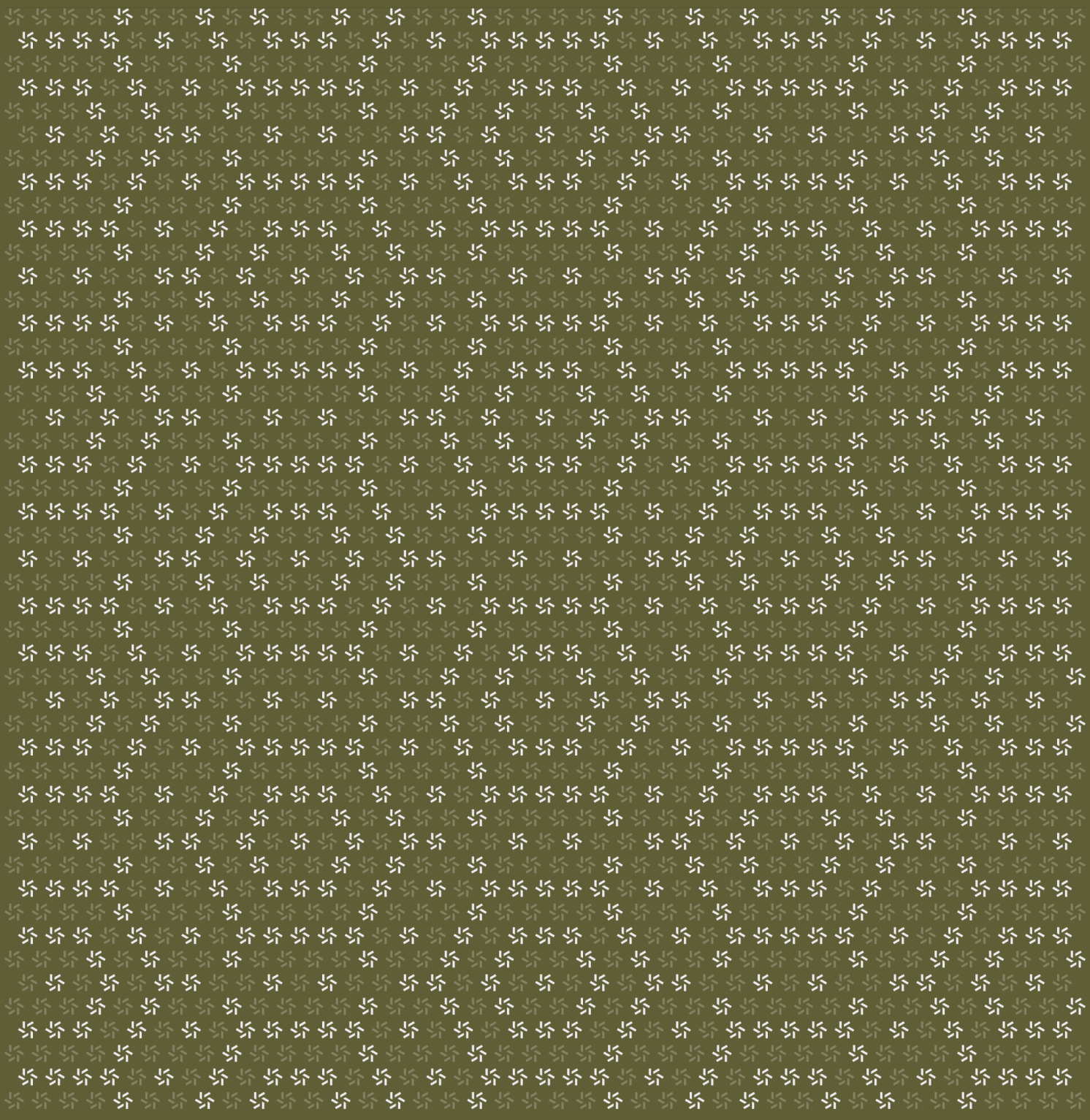
Zellic

Prepared for
Aave Labs
Aave Labs

Prepared by
Siwoo Mun
Seokchan Yoon
Zellic

May 26, 2026

Aave Accounts
Web Application Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Aave Labs from April 21st to May 19th, 2026. During this engagement, Zellic reviewed Aave Accounts's code for security vulnerabilities, design issues, and general weaknesses in security posture.

This engagement is documented in two separate reports. This report covers Component 1 and presents the associated findings and observations. Component 2 is addressed in a separate report.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the in-browser handling of recovery key shares preserve the wallet's threshold guarantees, even when the server or the recovery provider behaves maliciously?
- Are wallet secrets, session credentials, and JWTs held by the web front end protected from exposure to other origins, sibling tabs, and persistent browser storage?
- Do the signing approval flows in the web UI (`personal_sign`, `eth_signTypedData_v4`, `eth_sendTransaction`, and chain-management RPCs) bind each approval to the requesting dApp origin and to the exact payload the user approved?
- Do the cross-window, `postMessage`, iframe, and `BroadcastChannel` boundaries used by the web front end enforce origin and integrity such that a malicious dApp or sibling frame cannot impersonate the wallet UI or harvest session material?
- Is the client-side session life cycle (WebSocket subscriptions, JWT refresh, multitab synchronization, and revocation) consistent across in-memory state and persistent browser storage?
- Are user-controlled inputs handled by the web front end (typed-data payloads, transaction calldata, recovery codes, RPC URLs) bounded, validated, and decoded in a way that does not block the main thread or cause denial of service (DOS)?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- All autogenerated files, tests, and mocks
- Repository files outside the explicitly listed web front-end application and packages
- Server-side components under `apps/server`, including the GraphQL API, authentication services, transaction classification, and the server-side CoinCover integration

- Mobile SDKs and any native wallet code, including the iOS SDK (`packages/sdk-ios`) and the React Native SDK (`packages/sdk-react-native`)
- Shared TypeScript declarations under `packages/custom-types`
- Third-party dependencies and libraries, except where their usage by in-scope code introduces an issue
- Smart contracts and on-chain components interacted with by the wallet
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Aave Accounts files, we discovered 17 findings. No critical issues were found. Four findings were of high impact, six were of medium impact, and seven were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Aave Labs in the Discussion section (4. ↗).

Based on the number and severity of findings identified during this engagement, we recommend a comprehensive reassessment of the codebase following remediation efforts; see section 6. ↗

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	4
Medium	6
Low	7
Informational	0

2. Introduction

2.1. About Aave Accounts

Aave Labs contributed the following description of Aave Accounts:

> A non-custodial wallet and account infrastructure that lets end users create, recover, and use a crypto wallet through a familiar email/password experience, while keeping all sensitive key material client-side encrypted. An Aave Account (fka Family Account) is a cross-platform wallet. A user can register on web, log in on iOS, recover on React Native, and export a wallet back to web.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Unsafe code patterns. Many vulnerabilities in web-based applications arise from oversights during development. Missing an authentication check on a single API endpoint can critically impact the security of the overall application. Failure to properly sanitize and encode user input can lead to vulnerabilities like SQL injection, server-side request forgery, and more. We use both automated tools and extensive manual review to identify unsafe code patterns.

Business logic errors. Business logic is the heart of all web applications. We carefully review logic to ensure that the code securely and correctly implements the specified functionality. We also thoroughly examine all specifications for inconsistencies, flaws, and vulnerabilities, including for risks of fraud and abuse.

Integration risks. Web projects frequently have many third-party dependencies and interact with services and libraries that are not under the developer's control. We review the project's external interactions and identify potentially dangerous behavior resulting from compatibility issues and data inconsistencies.

Code maturity. We look for possible improvements across the entire codebase, reviewing violations of industry best practices and code quality standards. We suggest improvements to code clarity, documentation, and testing practices.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped files itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Aave Accounts Files

Type	TypeScript
Platform	EVM-compatible
Target	family-account
Repository	https://github.com/family/family-account ↗
Version	`dc3af8dc9c537a44a5d7fbc2f243f3d996e93da5`

Programs	`apps/web/src/**/*`
	`packages/client-api/**/*`
	`packages/sdk-common/**/*`
	`packages/sdk-dapp/**/*`
	`packages/sdk-web/**/*`
	`packages/utils/**/*`
	`packages/wallet/**/*`
	`packages/web-elements-bindings/**/*`
	`packages/sha256/**/*`
	`packages/shamir/**/*`
	`packages/test-helpers/**/*`
	`packages/eslint-config/**/*`
	`packages/hkdf/**/*`
	`packages/keccak/**/*`
	`packages/prettier-config/**/*`
	`packages/rsa/**/*`
	`packages/scrypt/**/*`
	`packages/crockford-base32/**/*`
	`packages/aes/**/*`
	`packages/bytes/**/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of six person-weeks. The assessment was conducted by two consultants over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Siwoo Mun
Engineer
mun@zellic.io ↗

Seokchan Yoon
Engineer
seokchan@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 15, 2026	Kick-off call
April 21, 2026	Start of primary review period
May 19, 2026	End of primary review period

3. Detailed Findings

3.1. Wallet instances expose private keys and mnemonics via public getters

Target	FAHdWallet, FAPrivateKeyWallet		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Both `FAPrivateKeyWallet` and `FAHdWallet` expose the underlying secret material through always callable public getters. There is no reason-bound export API; any caller that holds an instance reference can synchronously extract the private key and, for the HD wallet, the BIP-39 mnemonic as well. A leaked mnemonic is the root of the entire HD subtree, so a single leak compromises every account derived from it.

```
// packages/wallet/src/fa-hd-wallet.ts
export class FAHdWallet {
  // [...]
  public get privateKey(): PrivateKey {
    return new PrivateKey(this._wallet.privateKey);
  }

  public get mnemonic(): Mnemonic {
    return this._mnemonic;
  }
  // [...]
}
```

```
// packages/wallet/src/fa-private-key-wallet.ts
export class FAPrivateKeyWallet {
  // [...]
  public get privateKey(): PrivateKey {
    return new PrivateKey(this._wallet.privateKey);
  }
  // [...]
}
```

The backing fields are declared `private readonly`, but TypeScript's `private` is a compile-time annotation only. At runtime, the fields are ordinary properties, so even if the getters were removed, `wallet._wallet.privateKey` and `wallet._mnemonic` would still be reachable from any code in the same realm.

```
// packages/wallet/src/fa-hd-wallet.ts
export class FAHdWallet {
  // [...]
  private readonly _id: WalletId;
  private readonly _encryptionId: WalletEncryptionId;
  private readonly _index: number;
  private readonly _wallet: EtherWallet;
  private readonly _mnemonic: Mnemonic;
  // [...]
}
```

There is also no `dispose()` / zeroize path. The mnemonic and key remain on the JavaScript heap for the entire wallet lifetime, which, under normal UX, is the full session.

Impact

Successful cross-site scripting (XSS) in the dashboard realm, a malicious shared dependency, or any code execution in the same realm as the SDK can read the mnemonic and private key directly from the wallet instance. With the mnemonic, an attacker can rederive every account at every index the user has created, so the blast radius is the full HD tree rather than a single account.

Here is an attack scenario:

1. A user signs into the Family dashboard, which instantiates an `FAHdWallet` holding the BIP-39 mnemonic and the derived private key.
2. A same-realm script runs in the dashboard origin (compromised dependency in the dashboard bundle, stored XSS in a user-controlled field, or a malicious browser extension content script in the page world).
3. The script walks the React/SDK tree to obtain the wallet instance then reads `wallet.mnemonic` and `wallet.privateKey` synchronously through the public getters (or directly through `wallet._mnemonic` / `wallet._wallet.privateKey`, since TypeScript `private` is not enforced at runtime).
4. The script exfiltrates the mnemonic to an attacker endpoint. The attacker rederives every account at every index from the same seed and drains balances on every chain offline.

Recommendations

Replace the always-on getters with an explicit, audit-logged export API such as `exportPrivateKey(reason)` and `exportMnemonic(reason)`, and remove the `privateKey` / `mnemonic` properties from the public class surface. Use ECMAScript private fields (`#wallet`, `#mnemonic`) instead of TypeScript `private` so the runtime engine enforces the access boundary.

Add a `dispose()` method that clears references on log-out / wallet teardown so the secret does not outlive the session. Longer-term, consider moving signing into a context where the JavaScript heap never holds the raw key.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `13b84a95` ↗.

3.2. EIP-712 typed-data signing does not bind the signed `domain.chainId` to the active network

Target	SignerService		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Without comparing `domain.chainId` to the wallet's currently active network, `eth_signTypedData_v4` forwards the dApp-supplied `typedData.domain` to viem's `signTypedData()`. The approval modal renders the active network's logo and name, but the signature is computed against whatever `chainId` the dApp puts in the EIP-712 domain.

```
// apps/web/services/rpc/signer-service.ts
private async _signTypedData(typedData: EthereumTypedData): Promise<string> {
  const account = privateKeyToAccount(this.faWallet.privateKey.toHexString());

  return account.signTypedData({
    domain: typedData.domain,
    types: typedData.types,
    primaryType: typedData.primaryType,
    message: typedData.message,
  });
}
```

The approval call site forwards `typedData`, `appOrigin`, and `this.network` as independent arguments. Nothing in the path compares `typedData.domain.chainId` against `this.network.chainId`.

```
// apps/web/services/rpc/signer-service.ts
public async signTypedData(typedData: EthereumTypedData):
    Promise<SignerResult> {
  // [...]
  const result = await masterModalService.waitForTypedDataSignApproval(
    typedData,
    this.appOrigin.value,
    this.network,
  );
  // [...]
}
```

The approval modal pins the network header to the active wallet network only, so the user sees a familiar "Ethereum Mainnet" label even when the signed domain points to a different chain.

```
// apps/web/components/common/IntegratedModal/index.tsx
const IntegratedModalScreens = ({ /* [...] */ }) => {
  // [...]
  <GasPicker
    activeWallet={wallet}
    chainId={network.chainId}
    nativeCurrency={network.nativeCurrency}
  />
  // [...]
};
```

Impact

A connected dApp can request a typed-data signature whose `domain.chainId` differs from the chain the user thinks they are interacting with. The direct exploit class is a Permit signature for a token on the target chain; the user sees active-chain branding, but the dApp supplies the target chain's `domain.chainId` and `verifyingContract`, so the returned signature is valid on that other chain. If `domain.chainId` is omitted entirely, replay also becomes possible for verifiers whose domain separator does not bind the chain ID, especially when the same verifying contract address is deployed on multiple chains.

Here is an attack scenario:

1. A victim connects to a malicious dApp while their Family wallet is set to Ethereum Mainnet, and the dApp determines the wallet holds a balance of a permit-capable token on Polygon.
2. The dApp issues `eth_signTypedData_v4` with an EIP-2612 `Permit` message whose `domain.chainId = 137` (Polygon), `verifyingContract` is the token contract on Polygon, `spender = attacker`, and `value = MAX`.
3. The Family approval modal renders the active network header ("Ethereum Mainnet") and the message tree without showing the typed-data `domain`, so the user sees a familiar Permit prompt and approves.
4. The signer forwards `typedData.domain` as is to viem and returns a signature valid on Polygon.
5. The attacker submits `permit(owner, spender=attacker, value=MAX, deadline, v, r, s)` followed by `transferFrom(owner, attacker, balance)` on Polygon and drains the user's token balance there.

Recommendations

Reject signing requests whose `domain.chainId` is missing or does not equal the wallet's active `chainId` before invoking viem. The approval modal should display both the active wallet `chainId` and the typed-data `chainId`; if Family intentionally supports an exception flow, that flow should require a separate explicit confirmation before signing. Add a regression test that constructs a Polygon-domain Permit while the wallet is on Ethereum Mainnet and asserts the default signing path throws.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `6d1d6245` ↗.

3.3. Session credentials are broadcast in plaintext over an unauthenticated `BroadcastChannel`

Target	SessionSyncService		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Session changes across tabs are mirrored by `SessionSyncService` by serializing the full session DTO (`deviceId`, all three JWTs, client and server session keys) and posting it onto a `BroadcastChannel` whose name is derived from the dApp `host`. The receiving side feeds incoming messages straight into `sdk.session.refresh(...)` with no signature, MAC, sequence number, or timestamp check.

```
// packages/sdk-web/src/services/SessionSyncService.ts
constructor(
  name: string,
  private readonly sdk: AaveAccountSDK,
) {
  this.channel = new BroadcastChannel(name);

  sdk.session.onChange(async () => {
    const serialized = await sdk.session.serialise();
    this.channel.postMessage(
      serialized
        ? toJSON({
            deviceId: serialized.deviceId,
            jwtInfo: serialized.jwtInfo,
            clientSessionKey: serialized.clientSessionKey,
            serverSessionKey: serialized.serverSessionKey,
          })
        : null,
    );
  });
  this.subscribeToMessages();
}

public static forApp(appOrigin: AppOrigin, sdk: AaveAccountSDK):
    SessionSyncService {
  return new SessionSyncService(`${appOrigin.host}_session_sync_channel`,
    sdk);
```

```
}
```

```
// packages/sdk-web/src/services/SessionSyncService.ts
private subscribeToMessages() {
  this.channel.onmessage = async (event:
    MessageEvent<ToJSON<SessionRefreshConfig> | null>) => {
    await this.sdk.session.refresh(
      event.data
        ? {
            deviceId: new DeviceId(event.data.deviceId),
            jwtInfo: {
              accessToken: new Jwt(event.data.jwtInfo.accessToken),
              refreshToken: new Jwt(event.data.jwtInfo.refreshToken),
              idToken: new Jwt(event.data.jwtInfo.idToken),
            },
            clientSessionKey: new SessionKey(event.data.clientSessionKey),
            serverSessionKey: new SessionKey(event.data.serverSessionKey),
          }
        : null,
    );
  };
}
```

The `BroadcastChannel` provides origin isolation but no inter-tab authentication. Any same-origin code (XSS, compromised third-party script, browser-extension content script in the page world) can attach a listener and receive every credential the service publishes or post a forged payload of its own to swap or drop the session in every other tab. The service also has no `dispose()`, and `DashboardProvider` cleanup never calls `channel.close()`, so handlers can accumulate across remounts.

Impact

A single same-origin script gains durable read access to access/refresh/ID tokens plus both session keys whenever the user signs in, signs out, or rotates a token. The refresh token is sufficient to keep extending the session out of band. The same channel also accepts injected payloads, so an attacker can replace the active session with attacker-controlled credentials (priming a phishing flow) or post `null` to force–log-out the victim's other tabs.

Here is an attack scenario.

1. The dashboard origin loads a malicious script (XSS in a user-controlled field, a compromised npm dependency, or a content script injected by a hostile browser extension running in the page world).
2. The script opens its own `BroadcastChannel` on

`${location.host}_session_sync_channel` and registers an `onmessage` listener.

3. The legitimate `SessionSyncService` publishes a session payload (on log-in, refresh, or rotation). The attacker's listener receives the same DTO containing `accessToken`, `refreshToken`, `idToken`, `clientSessionKey`, and `serverSessionKey` and exfiltrates them.
4. With the refresh token alone, the attacker keeps the session alive out of band. As a follow-up, the same script posts an attacker-controlled DTO over the channel, causing the victim's other tabs to call `sdk.session.refresh(...)` with attacker credentials and silently swap to a session the attacker controls.

Recommendations

Treat the channel as a notification bus only: publish an opaque "session changed" signal and have each tab read the actual values from authenticated storage. If credentials must travel across tabs, compute a message authentication code (MAC) over the payload with a tab-local key established at boot and include a sequence number / timestamp to reject replays. Adopt a leader-election pattern (for example, `navigator.locks`) so only one tab publishes session updates, and add an explicit `dispose()` on `SessionSyncService` that closes the channel and unsubscribes the session listener during provider teardown.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `dd246ce8` ↗.

3.4. Browser `localStorage` holds JWTs, the device recovery private key, and session keys in plaintext

Target	JwtInfoStorageService, StorageService		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The Family web SDK persists multiple classes of authentication material as plaintext JSON in the browser `localStorage` sink — `JwtInfoStorageService` writes the access, refresh, and ID tokens under `family-accounts__jwt_info`, and the dashboard `StorageService` writes `deviceId`, `deviceRecoveryKey`, and `sessionKey` under `family-accounts__device_info`. Both services are wired to `window.localStorage`.

```
// packages/sdk-common/src/services/jwt-info-storage.service.ts
const JWT_INFO_STORAGE_KEY = 'family-accounts__jwt_info';

export class JwtInfoStorageService {
  // [...]
  public async set(data: JwtInfo): Promise<void> {
    await this.provider.setItem(JWT_INFO_STORAGE_KEY,
    JSON.stringify(toJSON(data)));
  }
  // [...]
}
```

```
// packages/sdk-web/src/services/StorageService.ts
const DEVICE_LOCAL_STORAGE_KEY = 'family-accounts__device_info';

export type DeviceInfo = {
  deviceId: string;
  deviceRecoveryKey: string;
  sessionKey: string;
};

export class StorageService {
  // [...]
  async setDeviceInfo(data: DeviceInfo): Promise<void> {
    await this.provider.setItem(DEVICE_LOCAL_STORAGE_KEY,
    JSON.stringify(data));
```

```
  }
  // [...]
}
```

```
// packages/sdk-web/src/services/AuthSessionService.ts
// inside AuthSessionService, after a successful shared-auth login:
if (result.status === SharedAuthResult.SUCCESS) {
  await this.storage.setDeviceInfo({
    deviceId: result.deviceId,
    deviceRecoveryKey: result.deviceRecoveryKey,
    sessionKey: this.state.clientSessionKey,
  });
}
```

The `deviceRecoveryKey` is not metadata; it is the RSA private key used for `DevicePoP` signatures during recovery. Storing it as a PEM string in `localStorage` turns the storage sink into a recovery-key escrow rather than a session cache.

```
// packages/sdk-common/src/utils/generateDevicePoP.ts
export const generateDevicePoP = async (
  recoveryPrivateKey: RecoveryPrivateKey,
): Promise<DeviceProofOfPossession> => {
  const message = getCurrentUnixTime().toString();
  const signature = await rsa().sign(
    recoveryPrivateKey.toPEMString(),
    stringToUtf8ByteArray(message),
  );
```

Impact

A single same-origin script can read both keys directly and exfiltrate the access token, refresh token, ID token, device ID, the device recovery RSA private key, and the client session key in one request. The refresh token alone is sufficient to keep the session alive; the device recovery key allows the attacker to mint fresh `DevicePoP` signatures offline until the key is rotated or revoked. Local malware that can read the browser profile gets the same material without any web exploit.

Here is an attack scenario:

1. The user has signed into the Family dashboard at least once, so `family-accounts__jwt_info` and `family-accounts__device_info` are populated in `window.localStorage`.
2. A same-origin script executes in the dashboard realm (XSS through a user-controlled field, a compromised dependency in the dashboard bundle, or a malicious

browser-extension content script in the page world).

3. The script reads both keys with `localStorage.getItem('family-accounts__jwt_info')` and `localStorage.getItem('family-accounts__device_info')` and POSTs the JSON to an attacker-controlled endpoint.
4. The attacker now holds the refresh token and the device recovery RSA private key. They use the refresh token to extend the session out of band and locally resign `DevicePoP` payloads (`rsa.sign(currentUnixTime)`) to satisfy device-bound checks for as long as the device is not rotated.

Recommendations

Move the refresh token to a server-set HttpOnly cookie so it is not reachable from JavaScript. Drop `deviceRecoveryKey` and `sessionKey` from the persisted DTO; if the recovery key must persist across reloads, hold it as a nonextractable `CryptoKey` in IndexedDB rather than a PEM string. For any data that still needs to live in storage, wrap it in Web Crypto AES-GCM envelope encryption with a nonextractable boot-time key so a storage dump alone is not sufficient to recover the values. Centralize all credential persistence behind a single secret-storage wrapper so future DTOs cannot bypass these policies.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `526011cb` ↗.

3.5. Typed-data approval UI omits EIP-712 `domain` and misses Permit2 spending semantics

Target	ParseTypedData, IntegratedModal, isErc20Permit		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `eth_signTypedData_v4` approvals iterate `typedData.message` and skip `typedData.domain`, so `verifyingContract`, `chainId`, and the contract/chain identity the signature is bound to are never shown. The "See Raw Message" toggle is gated behind `willAnalyseTx`, which covers only `SEND_TRANSACTION` and `TRANSACTION_APPROVAL`, not typed data.

```
// apps/web/components/Transactions/ParseTypedData/index.tsx
export const ParseTypedData = ({ typedData }: { typedData:
    EthereumTypedData }) => (
  <Box as="div" gap={16}>
    <pre>Primary Type: {typedData.primaryType}</pre>
    <Box as="div" gap={16}>
      {Object.keys(typedData.message).map((key, i) => {
        // [...] renders only typedData.message rows
      })}
    </Box>
  </Box>
);
```

```
// apps/web/components/common/IntegratedModal/index.tsx
const willAnalyseTx =
  (type === IntegratedModalTxInfoTypes.SEND_TRANSACTION ||
    type === IntegratedModalTxInfoTypes.TRANSACTION_APPROVAL) &&
  network.type === NetworkType.WELL_KNOWN;
// "See Raw Message" Dropdown is rendered only when willAnalyseTx is true
```

The modal header for typed data is generic: a `PermitTransferFrom` request shows the `primaryType` string under the `Signing Request` label, identical to a benign typed-data prompt.

```
// apps/web/components/common/IntegratedModal/index.tsx
const TxLabels = {
  [IntegratedModalTxInfoTypes.PERSONAL_SIGN]: 'Signing Request',
```

```
  [IntegratedModalTxInfoTypes.TYPED_DATA]: 'Signing Request',
  // [...]
};
case IntegratedModalTxInfoTypes.TYPED_DATA:
  return
    <div><h1>{props.typedData.primaryType}</h1><p>{TxLabels[props.txType]}
  </p></div>;
```

The server-side approval policy gates EIP-2612 `Permit` behind OTP by exact `primaryType` match. Permit2 types (`PermitTransferFrom`, `PermitBatchTransferFrom`, `PermitWitnessTransferFrom`, `PermitBatchWitnessTransferFrom`, `PermitSingle`, `PermitBatch`) are not detected and fall through to plain user approval.

```
// apps/server/src/services/transaction/utils.ts:34
export const isErc20Permit = (typedData: TypedData): boolean => {
  return typedData.primaryType === 'Permit';
};
```

```
// apps/server/src/services/transaction/unknown-typed-data-
    parsing.service.ts:13
if (isErc20Permit(typedData)) {
  return Promise.resolve({ approvalStatus:
    ApprovalStatus.VALIDATION_CODE_REQUIRED });
} else {
  return Promise.resolve({ approvalStatus:
    ApprovalStatus.USER_APPROVAL_REQUIRED });
}
```

Impact

A connected dApp can request a `PermitTransferFrom` signature, and the user sees only the message tree and the generic `Signing Request` label. The `verifyingContract` (the Permit2 contract) and `chainId` are EIP-712 `domain` fields, so they are omitted by `ParseTypedData`. The message rows show `spender`, `token`, and `amount` as raw key/value pairs with no human-readable spending summary. No OTP is enforced because Permit2 primary types are not recognized by `isErc20Permit`.

The drain path requires a victim to have previously granted the Permit2 contract an ERC-20 allowance for that token, which is the standard precondition for using any Permit2-integrated app (Uniswap UI, 1inch, etc.) and is a common state for active EVM users. Once that allowance exists, a single later phishing signature is sufficient to authorize an attacker `transferFrom` through Permit2 up to the signed amount and the existing allowance.

Here is an attack scenario:

1. A user previously approved `ERC20.approve(Permit2, MAX)` on a legitimate Permit2-integrated app. The on-chain allowance now exists.
2. The user connects to a phishing dApp with Family. The dApp issues `eth_signTypedData_v4` with `domain.verifyingContract = Permit2`, `domain.chainId` set to the target chain, `message.spender` set to the attacker, `message.permitted.amount` set to the drainable balance, and `primaryType = 'PermitTransferFrom'`.
3. The modal renders the message fields without `domain.verifyingContract` or `domain.chainId`. The header reads `PermitTransferFrom / Signing Request`. No raw-message toggle is reachable.
4. The server-side policy does not match `'Permit'` exactly, so the request goes through plain user approval.
5. The user approves. The attacker submits the signature to Permit2 with `transferFrom(owner, attacker, amount)`. The user's tokens move to the attacker through their preexisting Permit2 allowance.

Recommendations

Render `typedData.domain.verifyingContract`, `chainId`, `name`, and `version` on every typed-data approval, and make the "See Raw Message" toggle reachable for `IntegratedModalTxInfoTypes.TYPED_DATA` independently of `willAnalyseTx`.

Extend the server-side detector to cover Permit2 (`PermitTransferFrom`, `PermitBatchTransferFrom`, `PermitWitnessTransferFrom`, `PermitBatchWitnessTransferFrom`, `PermitSingle`, `PermitBatch`), and require the same OTP/validation-code flow that EIP-2612 already uses. The same detector should also classify common asset-transfer or order-authorization primary types from Seaport, Blur, and other marketplaces so the modal can present them with a context-specific header rather than the generic `Signing Request` label.

As defense in depth, derive a human-readable summary for known Permit/Permit2 message shapes (`token`, `spender`, `amount`, `deadline`) and render it above the raw message tree.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `324c58df` ↗.

3.6. Chain identification for `wallet_addEthereumChain` ignores `rpcURL`, approving an attacker RPC under a canonical chain identity

Target	identifyChain, NewNetworkApproval, JsonRpcService		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `identifyChain()` function matches `wallet_addEthereumChain` payloads against viem-known chains by `name`, `chainId`, and the `nativeCurrency.{name, symbol, decimals}` triplet only. The supplied `rpcURL` is not compared.

```
// packages/sdk-web/src/utils/chain.ts:95
export async function identifyChain(chain: AddCustomNetwork) {
  const { default: _, ...allChains } = await import('viem/chains');
  const allKnownChains: Chain[] = [lens,
    lensTestnet].concat(Object.values(allChains));

  return Object.values(allKnownChains).find((c) =>
    c.name === chain.name &&
    c.id === chain.chainId.value &&
    c.nativeCurrency.name === chain.nativeCurrency.name &&
    c.nativeCurrency.symbol === chain.nativeCurrency.symbol &&
    c.nativeCurrency.decimals === chain.nativeCurrency.decimals
  ) ?? null;
}
```

The approval modal gates its risk affordances on `!chainInfo`. Any payload that matches a known chain by metadata suppresses the unsafe-chain alert and the risk-acknowledgment checkbox and enables `Approve` on the first render frame:

```
// apps/web/surfaces/integrated-modals/network/NewNetworkApproval.tsx
const chainInfo = useChainInfoSuspense(network);
{!chainInfo && <Alert variant="warning">This chain may be unsafe.  Proceed with
    caution.</Alert>}
{!chainInfo && <Checkbox name="agree" label="I understand and acknowledge the
    risks..." onCheckedChange={setHasAcceptedRisk} />}
<Button onClick={()
    => approve()} disabled={!hasAcceptedRisk && !chainInfo}>Approve</Button>
```

Then, `NewNetworkApproval` is rendered through the `IntegratedModal` wrapper, not through `IntegratedModalScreens`. The 600ms `isDelayed` activation gate (`setIsDelayed(false)` after `setTimeout(..., 600)`) lives in `IntegratedModalScreens` and is used by the sign/typed-data/transaction approval surfaces, so it does not apply here.

The chain-ID precheck is trivially satisfied; any RPC that echoes the claimed `chainId` from `eth_chainId` passes. After approval, the handler commits the network and switches the dApp's active network to it:

```
// apps/web/services/rpc/json-rpc-service.ts:506
case HandledJsonRPCMethods.wallet_addEthereumChain: {
  const request = parseAddEthereumChainPayload(payload);
  const chainIdMatches = await this.matchesProvidedChainId(request.rpcURL,
    request.chainId);
  if (!chainIdMatches) throw rpcErrors.invalidParams({ message: 'Chain id does
    not match RPC chain id' });

  // [...] approval modal for non-allowlisted origins

  const result = await this.sdk.network.addNetwork(request);
  // [...]
  await this.changeActiveAppNetwork(result.value);
  return null;
}
```

The server-side duplicate check is keyed on `(chainId, rpcURL)`, so a new attacker-chosen RPC URL distinct from canonical or preexisting URLs is not blocked at the database layer.

The Chain URL is shown on the modal, but there is no signal that the URL is noncanonical: no comparison against viem's `chain.rpcUrls.default.http`, no host-allowlist warning, and both risk affordances are suppressed for `chainInfo`-truthy payloads. The defect is that what is known is determined solely by metadata, so an attacker dApp that echoes canonical values routes the request through the suppressed-affordance branch with an arbitrary RPC.

Impact

A connected dApp can register an attacker-controlled RPC under any viem-known chain's identity through a one-click approval. Postapproval impact depends on whether the targeted chain is one of Family's supported networks:

- **Chains in the Family supported list whose chain IDs are backed by the wallet's supported RPC proxy** (Ethereum, Polygon, Base, Arbitrum, Optimism, Scroll, Lens, Blast, zkSync, etc.). The dApp's read queries (`eth_call`, `eth_getBalance`, `eth_getCode`, etc.) route through the attacker's RPC, enabling balance / allowance / oracle / price spoofing of values the dApp reads through the wallet provider. The transaction broadcast may resolve to the legitimate supported RPC because `httpProxy` picks the

first SDK network with the matching `chainId`. Typed-data signatures returned directly to the dApp remain valid for the claimed chain because signatures bind to `chainId`, not `rpcURL`.

- **Chains outside the Family supported list** (BSC, Avalanche, Mantle, opBNB, Fantom, etc.). All of the above, plus the broadcast path, routes through the attacker's RPC because there is no legitimate supported entry to take precedence. The attacker can drop, delay, or selectively forward signed transactions.

Here is an attack scenario (BSC):

1. A user is connected to a malicious dApp with Family.
2. The dApp calls `wallet_addEthereumChain` with viem's canonical BNB Smart Chain metadata (`chainId: '0x38'`, `chainName: 'BNB Smart Chain'`, `nativeCurrency: { name: 'BNB', symbol: 'BNB', decimals: 18 }`) and `rpcUrls: ['https://attacker.example/rpc']`. The attacker RPC answers `eth_chainId` with `0x38`.
3. Then, `identifyChain` matches by metadata. The modal renders the entry under the `BNB Smart Chain` name, suppresses both risk affordances, and enables `Approve` on the first frame.
4. The user approves with a single click. Family stores the network and sets it active.
5. Subsequent reads route through the attacker. When the user signs a transaction, BSC is outside the server's `SupportedNetwork` set, so `httpProxy` resolves the broadcast to the attacker RPC. The attacker controls the broadcast as the active network RPC for that dApp session.

Recommendations

Treat a metadata-triplet match against a viem-known chain with a noncanonical `rpcURL` as an unknown-chain submission. When the supplied `rpcURL` host is not in `chain.rpcUrls.default.http` (or a curated allowlist of well-known providers), keep the warning alert visible, keep the risk-acknowledgment checkbox required, and stop rendering the chain by its canonical name.

Route `NewNetworkApproval` through `IntegratedModalScreens` so the `isDelayed` activation gate applies. As defense in depth, route high-value approvals through a pop-up–mode top-level browsing context for browsers where the integrated modal cannot be fully defended against parent CSS / layout primitives.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit 360ca610 ↗.

3.7. Without user confirmation, `wallet_switchEthereumChain` changes the per-dApp active chain

Target	JsonRpcService		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The per-origin active network is changed by `wallet_switchEthereumChain` silently as long as the requested `chainId` is already in the user's network list. The dApp SDK does not include this method in its approval-required list, the web `JsonRpcService` does not include it either, and the handler calls `changeActiveAppNetwork()` directly without invoking the approval modal. EIP-3326's "Security Considerations" section recommends displaying a confirmation for chain switches and canceling pending chain-specific confirmations.

```
// packages/sdk-dapp/src/sdk/rpc/rpc-request-handler.ts
const POTENTIALLY_APPROVAL_REQUIRED_METHODS = [
  'personal_sign',
  'eth_sendTransaction',
  'eth_signTypedData_v4',
  'wallet_addEthereumChain',
  'aave_switchAccounts',
];
```

```
// apps/web/services/rpc/json-rpc-service.ts
const APPROVAL_REQUIRED_METHODS = [
  HandledJsonRPCMethods.personal_sign,
  HandledJsonRPCMethods.eth_sendTransaction,
  HandledJsonRPCMethods.eth_signTypedData_v4,
  HandledJsonRPCMethods.wallet_addEthereumChain,
  HandledJsonRPCMethods.aave_switchAccounts,
];
```

```
// apps/web/services/rpc/json-rpc-service.ts
private async handleChainRequest(payload: JsonRpcPayload): Promise<unknown> {
  switch (payload.method) {
    case HandledJsonRPCMethods.wallet_switchEthereumChain: {
      const
    paramsResult = switchEthereumChainRpcSchema.safeParse(payload.params);
```

```
      if (!paramsResult.success) {
        throw rpcErrors.invalidParams();
      }

      const
    newChainIdResult = ChainId.fromHexString(paramsResult.data[0].chainId);
      // [...]
      const userNetworks = await this.sdk.network.getNetworks();

      const selectedNetwork = userNetworks.items.find((n) =>
        n.chainId.equals(newChainIdResult.value),
      );

      if (!selectedNetwork) {
        throw providerErrors.unrecognizedChainId();
      }

      await this.changeActiveAppNetwork(selectedNetwork);

      return null;
    }
    // [...]
  }
}
```

Moreover, `wallet_addEthereumChain` has a corresponding `waitForAddNetworkApproval()` gate, but the switch path has no equivalent.

Impact

A connected dApp can flip the active chain right before requesting a transaction approval. Because many DeFi UIs share addresses, token symbols, and visual layout across chains, a user who is mentally on Ethereum can end up signing a transfer or approval on Polygon or Base. The follow-up transaction modal does show a network logo, so detection is possible, but the chain switch itself is treated as a nonevent by the RPC layer and is not announced as a security-relevant action.

Recommendations

Add `wallet_switchEthereumChain` to both approval lists and gate the handler on an explicit modal that names the requesting origin, the source chain, and the target chain. Treat a switch to the currently selected chain as a no-op. When a switch is approved, reject any pending chain-specific approvals so the next signing request cannot inherit a stale network context.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `f3ee1a8a` ↗.

3.8. The `personal_sign` does not verify the SIWE `domain` against the requesting dApp origin

Target	SignerService		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

EIP-4361 requires wallets to verify the request origin against the SIWE `scheme` and `domain` fields before signing. Family's `personal_sign` path knows the trusted requesting origin as `appOrigin` and has access to the raw text, but it neither parses SIWE messages nor compares the SIWE `domain` against `appOrigin`. The RPC handler validates only that the payload is hex and that the requested address matches the active wallet.

```
// apps/web/services/rpc/json-rpc-service.ts
private async handleSignRequest(payload: JsonRpcPayload): Promise<unknown> {
  switch (payload.method) {
    // [...]
    case HandledJsonRPCMethods.personal_sign: {
      const [challenge, address] = payload.params;

      if (!isHex(challenge) || !isString(address)) {
        throw rpcErrors.invalidParams();
      }
      // [...]
      const signingResult = await signerService.signText(challenge);
      // [...]
    }
  }
}
```

The `SignerService` either signs immediately or shows a generic approval modal; `appOrigin` is forwarded only for display.

```
// apps/web/services/rpc/signer-service.ts
public async signText(text: Hex): Promise<SignerResult> {
  // [...]
  const signingRequiresUserApproval =
    this.forceUserApproval ||
    (await
```

```
    this.sdk.signingApproval.signingPersonalSignRequiresUserApproval(text));

  if (!signingRequiresUserApproval) {
    return {
      signedResult: await account.signMessage({ message: { raw: text } }),
      status: SigningStatus.APPROVED,
    };
  }

  const result = await masterModalService.waitForPersonalSignApproval(
    { text, network: this.network },
    this.appOrigin.value,
  );
  // [...]
}
```

The signing request DTO sent to the server carries `fromAddress` and `text` only; there is no `appOrigin` discriminator.

```
// packages/client-api/src/transaction/process-signing-request.api.ts
export type SigningRequest = {
  fromUsername: Username;
  fromAddress: EthereumAddress;
  tx?: UnsignedTx;
  typedData?: string;
  text?: PersonalSignText;
};
```

The approval UI just decodes the hex to UTF-8 and renders it; nothing parses SIWE structure or compares the message domain to the requesting origin. The server only matches a narrow set of well-known patterns (a Lens config, at the time of writing); unknown SIWE messages fall through to `USER_APPROVAL_REQUIRED` rather than being rejected for domain mismatch.

```
// apps/web/components/common/IntegratedModal/index.tsx
const IntegratedModalPreview = ({ /* [...] */ }) => {
  // [...]
  switch (type) {
    // [...]
    case IntegratedModalTxInfoTypes.PERSONAL_SIGN:
      return <pre>{fromHex(value, 'string').toString()}</pre>;
    // [...]
  }
};
```

Impact

A malicious dApp can relay a fresh SIWE challenge for an unrelated relying party (`target.example.com`), get the victim to sign it through `personal_sign`, and submit the signature on the relying party in the attacker's browser session. The relying party verifies the nonce and signature normally and binds the wallet address to the attacker's session. The wallet address is the log-in identifier, so the attacker gains whatever the relying party associates with that identity (gated content, off-chain permissions, administrative actions). This is a wallet-layer phishing path; the relying party itself can be fully compliant.

Recommendations

Before any `personal_sign` signature is produced, decode the payload as UTF-8, detect EIP-4361 messages by parsing the ABNF grammar, and compare the parsed `scheme`, `domain`, and effective port to the `appOrigin` known by `SignerService`. Reject mismatches by default and surface a clear warning rather than a generic preview when a SIWE message is detected. Apply the check in both the autosign and modal branches so a server response that does not require approval cannot still produce a mismatched signature. As defense in depth, include `appOrigin` in the signing approval DTO so the server can record and reevaluate the requesting origin.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `2c16c208` ↗.

3.9. The `ClickjackGuard` passes viewport-edge clipped iframes by checking only `isVisible`

Target	ClickjackGuard, rpc-request-handler		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `ClickjackGuard` configures `IntersectionObserver` with `threshold: 0`, `trackVisibility: true`, and `delay: 100` and reads `observerEntry.isVisible` only; `intersectionRatio` and the visible rectangle in the top-level viewport are never consulted.

```
// apps/web/components/common/ClickjackGuard/index.tsx
const options = {
  threshold: 0,           // catch even one pixel
  trackVisibility: true,  // IntersectionObserver v2
  delay: 100,
};
// [...]
if ('isVisible' in observerEntry) {
  return {
    ref: customRef,
    status: observerEntry.isVisible ? ElementVisibilityStatus.VISIBLE :
    ElementVisibilityStatus.NOT_VISIBLE,
  };
}
```

The companion `ResizeObserver` enforces a minimum only on the modal's *own* content rectangle, not on the visible portion:

```
// apps/web/components/common/ClickjackGuard/index.tsx
const MINIMUM_ALLOWED_WIDTH = 350;
const MINIMUM_ALLOWED_HEIGHT = 500;

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

If the parent positions the SDK-created iframe partially off screen, the modal still renders at its full internal size, so `ResizeObserver` passes. In the verified viewport-edge case, `IntersectionObserver V2` returns `isVisible: true` for the partially visible iframe even when only a small slice intersects the viewport (consistent with IO V2's `isVisible` semantics, which gate on active visibility rather than full-area intersection), so the guard reports `VISIBLE`.

Integrated-mode dApps always route approval RPCs through the master iframe when the browser supports IO V2, so this path is reached on every signing or transaction approval:

```
// packages/sdk-dapp/src/sdk/rpc/rpc-request-handler.ts
if (isIntersectionObserverV2Supported()) {
  return new MasterIframeRpcHandler(args);
}
```

The dApp can freely mutate the iframe's in-line `style`, so it can place the iframe so that only a 100x100 region containing the `Sign` / `Approve` button is in viewport while the rest is clipped off screen.

Impact

A connected dApp can construct a parent-side visual context that surrounds a small visible portion of the approval modal containing the action button, while the rest of the modal (network, amount, transaction summary, app-origin badge) is hidden by viewport-edge clipping. The user sees the attacker's fake UI plus the legitimate button slice, clicks the button, and approves a real signing or transaction request.

A hidden document (`document.visibilityState === 'hidden'`) is treated as `CALCULATING`. Both `clip-path: inset(...)` and `transform: scale(...)` are detected by IO V2 as occlusion / visibility loss and produce `isVisible: false`. Plain absolute positioning with negative offsets is not detected; a non-occluded element with at least one pixel intersecting the viewport still reports `isVisible: true`.

There is a verified attack: trigger `eth_signTypedData_v4` / `eth_sendTransaction` / `personal_sign` on a malicious dApp, and then set the iframe to `width:800px; height:600px; left:-700px; top:-500px; position:absolute;`. IO V2 reports `isVisible: true` with `intersectionRatio` around `0.0208`, `ClickjackGuard` returns `VISIBLE`, the `600ms` `isDelayed` activation window elapses, and a click on the visible 100x100 slice resolves the approval.

Here is an attack scenario:

1. A malicious dApp loads in a user's browser, and the user connects with Family.
2. The dApp draws a fake UI on top of the page that visually resembles a benign action (e.g., claim airdrop, confirm email, accept terms).
3. The dApp triggers a signing or transaction approval; the integrated modal is injected as a child iframe.

4. The dApp mutates the iframe's in-line style so the modal extends mostly outside the viewport, leaving only a 100x100 slice aligned with the "click here" button in the fake UI.
5. IO V2 returns `isVisible: true`, `ResizeObserver` passes on the 800x600 internal layout, and `ClickjackGuard` reports `VISIBLE`. The `600ms` activation gate elapses without the guard firing. The user clicks the slice, and the approval resolves to `approved: true`.

Recommendations

In `ClickjackGuard`, check that the visible portion of the modal in the top-level viewport meets a minimum threshold. The `IntersectionObserverEntry` provides `intersectionRect` and `intersectionRatio`; reject the modal as `NOT_VISIBLE` when `intersectionRatio` is below a threshold (for example, 0.9) or when `intersectionRect.width` and `intersectionRect.height` are below the same `350x500` minimum already enforced on `contentRect`.

For environments where the partial-visibility threshold cannot be made strict (e.g., the modal is intentionally allowed to be partially below the fold during scrolling), gate the action buttons on the strict-visibility check rather than the full modal: the `Sign` / `Approve` button only enables once its own IO V2 entry reports the full button region inside the viewport. As defense in depth, fall back to a top-level pop-up approval flow when the integrated path cannot satisfy the visibility check.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `0d4e4f18` ↗.

3.10. dApp-supplied native currency decimals are trusted by `wallet_addEthereumChain`, allowing transaction-amount spoofing

Target	JsonRpcService, NewNetworkApproval, AmountToken		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `nativeCurrency.decimals` field of `wallet_addEthereumChain` is the user-facing scale factor that the wallet uses to render every native-token amount on the resulting network. EIP-3085 defines it as a nonnegative integer that the wallet must validate; dApps cannot be trusted to provide a safe value. Family validates the field only with `z.number()` and never displays it on the add-network approval modal, so the user has no way to inspect the value that will later control how their native transfers are rendered.

```
// packages/sdk-web/src/services/rpc/types.ts
export const addEthereumChainRpcSchema = z.tuple([
  z.object({
    chainId: z.string().regex(hexNumberRegex),
    blockExplorerUrls: z.array(z.string()).optional(),
    chainName: z.string(),
    iconUrls: z.array(z.string()).optional(),
    nativeCurrency: z.object({
      name: z.string(),
      symbol: z.string(),
      decimals: z.number(),
    }),
    rpcUrls: z.array(z.string()).min(1),
  }),
]);
```

The handler copies the value straight into the stored custom network without any normalization against a canonical chain registry.

```
// apps/web/services/rpc/json-rpc-service.ts
case HandledJsonRPCMethods.wallet_addEthereumChain: {
  // [...]
  return {
    name: newNetworkRequest.chainName,
    rpcURL: new Url(newNetworkRequest.rpcUrls[0]),
```

```
    chainId: newChainIdResult.value,
    blockExplorer: blockExplorerUrl ? new Url(blockExplorerUrl) : undefined,
    nativeCurrency: {
      name: newNetworkRequest.nativeCurrency.name,
      symbol: newNetworkRequest.nativeCurrency.symbol,
      decimals: newNetworkRequest.nativeCurrency.decimals,
    },
  };
}
```

The add-network approval modal lists "Chain Name", "Chain URL", "Chain ID", and "Currency Symbol". The decimals row is absent, and there is no comparison against a known chain registry, so a custom network whose `symbol` matches a well-known chain but whose `decimals` differs is approved without warning.

```
// apps/web/surfaces/integrated-modals/network/NewNetworkApproval.tsx
export const NewNetworkApproval = ({ network }: Props) => {
  // [...]
  <dl>
    <dt>Chain Name</dt>
    <dd>{chain && <NetworkLogo chain={chain} alt={''} size={18} />}
    {network.name}</dd>
    <dt>Chain URL<Tooltip message={"The URL of the network's RPC
    endpoint"} /></dt>
    <dd><Tooltip
    message={network.rpcURL.value}>{network.rpcURL.value}</Tooltip></dd>
    <dt>Chain ID<Tooltip message={'The chain ID of the network'} /></dt>
    <dd>{network.chainId.value}</dd>
    <dt>Currency Symbol<Tooltip message={"The symbol of the network's native
    currency"} /></dt>
    <dd>{network.nativeCurrency.symbol}</dd>
  </dl>
}
```

Once stored, the same value is consumed as the only scale factor on the transaction approval header. The renderer divides the raw transaction value by `10 ** decimals` with no upper bound and no sanity check.

```
// apps/web/components/common/Amount/index.tsx
const formatTokenAmount = (
  amount: string | BigNumber | bigint,
  decimal: number,
  roundingMode?: BigNumber.RoundingMode,
): string => {
  const amountBigNumber = BigNumber(amount.toString());
```

```
  const amountFormatted = amountBigNumber.dividedBy(10 ** decimal);
  // [...]
};
```

A custom network whose `decimals` is set to 36 therefore renders 1e18 base units (one ETH on any 18-decimal chain) as `0.000000000000000001 ETH` on the approval header. The server-side scalar separately accepts up to 255, so very large `decimals` values can also be persisted.

Impact

A connected dApp can register a custom network whose `nativeCurrency.symbol` looks legitimate but whose hidden `decimals` is inflated and then induce the user to send a real native-value transaction. Because the approval header derives the displayed amount entirely from the stored `decimals`, the user reads a near-zero value and approves what is in fact a real funds transfer. The attack does not bypass any approval prompt; it bypasses the user's ability to judge the value on the prompt.

When combined with the supported-chain `chainId` collision behavior described in Finding 3.11. ↗, a custom network with an inflated `decimals` can be used so that the user sees an unknown-network preview with a downscaled ETH header while the actual send path executes against a supported mainnet route.

Here is an attack scenario:

1. A user visits a malicious dApp and clicks "Connect Wallet".
2. The dApp issues `wallet_addEthereumChain` with a real `chainId` and RPC URL but `nativeCurrency: { name: "Ether", symbol: "ETH", decimals: 36 }`. The RPC truthfully answers `eth_chainId`, so the chain identity check passes.
3. The add-network modal appears. The user sees the chain name, RPC URL, chain ID, and the `ETH` symbol. The `decimals: 36` field is never shown.
4. The user approves. Family stores the custom network and sets it as the active network for that origin.
5. The dApp immediately calls `eth_sendTransaction` with `value: 0xde0b6b3a7640000` (1e18 base units).
6. The approval header renders `1e18 / 10**36 = 1e-18 ETH`. The user reads "0.000000000000000001 ETH", believes the transfer is dust, and approves.
7. On chain, the transfer is one full ETH to the attacker.

Recommendations

Constrain `nativeCurrency.decimals` in the dApp-facing schema to a nonnegative integer with a wallet-enforced upper bound (for example, 36) so that out-of-range values are rejected at the

parser. Apply the same bound on the server-side `CurrencyDecimalsScalar`, and fix the existing `customNumberScalar` `minNumber` check so a `minNumber: 0` configuration is actually enforced.

Where the requested chain shares its `chainId` or symbol with a well-known network, ignore the dApp-supplied `nativeCurrency` entirely and normalize against a wallet-maintained chain registry. Display a `Currency Decimals` row on the add-network approval modal, and surface a warning when the supplied `name` or `symbol` matches a supported chain but the decimals differ.

As defense in depth, clamp the fraction-digit count used by the amount renderer so that even an out-of-range stored value cannot reach `Intl.NumberFormat` with an unsupported `minimumFractionDigits`.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `28315c5a` ↗.

3.11. Custom networks with a supported chain ID bypass the supported-chain transaction review path

Target	JsonRpcService, IntegratedModal		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

A dApp can register a custom network whose `chainId` collides with a supported network (for example, Ethereum Mainnet) by supplying a different RPC URL. The added network is represented client-side as `NetworkType.UNKNOWN`, so the approval modal disables transaction analysis even though the actual send path resolves the request back to the supported chain. EIP-3085 explicitly requires that the same `chainId` not be added more than once, since `chainId` is the wallet-level chain identifier.

The duplicate check in the backend is keyed on `(userAccountId, chainId, rpcURL)` rather than on `chainId` alone, so changing the RPC URL is sufficient to insert a second representation of the same chain.

```
// apps/server/src/database/network/queries/has-custom-network-by-chain-
    id.db.ts
const query = `
        SELECT *
        FROM user_custom_network ucn
        WHERE ucn.user_account_id = $<userAccountId>
        AND ucn.chain_id = $<chainId>
        AND ucn.node_rpc_url = $<rpcURL>
        LIMIT 1;
`;
```

The returned object is always `NetworkType.UNKNOWN`, even when the chain ID matches a supported chain.

```
// packages/client-api/src/network/add-network.ts
export const addNetworkApi = async (/* [...] */) => {
  // [...]
  const customNetwork: CustomNetwork = {
    type: NetworkType.UNKNOWN,
    customNetworkId: new CustomNetworkId(data.customNetworkId),
    // [...]
```

```
    chainId: new ChainId(data.chainId),
    rpcURL: new Url(data.rpcURL, { allowedProtocols: [UrlProtocol.HTTP,
    UrlProtocol.HTTPS] }),
  };
  // [...]
};
```

The approval modal gates analysis on the network type, not on the effective execution chain, so warnings, the transaction overview, and the changes list are all hidden for the unknown-network preview.

```
// apps/web/components/common/IntegratedModal/index.tsx
const IntegratedModalScreens = ({ /* [...] */ }) => {
  // [...]
  const willAnalyseTx =
    (type === IntegratedModalTxInfoTypes.SEND_TRANSACTION ||
      type === IntegratedModalTxInfoTypes.TRANSACTION_APPROVAL) &&
    network.type === NetworkType.WELL_KNOWN;
  // [...]
};
```

The send path then collapses identity back down to `chainId` — `getChainByNetwork()` returns viem mainnet for `chainId=1`, and the `httpProxy` transport selects the first SDK network with that chain ID, which is the supported Ethereum entry because supported networks are listed before custom networks.

```
// packages/sdk-web/src/utils/chain.ts
export function httpProxy({ sdk, proxyWebWorkerFactory }: HttpProxyConfig):
    CustomTransport {
  // [...]
  return ({ chain }) => {
    invariant(chain, 'Chain is required');

    const request = (async ({ method, params }) => {
      if (!CACHED_NETWORKS) {
        CACHED_NETWORKS = await sdk.network.getNetworks();
      }

      const network = CACHED_NETWORKS.items.find((n)
    => n.chainId.value === chain.id);
      invariant(network, 'Network should exist');

      const provider = new ProxyProvider(network, proxyWebWorkerFactory);
      // [...]
    });
```

```
    // [...]
  };
}
```

Impact

A connected dApp can broadcast a real Ethereum Mainnet transaction when the user had only ever seen the unknown-network review UI. Server-side approval policy still applies (the OTP gate is unchanged), but client-side simulation, scanner warnings, the transaction overview, and the asset-diff changes list are all suppressed for transactions that the user otherwise would have reviewed with full context. The Tenderly auto-approval host fast path, which is host-only, makes the flow even quieter when the attacker uses a URL that matches the configured host.

Recommendations

Reject `wallet_addEthereumChain` requests whose `chainId` matches a supported network unless the request is normalized to the canonical supported network entry. If custom RPCs for supported chains must be allowed, preserve the selected network identity end to end: key the viem client cache and transport on `(NetworkType, customNetworkId | chain)` rather than on `chainId`, and gate transaction analysis on the effective execution chain rather than on the stored network type. Replace the host-only Tenderly fast path with an exact URL or server-issued allowlist entry.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit d8574257 ↗.

3.12. Verified dApp status downgrades single-warning transaction scanner alerts from danger to warning

Target	TransactionWarnings		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `TransactionWarnings` computes the aggregate warning severity returned by the transaction-analysis scanner and then unconditionally overwrites it to `WarningSeverity.VERY_LOW` whenever the requesting dApp origin is present in verified-domains.json. The single-warning branch of `TxWarnings` then picks its alert variant from this overwritten aggregate value, so a `MEDIUM`, `HIGH`, or `VERY_HIGH` scanner label from a verified origin is rendered as a yellow `warning` instead of a red `danger`.

```
// apps/web/surfaces/shared/Transactions/TransactionWarnings/index.tsx
const txWarnings = parseValidationScanners(simulation.validation.scanners);

const appOriginDetails = getAppOriginDetails(appOrigin);
let warningSeverity = getRiskLevel(txWarnings);

// if app is verified, lower warning severity
if (appOriginDetails.verified) warningSeverity = WarningSeverity.VERY_LOW;

if (txWarnings.length === 0 && simulation.validation.actionType === 'Safe')
    return null;

return (
  <Box as="div" gap={24}>
    <TxWarnings warnings={txWarnings} warningSeverity={warningSeverity} />
  </Box>
);
```

The `txWarnings[i].severity` value is the backend `/proxy/family/v3/wallets/${chainId}/tx/analyze` response's `validation.scanners[*].label.severity` passed through unchanged; `DecoratedTransactionAnalysis()` returns the raw response, so the verified override is acting on the backend-assigned severity at the UI layer.

```
// packages/client-api/src/proxy-api/tx-analyze/decorators.ts
export const DecoratedTransactionAnalysis = (
  props: TransactionAnalysisResponseSuccess,
): AnalyseResponse => {
  return props;
};
```

The `TxWarnings` has two render paths. For multiple warnings, it inspects each warning's own severity, but for exactly one warning, it trusts the aggregate `warningSeverity` value that was already overwritten for verified dApps.

```
// apps/web/surfaces/shared/Transactions/TransactionWarnings/TxWarnings.tsx
if (warnings.length === 1) {
  const warning = warnings[0];
  return (
    <Alert variant={isSeverityDangerous(warningSeverity)
    ? 'danger' : 'warning'}>
      {warning.reason}
    </Alert>
  );
}

return (
  <Alert variant={txWarningHigh.length > 0 ? 'danger' : 'warning'}>
    <strong>{warnings.length} Transaction Warnings</strong>
    <ol>
      {warnings.map((warning) => (
        <li key={warning.kind}>{warning.reason}</li>
      ))}
    </ol>
  </Alert>
);
```

There are several cases the verified override can mask, since `isSeverityDangerous()` treats `MEDIUM`, `HIGH`, and `VERY_HIGH` as `danger`:

Backend scanner severity	Nonverified single warning	Verified single warning
VERY_LOW	warning	warning
LOW	warning	warning
MEDIUM	danger	warning
HIGH	danger	warning
VERY_HIGH	danger	warning

Verified status is derived only from the origin hostname matching apps/web/scripts/verified-domains.json, which is generated at prebuild time from the Family init endpoint and contains high-value DeFi front ends such as 1inch.io, aave.com, balancer.exchange, compound.finance, curve.fi, lido.fi, opensea.io, uniswap.org, yearn.finance, zapper.fi, and zerion.io.

```
// apps/web/utils/get-app-origin-details.ts
const hostname = appOrigin
  .replace('https://', '')
  .replace('http://', '')
  .replace('www.', '')
  .split(/[/?#]/)[0];

const verified = Object.keys(verifiedDomains).find((dapp)
    => dapp === hostname);
```

The multiple-warning branch is not affected in the same way — it filters each warning by `warning.severity` and renders `danger` if any individual warning is `MEDIUM` or higher. The separate dangerous-action checkbox driven by `simulation.validation.actionType === 'Dangerous'` is also independent of verified status. The issue is narrower; a verified-origin trust label visually downgrades a high-risk scanner result in the common single-warning path.

Impact

A verified dApp origin that serves an attacker-controlled transaction — through stored XSS, DNS, deployment-pipeline compromise, supply-chain compromise of a script loaded by the front end, or a verified aggregator that fails to constrain user-controlled target / calldata — can issue an `eth_sendTransaction` request that transaction analysis labels with a single `MEDIUM` / `HIGH` / `VERY_HIGH` scanner warning. The wallet computes that severity correctly, then overwrites it to `VERY_LOW` because the origin is verified, and renders the prompt as a yellow `warning` instead of a red `danger`.

Verified DeFi front ends are exactly the sites users already trust enough to approve transactions,

so the downgrade hits the threat model where a clear, severe visual cue is most needed.

Recommendations

Do not let dApp verification status reduce transaction scanner severity. Verified status should be rendered as a separate app-identity signal only — the modal header `VerifiedBadge` is already the appropriate surface. The direct fix is to remove the override:

```
const warningSeverity = getRiskLevel(txWarnings);
```

Also make `TxWarnings` use the warning's own severity for the single-warning path so that the aggregate value cannot mask an individual warning's severity:

```
if (warnings.length === 1) {
  const warning = warnings[0];
  return (
    <Alert variant={isSeverityDangerous(warning.severity)
    ? 'danger' : 'warning'}>
      {warning.reason}
    </Alert>
  );
}
```

If the product wants to use verified status in risk copy, apply it only to a dedicated dApp-identity scanner/category and never to asset-loss warnings. Add regression tests for verified origin, plus one `MEDIUM` / `HIGH` / `VERY_HIGH` warning rendering as `danger`, and for nonverified baselines remaining unchanged.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `ae20fa0e` ↗.

3.13. Parent-side CSS `zoom` rescaling bypasses `ClickjackGuard` visibility check

Target	ClickjackGuard		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `ClickjackGuard` evaluates the modal's visibility from inside the master iframe. Parent-side CSS that scales or clips the iframe is reflected in the child's `IntersectionObserver V2` entry (viewport-edge clipping reduces `intersectionRect`, `transform: scale()` makes `isVisible: false`, occluders make `isVisible: false`), but Chromium's `zoom` rescaling is not reflected.

This is observed on Chrome 148.0.7778.168:

Parent CSS on iframe (800x600 layout)	Top-level `getBoundingClientRect()`	`IntersectionObserver V2` inside child iframe
`transform: scale(0.125)`	100x75	`isVisible: false` (browser detects occlusion or scaling)
`zoom: 0.125`	100x75	`isVisible: true`, `intersectionRatio: 1`, `intersectionRect: 800x600`

The affected component is `ClickjackGuard`:

```
// apps/web/components/common/ClickjackGuard/index.tsx
if ('isVisible' in observerEntry) {
  return {
    ref: customRef,
    status: observerEntry.isVisible ? ElementVisibilityStatus.VISIBLE :
    ElementVisibilityStatus.NOT_VISIBLE,
  };
}
```

Chromium's `zoom` rescales the iframe's physical size at the top level but does not propagate that rescaling into the child document's intersection geometry. With the parent stylesheet applying `zoom: 0.125` to an iframe-sized 800x600, top-level `getBoundingClientRect()` returns `{ width: 100, height: 75 }`, while the child's IO V2 entry reports the following:

```
// chrome, parent iframe styled at zoom:0.125
{ "isIntersecting": true, "isVisible": true, "intersectionRatio": 1,
    "bounding": [800, 600], "intersection": [800, 600] }
```

The `ResizeObserver` guard is also bypassed because it observes the child's own `contentRect`, which remains 800x600.

Impact

A malicious dApp can rescale the integrated modal to roughly an eighth of its real size while every in-frame guard reports the modal as fully visible. The attacker overlays a fake call-to-action aligned with the rescaled `Sign` button's screen position. The user's click on the fake CTA lands at the corresponding internal coordinates inside the iframe, where the real `Sign` button receives it and the approval resolves.

The attack defeats `ClickjackGuard`'s `isVisible` check, the same check the viewport-edge bypass targets.

Here is an attack scenario:

1. A malicious dApp loads in Chromium, and the user connects with Family.
2. The dApp triggers `eth_signTypedData_v4`, `personal_sign`, or `eth_sendTransaction`. The integrated modal mounts as a child iframe.
3. The dApp mutates the iframe's in-line style to `width: 800px; height: 600px; zoom: 0.125`. The modal is rendered as a `100x75` strip at the top level.
4. Inside the iframe, `IntersectionObserver V2` continues to report `isVisible: true`, `intersectionRatio: 1`, `intersectionRect: 800x600`; `ClickjackGuard` returns `VISIBLE` after the `600ms` `isDelayed` activation gate elapses.
5. The dApp draws an attacker-controlled overlay around the rescaled modal directing the user where to click. The browser routes the click to the corresponding position inside the rescaled iframe document, which still has the approval button at its original layout coordinates, and the approval resolves.

Recommendations

The child-frame view of intersection geometry cannot reliably detect parent-side `zoom`. Any measurement performed by the SDK loader script also runs in the dApp's JavaScript realm alongside attacker-controlled code, so values it forwards can be intercepted or rewritten before they reach the master iframe and are not a security boundary.

The primary fix is to route any security-sensitive approval (signature, transaction, chain registration, wallet switch) through a pop-up–mode flow that renders in a separate top-level browsing context the dApp cannot style.

As a defense-in-depth signal, the SDK loader can observe the iframe element with `ResizeObserver` from the parent realm and forward the top-level `getBoundingClientRect()` into the master iframe via `post-robot`. The guard then rejects the modal as `NOT_VISIBLE` when the forwarded dimensions fall below the `350x500` minimum or disagree with the child-document `contentRect`.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `0d4e4f18` ↗.

3.14. Recovery 2FA backup-code submission is wired to the log-in SDK and fails before reaching the server

Target	AuthRecoveryService		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

When a user starts account recovery from `/recovery`, the recovery state machine drives the flow through `sdk.recovery.start()` and `sdk.recovery.submitValidationCode()`. After the server requests 2FA, `RecoverySDK.state.type` becomes `'TWO_FACTOR'`, and the authenticator-code screen correctly calls into the recovery SDK. The "Use Backup Code" branch, however, calls a wrapper that delegates to the log-in SDK instead.

```
// packages/sdk-web/src/services/AuthRecoveryService.ts
public async submitLogin2FARecoveryCode(recoveryCode: string) {
  return this.sdk.auth.loginSubmitTwoFactorRecoveryCode(recoveryCode);
}
```

The recovery-aware wrapper exists in the same service but is unused on the recovery screen.

```
// packages/sdk-web/src/services/AuthRecoveryService.ts
public async submitTwoFactorRecoveryCode(recoveryCode: string) {
  return this.sdk.recovery.submitTwoFactorRecoveryCode(recoveryCode);
}
```

The auth state machine is required by `AuthSDK.loginSubmitTwoFactorRecoveryCode()` to be in `LOGIN_TWO_FACTOR`. A fresh recovery flow never starts the auth log-in flow, so the invariant throws before any HTTP request is made.

```
// packages/sdk-common/src/sdk-interfaces/auth.sdk.ts
private _requireLoginTwoFactorState(): AuthLoginTwoFactorState {
  if (!this.state || this.state.type !== 'LOGIN_TWO_FACTOR') {
    throw new Error(
      'You can not call this method directly without starting the auth login
    process first',
    );
  }
```

The recovery UI calls an incorrect wrapper directly:

```
// apps/web/surfaces/dashboard/recovery/RecoveryTwoFactorRecoveryCode.tsx
export const RecoveryTwoFactorRecoveryCode: FC<Props> = ({ setPage }) => {
  // [...]
  const handleSubmit = async () => {
    // [...]
    const result = await
    authRecoveryService.submitLogin2FARecoveryCode(unhyphenate(recoveryCode));
    // [...]
  };
  // [...]
};
```

The backend and client API already accept `recoveryCode` as an alternative to a TOTP `code` for the recovery 2FA mutation, so the failure is purely a client-side misrouting.

Impact

Users who enabled 2FA, lost access to their authenticator, and still hold their backup codes cannot complete account recovery. The button does not produce a visible API error either; the throw is uncaught in the submit handler, so the screen stays in its loading state with no transition to the next recovery step (device, offline code, CoinCover). This breaks the documented purpose of 2FA backup codes and blocks downstream recovery options that require completing the 2FA step first.

Recommendations

Point the recovery backup-code screen at the recovery SDK wrapper. Either change the UI to call `submitTwoFactorRecoveryCode` directly or repoint `submitLogin2FARecoveryCode` to `this.sdk.recovery.submitTwoFactorRecoveryCode(recoveryCode)` so the wrapper name matches the recovery context. Add a regression test that mocks `sdk.auth` and `sdk.recovery` and asserts the recovery screen routes to the recovery SDK without depending on `LOGIN_TWO_FACTOR` state.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit 73c70207 ↗.

3.15. The `ProxyProvider` allocates an unbounded `MessageChannel` and pending promise per RPC

Target	ProxyProvider, provider-web-worker		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

`ProxyProvider.request()` allocates a fresh `MessageChannel` and a pending promise for every dApp RPC and only closes the port when the worker response arrives. There is no per-request time-out, no `AbortController`, and no cap on concurrent in-flight requests.

```
// packages/sdk-web/src/services/rpc/ProxyProvider.ts
request<T = unknown>({ method, params }: RequestArguments): Promise<T> {
  const channel = new MessageChannel();

  const message: ProxyRpcRequest = {
    url: this.proxyNetwork.rpcURL.value,
    chainId: this.proxyNetwork.chainId.value,
    method,
    params,
  };

  const listener = new Promise((resolve, reject) => {
    channel.port1.onmessage = ({ data }: { data: ProxyRpcResponse }) => {
      channel.port1.close();
      if (data.error) reject(data.error);
      else resolve(data.result);
    };
  });

  this.workerFactory.getWorker().postMessage(message, [channel.port2]);
  return listener as Promise<T>;
}
```

The worker side compounds the pattern: each incoming message triggers a `fetch()` with no abort signal, no time-out, and no response-size limit.

```
// packages/sdk-web/src/services/rpc/provider-web-worker.ts
export const init = () => {
```

```
  const sendRpcRequest = async (rpcUrl: string, method: string, params:
    unknown | undefined) => {
    // [...]
    const response = await fetch(rpcUrl, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(payload),
    });
    // [...]
  };
  // [...]
};
```

A connected dApp can request methods that do not require user approval (`eth_call`, `eth_blockNumber`, etc.) thousands of times per second. Against a slow or attacker-controlled RPC endpoint, every request stays pinned in memory along with its `MessageChannel` port and worker fetch, and approval-critical reads on the same worker stall behind the queue.

Impact

A malicious connected dApp can keep the renderer and RPC worker saturated by flooding nonapproval RPC methods against a slow or attacker-controlled endpoint. Each request leaves a pending promise, a `MessageChannel` port, and a worker `fetch()` alive until a response arrives. This can exhaust memory in the user's own tab, make the wallet UI unresponsive, and starve approval-critical reads such as gas estimation or transaction simulation behind the same worker queue.

The blast radius is limited to that browser tab. The trigger requires the user to connect to the malicious dApp, and closing the tab fully recovers the renderer. No backend component, other user, other tab, or persisted wallet session is affected.

Here is an attack scenario:

1. A user connects Family to a malicious dApp.
2. The dApp selects or uses an RPC endpoint that delays responses.
3. The dApp issues many nonapproval RPC requests such as `eth_blockNumber` or `eth_call`.
4. Family allocates one `MessageChannel`, pending promise, and worker fetch per request with no time-out or in-flight cap.
5. The tab becomes unresponsive, and signing approvals that depend on RPC reads stall or fail.

Recommendations

Add a per-request time-out and close the `MessageChannel` on both success and failure. Pass an `AbortController` signal into the worker-side `fetch()`, and reject requests whose response body exceeds a fixed size cap before JSON parsing. Track in-flight requests per provider, and reject or queue overflow once the cap is reached. Consider adding a per-origin token bucket in the dApp SDK so a connected dApp cannot submit unbounded RPC traffic in the first place.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `5fa3e1c4` ↗.

3.16. Dashboard "Remove Connection" leaves a dApp-side `eth_accounts` cache that still returns the wallet address

Target	EthereumProvider		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The dashboard "Remove Connection" flow calls `sdk.userConnection.reject(appOrigin)`, which deletes the server-side connection row but cannot reach the dApp origin's `localStorage`. The dApp provider answers `eth_accounts` and `isConnected()` from a cached session entry before the lazy reconnection completes, so the address keeps leaking until the background revalidation explicitly fails and clears the cache.

```
// packages/sdk-web/src/react/hooks/sdk/useRemoveConnection.ts
export const useRemoveConnection = () => {
  // [...]
  return useMutation({
    mutationFn: (appOrigin: string) => {
      return sdk.userConnection.reject(appOrigin);
    },
    // [...]
  });
};
```

```
// packages/sdk-dapp/src/sdk/rpc/ethereum-provider.ts
public async request(args: RequestArguments): Promise<any> {
  const cachedSession = cachedSessionStorage.get();
  if (cachedSession && !AaveAccountSdk.isConnected()) {
    if (args.method === 'eth_accounts') {
      return [cachedSession.address];
    }
    if (args.method === 'eth_chainId') {
      return cachedSession.chainId;
    }
  }

  await this.waitForConnection();
```

Background lazy connection only clears the cache after the failure path runs, by which point the dApp script has already received the address.

```
// packages/sdk-dapp/src/sdk/rpc/ethereum-provider.ts
private async waitForConnection(): Promise<void> {
  // [...]
  try {
    // [...]
  } catch (error) {
    logger.error('Failed to establish lazy connection', error);
    if (cachedSession) {
      cachedSessionStorage.remove();
      this._eventEmitter.emit('disconnect');
    }
  }
  // [...]
}
```

The unauthenticated executor path in the web app correctly returns an empty array, but the dApp-side cache short-circuit returns the cached address before that path is reached.

Impact

A dApp that the user removed from the dashboard can still reidentify the wallet address on the first page load after the user revisits that origin. Subsequent privileged RPCs fail once revalidation completes, so this is not a signing bypass, but it breaks the privacy expectation that a removed dApp no longer receives `eth_accounts`.

Recommendations

Do not use `aave_wallet__cached_session` as authority for `eth_accounts`. Restrict the cache short-circuit to `eth_chainId`, wait for authenticated session revalidation before returning a nonempty account list, and emit `accountsChanged([address])` only after the connection is confirmed. Dashboard-side connection removal should also guarantee that session refresh or lazy reconnection checks the current app-origin permission before returning an account.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit 02980547 ↗.

3.17. Sibling iframe can preempt master-iframe `sessionInit` listener and cause a denial of service to the connect flow

Target	InjectedMasterController, post-robot communication wrapper		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `sessionInit` listener is registered by `InjectedMasterController.initialize()` with `domain: '*'` and post-robot's `once`. The handler performs an ancestor-origin equality check against `event.origin` and throws on mismatch:

```
// apps/web/surfaces/injected/injected-master-controller.ts
public static initialize(): Promise<InjectedMasterController> {
  return new Promise((resolve) => {
    void communication().once<Session, SessionInitData>(
      CommunicationTypes.sessionInit,
      { domain: '*' }, // can be started from any origin
      async (event) => {
        const appOrigin = new AppOrigin(event.origin);
        const knownAncestorOrigin = getKnownAncestorOrigin();
        if (knownAncestorOrigin !== null &&
    !knownAncestorOrigin.equals(appOrigin)) {
          throw new Error('Mismatch between the origin of the event and parent
    of the master iframe');
        }
        // [...]
      },
    );
  });
}
```

The defense assumes the handler throws before any state changes. That is incorrect against `post-robot`'s `once`, which cancels the listener *before* the handler executes:

```
// post-robot@10.0.46 src/public/on.js
listener = on(name, options, event => {
    listener.cancel();
    promise.resolve(event);
    if (handler) {
```

```
        return handler(event);
    }
});
```

As soon as a matching `sessionInit` reaches this iframe, the underlying listener is destroyed. The handler's ancestor check does throw on the bad message, but the throw occurs after the listener has already been removed. A subsequent legitimate `sessionInit` from the real parent finds no handler. The first `sdk.connect()` call rejects with the post-robot no-handler error; subsequent retries hit `SdkStatus.CONNECTING` and return a deferred promise that was never resolved because the throw happens before `deferred.resolve()`:

```
// packages/sdk-dapp/src/sdk/sdk.ts:164
async connect(config: ConnectConfig = { environment: production }):
    Promise<void> {
  if (this.sdkState.status === SdkStatus.CONNECTED) throw new
    Error('...already connected...');
  if (this.sdkState.status === SdkStatus.CONNECTING) return
    this.sdkState.deferred;

  const deferred = new Deferred();
  this.sdkState = { status: SdkStatus.CONNECTING, deferred:
    deferred.promise };

  await connect(config);
  await this.initSession();       // throws here on the failed sessionInit
  this.sdkState = { status: SdkStatus.CONNECTED };
  deferred.resolve();
}
```

Cross-frame sends skip the `awaitWindowHello` synchronization whenever `isAncestor(window, win)` is false, which is the sibling-to-sibling case, so an attacker iframe can preemptively spray `sessionInit` to the wallet master iframe without waiting for it to come up:

```
// post-robot@10.0.46 src/public/send.js
if (isAncestor(window, win)) {
    return awaitWindowHello(win, childTimeout);
}
```

A connected dApp is not required. The attacker only needs to be loaded as a cross-origin sibling iframe in the page that embeds the wallet (a compromised ad slot, a third-party widget, an embed).

Impact

A cross-origin sibling iframe loaded with the dApp page can reliably cause a denial of service to the wallet connect flow by emitting `sessionInit` from page load, ahead of the legitimate parent's `sdk.connect()` call (which is typically triggered by user interaction or runs after dApp initialization). The first `sessionInit` from the attacker iframe consumes the `once`; the legitimate `sessionInit` from the real parent afterwards finds no handler. The dApp's `sdk.connect()` first call rejects, and any retry hangs on the unresolved `CONNECTING` deferred. This persists across reloads as long as the attacker iframe is loaded with the dApp.

No fund movement or signing path is involved. For any dApp whose page embeds the attacker as a sibling (third-party widget, ad, embed), the wallet integration becomes unusable for every visitor.

Here is an attack scenario:

1. A dApp embeds Family as one iframe and a third-party widget (attacker-controlled) as another iframe in the same top-level page.
2. The attacker iframe imports `post-robot`, locates the Family master iframe via `window.top.frames`, and posts `{ type: 'sessionInit', ... }` to that frame as soon as it is reachable.
3. The Family `once` listener fires: `listener.cancel()` runs first, and then the handler throws on the origin mismatch. The throw occurs after the listener has already been removed.
4. The dApp's `sdk.connect()` later sends `sessionInit` from the real parent. No handler is registered. The first call rejects with the post-robot no-handler error; retries return the previously assigned `CONNECTING` deferred, never resolved. The dApp's wallet UI stays in its loading state.
5. Reloading reproduces the same outcome.

Recommendations

Constrain the `sessionInit` listener at the registration layer rather than inside the handler. Both `post-robot.on` and `post-robot.once` accept a `window` option that filters incoming messages to a specific `Window` reference (post-robot's on.js reads `options.window` and passes it into `addRequestListener({ win })`). The Family wrapper at apps/web/services/post-robot-service.ts exposes only `domain` on `PostRobotProxyOnOptions`; extend that type with `window?: Window`, and register `sessionInit` with `window: window.parent`. This prevents a sibling iframe from matching the listener at all.

If the listener must keep `domain: '*'` for browsers that do not expose `window.location.ancestorOrigins` (for example, Firefox), drop the `once` wrapper for `sessionInit` and use a regular `on` listener that performs the ancestor / source check first then removes itself only after the check passes.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `f2e7b689` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. At the original scope commit, `apps/web` pinned `next@14.2.33`

At the original in-scope commit `dc3af8dc` ↗, `apps/web` pinned `next` to `14.2.33`, which was inside the affected range of the December 2025 React Server Components / App Router HTTP deserialization advisories (GHSA-mwv6-3258-q52c / CVE-2025-55184 and GHSA-5j59-xgg2-r9c4 / CVE-2025-67779).

```
// apps/web/package.json
    "next": "14.2.33",
```

The issue is relevant even for App Router applications that do not define Server Actions. Next classifies a request as a Server Action purely on `method`, `Content-Type`, and the `Next-Action` header. The presence of any actual Server Action in the app is not required.

```
// next/dist/server/lib/server-action-request-meta.js
const isURLEncodedAction = Boolean(req.method === "POST" && contentType ===
    "application/x-www-form-urlencoded");
const isMultipartAction  = Boolean(req.method === "POST" &&
    contentType?.startsWith("multipart/form-data"));
const isFetchAction      = Boolean(actionId !== undefined && typeof actionId
    === "string" && req.method === "POST");
const isServerAction     = Boolean(isFetchAction || isURLEncodedAction ||
    isMultipartAction);
```

The multipart-fetch branch decodes the request body with `decodeReplyFromBusboy` before performing any action lookup. The flight chunk machinery in `14.2.33` resolves `INITIALIZED` chunks to their value without checking for cyclic thenables, so a chunk whose value is itself triggers an infinite Promise loop on the event loop thread. A four-line POST against any App Router route with `Next-Action: x`, `Content-Type: multipart/form-data`, and a body whose first field has the JSON value `"$@0"` wedges Node's event loop and prevents subsequent requests from being answered.

We reproduced the issue on a minimal Next `14.2.33` App Router application with no Server Actions defined. The request body was 83 bytes:

```
--boundary123
Content-Disposition: form-data; name="0"
```

```
"$@0"
--boundary123--
```

```
curl -X POST \
  -H 'Next-Action: x' \
  -H 'Content-Type: multipart/form-data; boundary=boundary123' \
  --data-binary @body.bin \
  http://127.0.0.1:3939/
```

The `curl` request never returned. The Node process became single-thread saturated, and a follow-up `GET /` to the same server also timed out, so the entire event loop was wedged by one unauthenticated request.

On the original scope commit, an unauthenticated attacker could send the same class of POST to any App Router path on `app.family.co`. Requests for the dashboard, `/popup/auth`, `/popup/rpc`, `/injected/master`, and `.well-known` routes on the same deployment would stop being served until the runtime killed and restarted the process. On Vercel, each affected invocation would burn until function time-out; on a self-hosted `next start` deployment, the worker would need to be killed externally, and sustained traffic could drain all workers.

We confirmed against the updated in-scope commit `d9ffd2082bfaa50172112bb17aa83a98e409cbe0` that apps/web/package.json has been bumped to `next@16.1.6`, which is past both the original fix in `14.2.34` and the incomplete-fix follow-up in `14.2.35` as well as the patched 16.x line. The vulnerable runtime is therefore no longer reachable on the rescope commit. Keeping `corepack pnpm audit --prod` or an equivalent production dependency audit in CI would help future framework advisories surface before deployment.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Wallet and secret material

Description

The wallet package implements the local wallet objects used by the web SDK and the dashboard. The `FAHdWallet` derives an `EtherWallet` from a mnemonic and an index, while `FAPrivateKeyWallet` wraps an imported private key. Both wallet types expose the active Ethereum address and provide local signing helpers. The web signing path later converts the selected wallet's private key into a viem account before signing messages, typed data, or transactions.

The project design assumes that sensitive wallet material is protected before it is stored remotely. The in-memory wallet objects therefore sit at the boundary between encrypted account state and raw signing capability: once a wallet is reconstructed client side, the mnemonic or private key can authorize all assets controlled by that wallet.

Invariants

- A mnemonic or private key should only be available to code that needs to derive an account, export a wallet, or produce a signature.
- A derived HD wallet address must correspond to the stored mnemonic and index.
- Wallet serialization should not include raw key material; it should only include identifiers and derivation metadata needed to reload encrypted wallet state.
- Session teardown and wallet switching should not leave stale signing capability available to a disconnected application.

Test coverage

Cases covered

- Wallet construction, import, and signing behavior in the wallet package tests
- Web wallet integration with the configured browser-side Ethereum wallet implementation
- Encryption-standard setup and wallet primitive validation
- End-to-end wallet flows such as importing, ordering, hiding, and removing wallets

Cases not covered

- Runtime exposure of raw mnemonic or private-key getters to same-origin JavaScript.
- Negative tests proving that non-export code paths cannot read raw wallet material.
- Memory lifetime or disposal behavior after log-out, account switch, or pop-up close.
- Browser adversary models such as XSS, compromised dependencies, and malicious injected scripts running in the same JavaScript realm.

Attack surface

- Any same-origin script that obtains a live wallet object can read raw key material through public getters or TypeScript-private backing fields.
- The signing service converts the active wallet private key into a local signing account, so bugs in approval, session, or origin binding can become direct signing capability bugs.
- HD mnemonics have broader blast radius than a single imported private key because the same mnemonic can derive multiple account indexes.
- Export and recovery flows intentionally handle plaintext wallet secrets, making strict call-site separation important.

5.2. Component: Session and cross-tab state

Description

The session layer tracks authenticated dashboard and SDK state across browser storage, tabs, iframes, and dApp integrations. The `JwtInfoStorageService` stores JWT information under the backward-compatible `family-accounts__jwt_info` key, while `StorageService` stores device information and the device recovery key under `family-accounts__device_info`.

For multi-tab behavior, `SessionSyncService` creates a same-origin `BroadcastChannel` whose name is derived from the relevant app origin and republishes serialized session data whenever `sdk.session.onChange` fires. Other Family contexts on that same origin refresh their local session from that channel. The dApp SDK also keeps local connection state for `eth_requestAccounts` and subsequent account queries.

Invariants

- Access tokens, refresh tokens, ID tokens, device identifiers, device recovery keys, and session keys should only be readable by trusted code for the relevant origin.
- Cross-tab synchronization should preserve session freshness without allowing an unrelated same-origin page to overwrite or resurrect credentials.
- Disconnecting or revoking account access should make future dApp-side `eth_accounts` responses reflect the revoked state.
- Refresh and log-out flows should clear all storage locations that can reauthenticate the user.

Test coverage

Cases covered

- Session SDK state transitions and refresh behavior in the common SDK tests
- Session refresh service behavior for valid and invalid refresh inputs
- SDK bootstrap paths that wire browser storage and SDK services together
- End-to-end authentication, two-factor authentication, device, connection, and revoke flows

Cases not covered

- Adversarial `BroadcastChannel` participants on the same origin
- Tampered or stale serialized session data sent between tabs
- Plaintext local-storage exposure under XSS or compromised same-origin code
- dApp-side cached session behavior after dashboard log-out or permission revocation
- Multi-tab races between refresh, log-out, account switch, and reconnect

Attack surface

- Browser local storage is readable and writable by any script that executes on the same origin.
- The `BroadcastChannel` messages are accepted from same-origin contexts without sender authentication.
- Cached dApp SDK connection state can diverge from dashboard state if revocation does not propagate through all storage and runtime layers.
- Device recovery metadata has account-recovery significance, so corruption or disclosure can affect both log-in continuity and recovery flows.

5.3. Component: dApp RPC and network state

Description

The dApp SDK exposes an EIP-1193–style provider that receives JSON-RPC requests from integrating applications. Requests are routed locally when possible, forwarded to the master iframe for authenticated handling, or opened in a pop-up for approval-sensitive Safari paths. The web app's `JsonRpcService` handles Family-specific methods, including legacy `aave_*` method names, account access, signing requests, chain switching, custom-network addition, permission revocation, and passthrough RPC calls.

For chain-specific calls, `JsonRpcService` keeps an active provider for the requesting app origin. Custom network RPC traffic is sent through `ProxyProvider`, which uses a web worker and validates the configured RPC endpoint by comparing `eth_chainId` with the expected chain ID before storing or selecting the network.

Invariants

- The `eth_accounts` should only return an address for an authenticated and connected dApp session.
- The selected app network should match the chain the user approved for that dApp origin.
- The `wallet_addEthereumChain` should only add networks whose RPC endpoint reports the requested chain ID.
- The `wallet_switchEthereumChain` should not switch a dApp onto a user network without an approval model appropriate for the UI and threat model.
- Passthrough RPC calls should have bounded resource usage and should not let a dApp wedge the wallet runtime.

Test coverage

Cases covered

- The `PostMessage` utility tests around cross-window communication helpers
- SDK bootstrap and end-to-end network flows for adding, editing, removing, and retrieving network information
- End-to-end connection and revocation flows for user-facing SDK behavior

Cases not covered

- Unit tests for `JsonRpcService` approval-required method classification
- Negative tests for silent `wallet_switchEthereumChain` behavior
- Duplicate-chain custom network handling when an attacker controls chain metadata
- Custom native-currency metadata spoofing in approval and transaction review screens
- Time-out, cancellation, and backpressure behavior for proxied RPC requests
- dApp-side stale account cache behavior after permission revocation

Attack surface

- The integrating dApp controls JSON-RPC method names and parameters.
- A dApp adding a custom network controls the network name, RPC URL, block-explorer URL, native-currency display metadata, and chain-ID claim.
- The configured RPC endpoint controls responses to proxied calls and can delay or withhold replies.
- Chain switching changes the execution context for later signing and transaction review, so network state must stay tightly bound to the requesting app origin.

5.4. Component: Signing and transaction review

Description

The signing path is implemented across `JsonRpcService`, `SignerService`, the signing-approval SDK, and the master modal service. dApp requests for `personal_sign`, `eth_signTypedData_v4`, and `eth_sendTransaction` are parsed and validated by `JsonRpcService`, checked against the active wallet address, and then passed to `SignerService`.

The `SignerService` asks the backend-facing signing-approval SDK whether a request requires user approval. Requests that do not require approval are signed locally with the active wallet private key. Requests that require approval are shown through the master modal service before the signature or transaction is produced. Transaction sending uses the currently selected app network, and typed-data signing forwards the supplied EIP-712 domain, types, primary type, and message to viem.

Invariants

- The requested signing address must equal the active wallet address.
- The reviewed network should match the chain on which a transaction will be sent.
- EIP-712 typed data should be bound to the intended chain when the domain contains a `chainId`.
- SIWE-style personal-sign messages should be reviewed against the requesting dApp origin.
- A server-side "approval not required" response should only bypass the modal for requests that are safe under the wallet's policy.

Test coverage

Cases covered

- Browser and SDK end-to-end flows covering representative personal-sign, typed-data, and transaction approval/signing paths
- Address and custom-type parsing tests for Ethereum addresses, signatures, and typed-data–adjacent primitives
- Wallet primitive tests covering local signing behavior

Cases not covered

- Unit tests for `JsonRpcService` signing parameter validation and approval-bypass decisions
- EIP-712 `domain.chainId` mismatch between typed data and the selected app network
- SIWE domain mismatch between message contents and the requesting app origin
- Transaction review downgrade behavior on custom or duplicated networks
- Force-approval and backend auto-approval edge cases for malicious dApp inputs

Attack surface

- The dApp controls the message bytes for `personal_sign`, the full typed-data JSON for `eth_signTypedData_v4`, and transaction fields for `eth_sendTransaction`.
- The approval service response controls whether the user sees a modal before local signing.
- The transaction and signing review UI must faithfully display origin, network, address, value, calldata, and typed-data domain information.
- Any mismatch between reviewed context and signed context can turn a normal approval into a signature or transaction on a different security domain.

5.5. Component: Authentication and recovery flows

Description

The authentication and recovery services coordinate user log-in, two-factor checks, device-based recovery, offline-code recovery, and CoinCover recovery from the web dashboard; `AuthRecoveryService` starts recovery with a sanitized identity, creates a fresh client session key, and stores successful device recovery metadata through `StorageService`.

Recovery and log-in share some credential concepts, but they are separate state machines in the SDK. Validation codes, two-factor codes, two-factor recovery codes, device recovery keys, offline codes, and password credentials must be routed to the correct SDK method for the current flow.

Invariants

- Recovery UI paths should call recovery SDK methods, not log-in SDK methods, after recovery has started.
- A validation code or two-factor code should only advance the state machine it belongs to.
- Successful recovery should persist the new device identifier, device recovery key, and session key consistently.
- Backup-code and offline-code flows should not depend on a preexisting authenticated log-in session.
- Device recovery keys and temporary recovery state should be cleared or replaced when a recovery attempt is abandoned or completed.

Test coverage

Cases covered

- SDK end-to-end flows for authentication start, registration, two-factor, device recovery, offline recovery, and CoinCover recovery

- SDK bootstrap tests that ensure recovery and authentication services are wired into the web SDK
- Custom-type validation tests for recovery codes, recovery keys, validation codes, and credential-related data

Cases not covered

- Dashboard route-level tests proving each recovery screen calls the matching recovery SDK method
- Backup-code recovery routing when log-in and recovery expose similarly named methods
- Recovery state cleanup after user cancellation, page refresh, or failed two-factor submission
- Corrupted local device information and stale recovery-session metadata
- Cross-flow regression tests where log-in state and recovery state coexist in one browser profile

Attack surface

- Users and attackers can supply identities, validation codes, two-factor codes, backup codes, offline codes, and new password credentials through recovery forms.
- Local device information influences whether device recovery is possible and which recovery key is sent to the SDK.
- Similar log-in and recovery method names increase the risk of UI wiring mistakes.
- Recovery flows can regain wallet access, so route-level correctness and state isolation are part of the security boundary.

6. Assessment Results

During our assessment on the scoped Aave Accounts files, we discovered 17 findings. No critical issues were found. Four findings were of high impact, six were of medium impact, and seven were of low impact.

Based on the scope of findings identified during this engagement, we recommend a comprehensive reassessment of the codebase following remediation efforts. This is particularly important because most of the findings sit in one area, the dApp-facing approval and RPC path, and share a single root cause: data supplied by a connected dApp is trusted and shown to the user without first being checked against the network and origin the user is actually on. When the same class of issue turns up across this many separate entry points, it usually means the area needs one systematic pass rather than a patch per method, and that similar issues may remain elsewhere in it.

A follow-up assessment would provide greater confidence in the overall security posture of the application and help ensure that remediation efforts have not introduced new issues.

To maximize the value of a follow-up engagement, we recommend first addressing the following items:

- Review the approval and RPC path as a whole rather than fixing each method on its own. Every signing and transaction approval should be validated against the active network and the requesting origin before it is signed: reject `eth_signTypedData_v4` when `domain.chainId` does not match the active chain, render the EIP-712 `domain` (`verifyingContract` and `chainId`) on every typed-data prompt, verify the SIWE `domain` against the requesting origin in `personal_sign`, require an explicit confirmation for `wallet_switchEthereumChain`, and validate the `wallet_addEthereumChain` `rpcURL` and `nativeCurrency.decimals` against a canonical chain registry. These were reported as separate findings, but they are the same defect repeated.
- Move signing, transaction, and chain add/switch approvals into a pop-up top-level window that the embedding dApp cannot style. The in-iframe `ClickjackGuard` was bypassed two different ways (viewport-edge clipping and CSS `zoom`), and a visibility check made from inside the child frame cannot be relied on against a parent that controls the iframe's layout.
- Stop holding secret material where same-origin script can read it. The JWTs, the device recovery private key, and the session keys are currently kept as plaintext in `localStorage` and broadcast across tabs over an unauthenticated `BroadcastChannel`; the wallet classes also expose `privateKey` and `mnemonic` through always-on getters. Keep keys as non-extractable `CryptoKey` material, move the refresh token to an HttpOnly cookie, and replace the getters with a reason-bound export call.
- Keep the "verified dApp" label as an identity hint only. It currently lowers the transaction scanner's severity, which masks high-risk warnings on exactly the origins users are most likely to trust. The scanner result should reach the user unchanged regardless of verification status.
- Add tests on the paths these findings touch, which have little coverage today: a Permit signed while the wallet is on a different chain than the Permit's domain, a verified origin returning a HIGH scanner warning, and the recovery 2FA backup-code flow.

As a longer-term investment in code quality, we encourage continuing to strengthen security-focused development workflows, including:

- Comprehensive test coverage with end-to-end tests that exercise critical paths and edge cases
- Clear documentation of security assumptions, invariants, and known limitations
- Structured code review processes that explicitly consider security implications

A reassessment following these improvements would allow for deeper analysis and provide greater assurance for production deployment.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.