Medium	Resolved
12	SDK release workflow executes dependency lifecycle scripts while holding GitHub Packages publish token	Low	Resolved
13	Production CI runner installs unpinned Vercel package	Low	Resolved
14	Self-hosted CI runner shared across PR-triggered and production-deploy workflows	Low	Resolved
15	IDOR in editWalletMetadata allows cross-user edits to wallet metadata	Medium	Resolved
16	Third-party GitHub actions not pinned to commit hash	Low	Resolved
17	Minimum element size clickjacking guard can be bypassed with CSS transforms	Medium	Resolved
18	Password normalization is different across platforms	Low	Resolved
19	Offline recovery master key derivation is different across platforms	Low	Resolved

Detailed Fix Review Results

TOB-AAVEFAM-1: Refresh and access tokens are not invalidated when credentials change
Resolved in PR #1200. Credential changes now update a timestamp in the database, and tokens that were minted before this timestamp are rejected, effectively invalidating them.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-2: Web socket unconditionally skips device metadata checks
Resolved in PR #1195. Device metadata checks are enforced when the metadata is present at the time of starting the session. Requests with multiple `User-Agent` headers are treated as having no metadata, which helps mitigate header injection attacks.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-3: Unauthenticated JSON-RPC proxy for Alchemy has no rate limiting or RPC method restrictions
Resolved in PRs #1202 and #1211. Per-IP rate and gas limits are now enforced and the accepted RPC methods are now restricted.

TOB-AAVEFAM-4: Passkey replay protection can be strengthened
Resolved in PRs #1198 and #1201. PR #1198 resolves challenge deletion, and PR #1201 resolves the counter checks.

TOB-AAVEFAM-5: Unauthenticated CoinCover public keys allow server-side account recovery
Resolved in commit fec9fe2. Both clients now verify a CoinCover signature over the public key using CoinCover-provided verification keys. Both setup and re-enrollment verify before encrypting a new provider share. This prevents key-substitution attacks.

TOB-AAVEFAM-6: Unauthenticated client public keys allow server-side account recovery
Unresolved. The Aave team has decided to accept this risk and provided the following context for this finding's fix status:

> *CoinCover recovery, like the rest of Family Accounts, assumes our server runs in a trusted environment. An attacker holding the server's secrets, database and CoinCover API key already controls authentication and sessions for every account. A compromised server would therefore be a much broader incident than this finding, and it's outside the threat model this feature is designed for.*

TOB-AAVEFAM-7: A malicious Shamir secret-share holder can choose the recombined secret key
Resolved in commits 991a180 and ab57caf. The TypeScript wrapper rejects shares ending in a zero x-coordinate before calling the library. Web and React Native both use this wrapper. Swift now invokes `ShamirShareValidator` on decoded shares immediately before combination. This validator rejects zero x-coordinates, which fixes the same vulnerability in the Swift client.

TOB-AAVEFAM-8: Swift SDK persists JWTs without enforced data protection
Resolved in PR #1232. Access, refresh, and ID tokens are now persisted through `PocketKeychainStorage`, and use the recommended `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` accessibility class.

TOB-AAVEFAM-9: Two-factor disable attempts are not rate-limited
Resolved in PR #1226. The same user-lock mechanism that is used in the login- and recovery-2FA flows is now used for 2FA-disable flows.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-10: Race condition in credential removal database query could lead to permanent account lockout
Resolved in PR #1223. All credential records for the user are locked for update during credential removal operations.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-11: Race condition in password addition could lead to permanent account lockout
Resolved in PR #1228. A database constraint now prevents the creation of multiple credential records of the same credential type for a single user.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-12: SDK release workflow executes dependency lifecycle scripts while holding GitHub Packages publish token
Resolved in PR #1221. The `--frozen-lockfile` and `--ignore-scripts` flags are now passed to `pnpm install` commands, and credential persistence is disabled on `actions/checkout`.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-13: Production CI runner installs unpinned Vercel package
Resolved in PR #1220. Vercel is pinned to a specific version and `--ignore-scripts` is passed during installation.

TOB-AAVEFAM-14: Self-hosted CI runner shared across PR-triggered and production-deploy workflows
Resolved in PR #1299. All CI workflows have been updated to use per-job, GitHub-hosted runners.

TOB-AAVEFAM-15: IDOR in editWalletMetadata allows cross-user edits to wallet metadata
Resolved in PR #1224. A wallet ownership check is performed using `getWalletById`, passing in the logged-in user's ID, and the SQL query for the `editWalletIndexMetadataDb` function now contains an ownership constraint.

TOB-AAVEFAM-16: Third-party GitHub actions not pinned to commit hash
Resolved in PR #1230. All third-party actions are now pinned to commit hashes.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-17: Minimum element size clickjacking guard can be bypassed with CSS transforms
Resolved in PR #1229. Approval-required RPC methods are routed to a popup window, and the master-frame channel now rejects approval-required methods.

The long-term recommendations for this issue were not implemented at the time of the fix review.

TOB-AAVEFAM-18: Password normalization is different across platforms
Resolved in PRs #1234 and #1239. PR #1234 resolves the key derivation aspect of the problem by applying NFKC normalization before key derivation. PR #1239 normalizes password validation on signup to ensure that a password initially entered on the website will not be rejected later by the iOS client and vice versa. It also removes validation for existing credentials.

TOB-AAVEFAM-19: Offline recovery master key derivation is different across platforms
Resolved in PR #1235. Both platforms now pass the raw recovery-code bytes to scrypt.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2026 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

Trail of Bits considers this report to be business confidential information; it is licensed to Aave under the terms of the project statement of work and is intended solely for internal use by Aave. Material within this report may not be reproduced or distributed in part or in whole without Trail of Bits' express written permission.

If published, the sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.