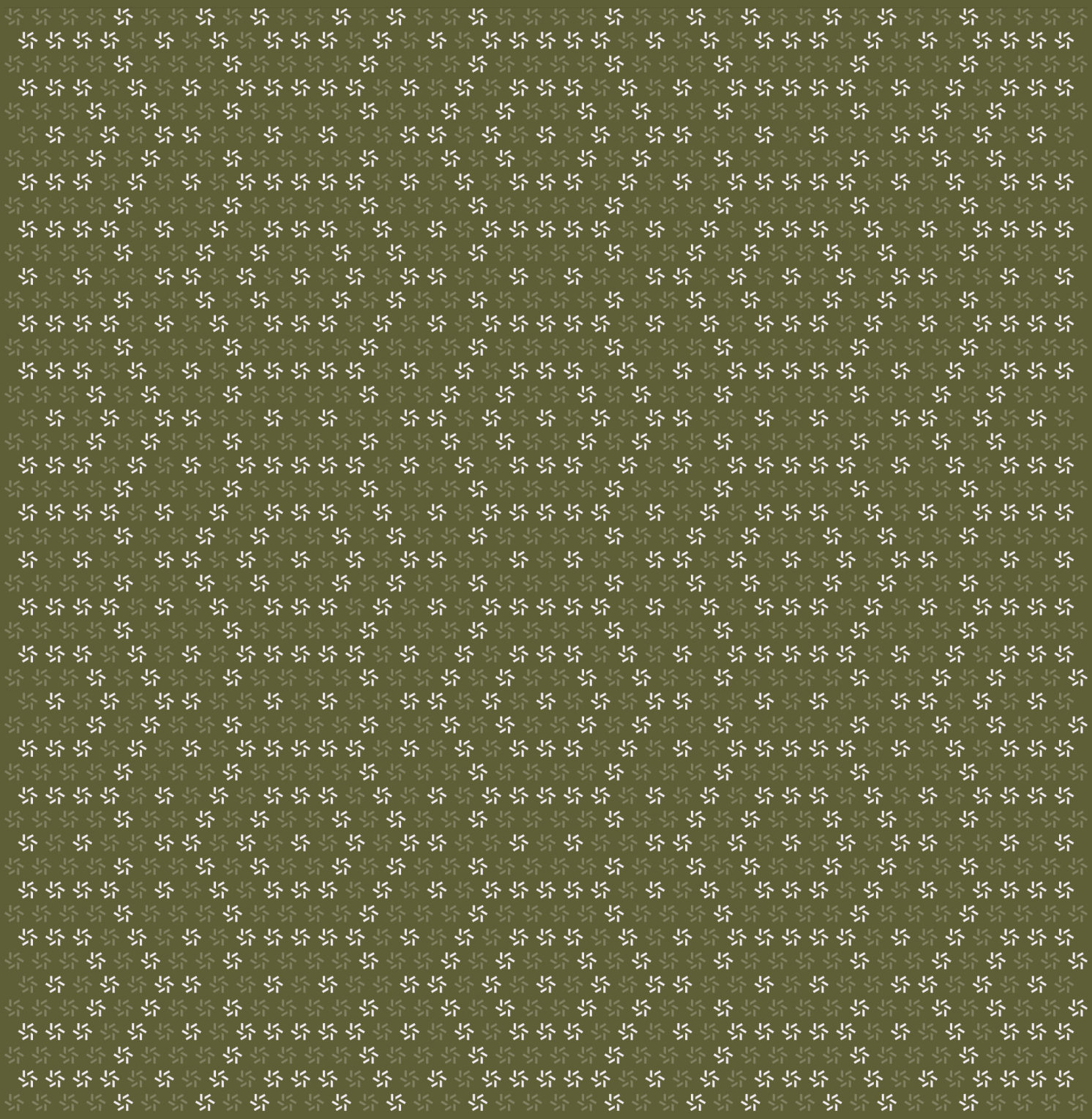
Zellic

Prepared for
Aave Labs
Aave Labs

Prepared by
Bradan Beaver
Nathanial Lattimer
Zellic

May 26, 2026

Aave Accounts
Web Wallet 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 13th to May 14th, 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 2 and presents the associated findings and observations. Component 1 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:

- How does authentication and authorization of requests to the backend work?
- Was server-side request forgery (SSRF) accounted for in the handling of web-accessible addresses?
- Does the API account for cross-site–scripting (XSS) risks, despite not being the typical attack surface in which one would find that kind of vulnerability?

1.3. Non-goals and Limitations

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

- dApps users might interact with
- Security posture of third-party components
- Infrastructure used in deployment of the application

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

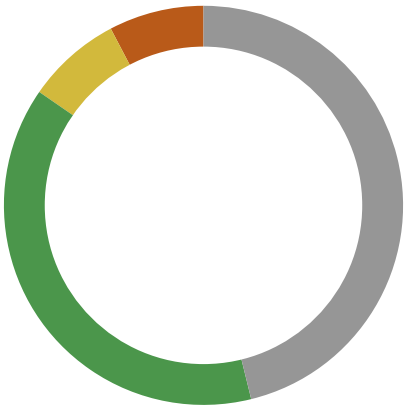
1.4. Results

During our assessment on the scoped Aave Accounts files, we discovered 13 findings. No critical issues were found. One finding was of high impact, one was of medium impact, five were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	5
Informational	6



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

Types	TypeScript, Swift, Kotlin, C++
Platforms	Web, iOS, Android
Target	family-account
Repository	https://github.com/family/family-account ↗
Version	`d9ffd2082bfaa50172112bb17aa83a98e409cbe0`

Programs	`apps/server/src/**/*`
	`packages/sdk-ios/**/*`
	`packages/sdk-react-native/**/*`
	`packages/custom-types/**/*`

2.4. Project Overview

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

Bradan Beaver
Engineer
bradan@zellic.io

Nathanial Lattimer
Engineer
d0nut@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 13, 2026	Start of primary review period
April 14, 2026	Kick-off call
May 19, 2026	End of primary review period

3. Detailed Findings

3.1. Heap overflow in scrypt JNI binding allocates a single byte

Target	sdk-react-native (Android JNI scrypt)		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Android JNI entry point that exposes `libscrypt_scrypt` to Kotlin allocates the output buffer with an incorrect `new` form.

At `packages/sdk-react-native/android/src/main/cpp/libscrypt.cpp:102`,

```
std::unique_ptr<uint8_t> buffer(new uint8_t(buffer_length));

int scryptError(libscrypt_scrypt(
  reinterpret_cast<const uint8_t *>(pass.nativePointer),
    static_cast<size_t>(passLength),
  reinterpret_cast<const uint8_t *>(salt.nativePointer),
    static_cast<size_t>(saltLength),
  n, r, p,
  buffer.get(), buffer_length));
```

`new uint8_t(buffer_length)` is the non-array `new`. It allocates *one* `uint8_t` on the heap and value-initializes it to `buffer_length` truncated to eight bits. The intended form is `new uint8_t[buffer_length]`. Then, `buffer.get()` is passed to `libscrypt_scrypt` together with the original `buffer_length`, and `libscrypt_scrypt` writes `buffer_length` bytes into a one-byte allocation.

The JavaScript caller invokes this path during every log-in through bridged-encrypted-standard.ts with `buffer_length = 32`, so every successful Android login triggers a 31-byte heap write past the end of a one-byte allocation. In practice, the Android allocator rounds the one-byte allocation up to a small bin (typically 16 bytes under `jemalloc` / Scudo), so the effective overflow is roughly 16 bytes into the next live allocation on the same heap arena.

A second, latent issue compounds the bug: the buffer is held in `std::unique_ptr<uint8_t>` (non-array deleter). If a future change "fixes" the allocation to `new uint8_t[buffer_length]` without also switching the `unique_ptr` to `std::unique_ptr<uint8_t[]>`, the resulting `delete` (instead of `delete[]`) is undefined behavior. Using `std::vector<uint8_t>(buffer_length)` avoids both pitfalls in one step.

Impact

This is a preauthentication heap overflow on the normal log-in path. Every Android client that completes a scrypt KDF writes attacker-influenced bytes (the scrypt output is a function of the user-supplied password and salt) past the end of a small heap allocation. Observed and likely consequences include the following:

- Silent corruption of an adjacent Kotlin/JNI allocation, producing incorrect-looking scrypt output and locking legitimate users out of their accounts
- Process crashes (`SIGSEGV`) that surface as flaky, hard-to-reproduce log-in failures
- Memory corruption suitable for exploitation — the overflow contents are derived from controllable inputs (password plus salt) and land in a heap arena that holds JNI references and allocator metadata

The derived key is also not zeroized before the owning `unique_ptr` goes out of scope, leaving a copy of the user master key resident in freed heap memory until that region is reused.

Recommendations

Replace the allocation with an array form and a matching owner. Either of the following resolves the buffer bug:

```
auto buffer = std::vector<uint8_t>(buffer_length);
// ...
buffer.data();
```

```
std::unique_ptr<uint8_t[]> buffer(new uint8_t[buffer_length]);
```

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit 959f818b ↗.

3.2. Shared `javax.crypto.Cipher` field is not thread-safe and can swap key/IV across concurrent callers

Target	sdk-react-native (Android AES adapter)		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The Android AES adapter at `packages/sdk-react-native/android/src/main/java/com/avaralabs/aaa/reactNative/adapter/AES.kt:53` caches a single `javax.crypto.Cipher` instance in a class field and reinitializes it on every `encrypt` / `decrypt` call:

```
class AES : PocketDevelopmentKitModule.AESProvider {
  private val cipher = Cipher.getInstance(
    "AES/GCM/NoPadding"
  )

  override fun encrypt(data: ByteArray, key: ByteArray, iv: ByteArray):
    ByteArray {
    val keySpec = buildKeySpecFrom(key)
    cipher.init(Cipher.ENCRYPT_MODE, keySpec, buildGCMParameterSpecFrom(iv))
    return cipher.doFinal(data)
  }

  override fun decrypt(data: ByteArray, key: ByteArray, iv: ByteArray):
    ByteArray {
    val keySpec = buildKeySpecFrom(key)
    cipher.init(Cipher.DECRYPT_MODE, keySpec, buildGCMParameterSpecFrom(iv))
    return cipher.doFinal(data)
  }
  // ...
}
```

The class `AES` is constructed once and held on `PocketDevelopmentKitModule` as a long-lived `AESProvider`. The RN module dispatches crypto operations on `Dispatchers.Default`, which is a multithreaded coroutine pool sized to the device core count, and `javax.crypto.Cipher` is documented as *not* thread-safe. Per the JCE specification, "a Cipher object in a particular state (for example, after initialization) is not thread-safe".

Two concurrent callers therefore race on the same `cipher` field. The two-line `init` / `doFinal` body is not atomic, so the following interleaving is reachable:

```
Thread A: cipher.init(ENCRYPT, keyA, ivA)
Thread B: cipher.init(ENCRYPT, keyB, ivB)   // overwrites A's state
Thread A: cipher.doFinal(dataA)         // produces ciphertext under keyB / ivB
Thread B: cipher.doFinal(dataB)          // throws (state already consumed) or
    returns garbage
```

A failed call after `init` but before `doFinal` (e.g., the `InvalidKeyException` thrown when `key.size` is not 16/24/32 or `IllegalBlockSizeException` for short GCM ciphertext) leaves the shared `cipher` in an unknown state. The next legitimate caller inherits that state and produces undefined output.

The companion `RSA.kt` in this package correctly allocates a fresh `Cipher.getInstance("RSA/...")` on every call. The iOS `AES.swift` similarly creates per-request `CryptoKit` primitives. This file is the outlier.

Impact

Under any concurrent load on the SDK's crypto path, the following are reachable:

1. Caller A's plaintext is encrypted under caller B's key/IV. AES-GCM still authenticates correctly from the algorithm's perspective, so the failure is silent.
2. An `IllegalStateException` is thrown when `init()` interrupts an in-progress `doFinal`, producing flaky crashes under load that are extremely hard to reproduce in isolation.
3. Truncated/garbled ciphertext where the AEAD tag fails verification on decrypt is reachable, surfacing as silent data loss to the user.

Because the cipher is reused after exceptions, a single malformed input from one caller (incorrect key size, undersized GCM ciphertext) can corrupt the state observed by the next caller, amplifying a single malformed request into a data-integrity issue for an unrelated session.

Recommendations

Construct a fresh `Cipher` instance per operation, matching the RSA adapter and the iOS implementation:

```
override fun encrypt(data: ByteArray, key: ByteArray, iv: ByteArray):
    ByteArray {
  val cipher = Cipher.getInstance("AES/GCM/NoPadding")
  cipher.init(Cipher.ENCRYPT_MODE, buildKeySpecFrom(key),
    buildGCMParameterSpecFrom(iv))
```

```
  return cipher.doFinal(data)
}
```

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit 959f818b ↗.

3.3. Image proxies accept `image/svg+xml` and parameter-smuggled content types — no `nosniff` or CSP on the response

Target	server (REST image proxies)		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The two REST image proxies, token-logo and favicon, both gate the upstream response on `contentType.startsWith('image/')` and forward the upstream `content-type` verbatim, with no `Content-Security-Policy` configured on this API. The affected call sites are as follows:

Proxy	Location
`getTokenLogo`	`apps/server/src/rest/proxy/token-logo.ts:166`
`getFavicon`	`apps/server/src/rest/proxy/favicon.ts:170`

Both call sites look like this:

```
const contentType = imageResponse.headers.get('content-type');

if (!contentType || !contentType.startsWith('image/')) {
  res.status(500).json({ error: 'Error ...' });
  return;
}

res.setHeader('content-type', contentType);
// ... pipe upstream body to client
```

The main problem with the `startsWith('image/')` gate is that **`image/svg+xml` is accepted**. SVG is an XML document that can carry `<script>` with embedded HTML, `onload` / `onerror`, and external resource references. Token-logo serves attacker-influenced upstream content from `https://token-logos.family.co`; favicon serves attacker-influenced upstream content from `https://www.google.com/s2/favicons?domain=<attacker>`. When loaded as an `<img>`, the script does not execute, but the same URL is reachable through `<iframe src="…">`, `<object data="…">`, direct navigation, or right-clicking "open image in new tab", all of which execute script in the FA origin's context.

Finally, the family-account API surface that serves these proxies has **no `Content-Security-Policy` header configured**. Even after pinning the content-type allowlist, a CSP would be a meaningful defense-in-depth backstop; a response that ships with `Content-Security-Policy: default-src 'none'; sandbox` (or, at minimum, `script-src 'none'; object-src 'none'`) cannot execute script under the FA origin regardless of how it ends up rendered. The current responses ship no CSP at all, so any future regression in the content-type gate immediately becomes an XSS on the FA origin.

Impact

An attacker can land an XSS on the family-account origin by inducing a victim to load `/proxy/favicon?domain=attacker.example` (or, for token-logo, by causing the upstream `token-logos.family.co` to serve an SVG for an attacker-chosen token ID) in a context that executes script.

Recommendations

We recommend the following.

Replace `startsWith('image/')` with an explicit allowlist, normalized to lowercase and parameter-stripped before compare. For both proxies —

```
const ALLOWED = new Set([
  'image/png',
  'image/jpeg',
  'image/gif',
  'image/webp',
  'image/vnd.microsoft.icon',
  'image/x-icon',
]);

const normalized = (contentType ?? '').split(';')[0].trim().toLowerCase();
if (!ALLOWED.has(normalized)) { /* 500 */ }
```

`image/svg+xml` must be excluded.

Add a `Content-Security-Policy` to this API surface. At the time of writing, the responses ship no CSP at all, which means any regression in the content-type allowlist is a direct XSS. Apply the CSP at the API/middleware layer rather than per handler so future proxies inherit it.

Also pin `res.setHeader('content-type', normalized)` rather than echoing the raw upstream string, so future upstream changes cannot reintroduce parameter smuggling through this proxy.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `1dc2938b` ↗.

3.4. HMAC key for GraphQL `CursorScalar` is hardcoded in source

Target	server (GraphQL Cursor scalar)		
Category	Cryptography	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The GraphQL `CursorScalar` signs and verifies pagination cursors with an HMAC key that is checked into source control as a `const`.

See `apps/server/src/graphql/scalars/cursor.scalar.ts:52`:

```
const SECRET_KEY = '<secret key string>';

// ...
serialize: (cursor: unknown): string => {
  const encodedString = Buffer.from(cursor.stringValue).toString('base64');
  const signature = sign(encodedString, SECRET_KEY);
  return `${encodedString}.${signature}`;
},

parseValue: (value: unknown): Cursor => {
  const [receivedCursor, receivedSignature] = value.split('.');
  if (!verify(receivedCursor, receivedSignature, SECRET_KEY)) {
    throw new AuthenticationError('Cursor has been modified');
  }
  // ...
}
```

Anyone with read access to the repository, such as past and current contributors, can recover the key. A constant HMAC key signing attacker-supplied input is, in practice, equivalent to no signing at all.

A `Cursor` wraps an arbitrary `UnknownObject` that each resolver parses against its own zod schema. Cursors are routinely used to carry tie-breaker fields (`created_at`, `id`), filter criteria, and in some resolvers, user-scoped identifiers (`userAccountId`, `walletId`). Where a resolver trusts a cursor field as a value that has already been authenticated on the previous page, forging the signature breaks that trust boundary.

Impact

Anyone who has ever seen the source code can forge arbitrary cursor payloads. Depending on the resolver, this allows skipping result windows, sort injecting, or pivoting a cursor-scoped query to another `userAccountId` / `walletId` that the attacker is not authorized for. Every resolver that decodes a cursor and uses its fields as a trusted oracle becomes implicitly vulnerable.

Recommendations

Load the HMAC key from the same secret store the rest of the server uses for sensitive material, and call `invariant` at startup to assert that it is nonempty and of the expected length. Do not provide a constant fallback.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit ec3c505a ↗.

3.5. Authentication material is compared in nonconstant time

Target	server (S2S middleware, JWT service)		
Category	Cryptography	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

Two server paths compare authentication material with operations that short-circuit on the first byte mismatch. These are the three call sites:

Compared value	Location	Operator
S2S API key vs configured allowlist	`apps/server/src/middleware/s2s-api-key-middleware.ts:52`	`Array.prototype.includes` (uses `===`)
JWT-bound device-ID hash vs header-derived device-ID hash	`apps/server/src/services/jwt.service.ts:412`	`DeviceIdHash.equals(...)` (inherits `BaseType.equals`, which uses `===`)
Query-provided user identity vs database-stored user identity	`apps/server/src/services/user.service.ts:235`	`Identity.equals(...)` (inherits `BaseType.equals`, which uses `===`)

S2S API key middleware

Javascript's `Array.prototype.includes` walks the array and uses `===` against each entry. JavaScript's `===` on strings is a byte-by-byte comparison that returns on the first mismatch, so the call's wall-clock time is a monotonic function of how many leading bytes of `providedKey` match an entry of `S2S_API_KEYS`.

```
const providedKey = req.header(API_KEY_HEADER);

if (!providedKey || !S2S_API_KEYS.includes(providedKey)) {
  res.status(401).json({ error: 'Unauthorized' });
  return;
}
```

BytesType equals inheritance

The `DeviceIdHash` type extends `BytesType` via the custom-type hierarchy, but `equals` it inherits depends on the base class,

```
const headerDeviceIdHash = DeviceIdHash.fromDeviceId(deviceId);
const jwtDeviceIdHash = new DeviceIdHash(result.deviceIdHash);

if (!headerDeviceIdHash.equals(jwtDeviceIdHash)) {
  throw new AuthenticationError(
    'Jwt token has a deviceId but it does not match the provided deviceId',
  );
}
```

and `BaseType.equals` (`packages/custom-types/src/shared-bases/base.type.ts:38`) is as follows:

```
public equals(other: this): boolean {
  return other.value === this.value;
}
```

Aave Labs additionally noted that the `Identity` type similarly extends `BytesType` via the custom-type hierarchy, reachable through `isSameIdentity` in `user.service.ts`.

```
const isSameIdentity = (identity: Identity, rawIdentity: RawIdentity) => {
  ...
  if (identity.equals(providedIdentity)) {
    return {
      normalized: rawIdentity,
      hasMatch: true,
    };
  }
}

...

export async function verifyRawIdentity(
  userAccountId: UserAccountId,
  rawIdentity: RawIdentity,
): Promise<VerifiedUsernameUser | VerifiedExternalAccountIdUser> {
  const user = await requireUserAccountByUserId(userAccountId);

  const { normalized, hasMatch } = isSameIdentity(user.identity, rawIdentity);
  ...
}
```

Impact

Timing side channels over a network is difficult but not impossible to exploit. With a stable RTT path and a large enough sample size, an attacker can recover the compared secret one byte (or a few bits) at a time.

Recommendations

In `s2s-api-key-middleware.ts:52`, replace the array `includes` with a constant-time scan over Buffers.

Remediation

This issue has been acknowledged by Aave Labs, and fixes were implemented in the following commits:

- `69b0a30a` ↗
- `826050f9` ↗

3.6. Precision loss on JSON round trip since `NumberType.validate()` accepts nonfinite and unsafe numbers

Target	custom-types (NumberType)		
Category	Soundness	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `NumberType` accepts any IEEE-754 double as long as `typeof value === 'number'`. The validator at `packages/custom-types/src/shared-bases/number.type.ts:25` is as follows:

```
export class NumberType extends BaseType<number> {
  constructor(value: number) { super(value); }

  protected validate(): void {
    if (typeof this._value !== 'number') {
      throw new Error(`Number supplied is not type number -
    ${String(this._value)}`);
    }
  }

  public toJSON(): number { return this._value; }

  public toString(): string {
    return this._value.toString();
  }
}
```

The `typeof` in JavaScript returns `'number'` for the values `NaN`, `Infinity`, and `-Infinity`, all of which pass `validate()` unchanged. Likewise, every integer larger than `Number.MAX_SAFE_INTEGER` ($2^{53} - 1$) is a valid `number` to the runtime but cannot be exactly represented as an IEEE-754 double. Constructing `new NumberType(9007199254740993)` stores 9007199254740992, and any downstream consumer that JSON-serializes and deserializes the value will observe the drifted value with no error.

Subclasses inherit this — `ChainId` overrides `validate()` to reject values `< 1`, but `NaN < 1` is `false`, so `NaN` passes the chain-ID check too; `ValidationCode` and other `NumberType` subclasses do not override `validate()` at all and inherit the same hole.

Impact

Any code path that constructs a `NumberType` from external input (JSON, query parameter, GraphQL scalar) can land `NaN`, `Infinity`, or an unsafe integer in the store. Comparisons and arithmetic on `NaN` silently fail (every comparison except `!==` returns `false`), so guard logic that uses a `NumberType` value can be bypassed in surprising ways. Numbers above `Number.MAX_SAFE_INTEGER` round-trip with silent precision loss; a `ChainId` of `9_007_199_254_740_993` becomes `9_007_199_254_740_992` after a JSON round trip, producing cross-system value drift that is hard to detect.

Recommendations

Tighten `validate()` to require both `Number.isFinite` and `Number.isSafeInteger` (or, where fractional values are legitimate, just `Number.isFinite`).

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `c8412e9c` ↗.

3.7. The `InitialisationVector` is fixed at 16 bytes; AES-GCM canonical IV length is 12

Target	custom-types (InitialisationVector)		
Category	Cryptography	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `InitialisationVector` (`packages/custom-types/src/initialisation-vector.type.ts:25`) hardcodes the required IV length to 16 bytes for AES-GCM, both in `getRequiredByteLength()` and in the static `generate` helper:

```
export class InitialisationVector extends BytesType {
  constructor(initialisationVector: Uint8Array<ArrayBuffer> | string) {
    super(initialisationVector);
  }

  protected getRequiredByteLength(): number {
    return 16;
  }

  public static generate(randomBytes: RandomBytesFunction):
    InitialisationVector {
    return new InitialisationVector(randomBytes(16));
  }
}
```

NIST SP 800-38D specifies 96 bits (12 bytes) as the canonical IV length for AES-GCM. Implementations are allowed to accept other lengths, but values other than 12 trigger the GHASH-based IV derivation path inside GCM. That path is still secure when the IV is sampled from a CSPRNG, but it 1) is nonstandard and 2) reduces the birthday bound on accidental IV collisions from $\sim 2^{48}$ to $\sim 2^{64}$ when sampled uniformly at random. AES-GCM's single most important invariant is `(key, IV)` uniqueness; a single collision in IV under the same master key leaks the keystream XOR of the two messages.

Impact

At sustained encryption volumes against a single master key, the probability of an IV collision under random 16-byte selection is small but nontrivial.

Recommendations

Change `getRequiredByteLength()` and `generate(...)` to 12 bytes.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `87139d64` ↗.

3.8. Gas-prices proxy does not validate `chainId` against `SupportedChain`

Target	server (gas-prices proxy)		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The gas-prices proxy at `apps/server/src/rest/proxy/gas-prices.ts:62` coerces `chainId` to a number and forwards it directly into the upstream Family API path without checking it against the `SupportedChain` enum that other proxies (e.g., estimate-tx.ts) use as a gate:

```
export const getGasPrices = async (req: Request, res: Response) => {
  const chainIDResult = z.coerce.number().safeParse(req.params.chainId);

  if (!chainIDResult.success) {
    res.status(422).json({ error: 'Invalid chain id provided, must be a
    number' });
    return;
  }

  const uri = `${FAMILY_API_ENDPOINT}/v3/${chainIDResult.data}/gas-prices`;
  const response = await fetch(uri, { headers: { 'x-api-key':
    FAMILY_API_KEY } });
  // ...
};
```

The `z.coerce.number()` schema is broad: it accepts negative integers, floats, scientific notation, very large numbers, and anything that `Number(value)` does not return `NaN` for. Each of those values is interpolated verbatim into the path segment of the upstream URL. Here are examples of inputs that pass the current check:

- `chainId=0` → `/v3/0/gas-prices`
- `chainId=-1` → `/v3/-1/gas-prices`
- `chainId=1.5` → `/v3/1.5/gas-prices`
- `chainId=1e10` → `/v3/10000000000/gas-prices`

The sibling endpoint estimate-tx.ts already validates the same input against `SupportedChain`. This endpoint is the odd one out.

Impact

The exposed surface is path injection into the upstream Family API; a caller can drive the server to request arbitrary `/v3/<n>/gas-prices` URLs against `FAMILY_API_ENDPOINT`.

There is no direct in-server vulnerability because `URL`-style path interpolation here does not enable a host or scheme change, and `URLSearchParams`-encoded query values are unaffected.

Recommendations

Validate `chainId` against the same `SupportedChain` enum the rest of the proxies use.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `6cd479b2` ↗.

3.9. The `Signature` does not enforce low-`s` (EIP-2) canonicalization

Target	custom-types (Signature)		
Category	Cryptography	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `Signature` custom type at `packages/custom-types/src/signature.type.ts:33` validates only that the input is 65 bytes long (`r || s || v`) and stores the raw value:

```
export class Signature extends BytesType {
  constructor(signedData: string | Uint8Array<ArrayBuffer>) {
    super(signedData);
  }

  protected getRequiredByteLength(): number {
    return 65;
  }

  public splitSignature(): EtherSignature {
    return splitSignature(this.value);
  }
}
```

ECDSA over Secp256k1 admits two valid `s` values for any signed message: a low-`s` form (in the lower half of the curve order) and its negation in the upper half. EIP-2 (Homestead) mandates the low-`s` form as canonical on Ethereum mainnet, but the underlying cryptography still verifies the high-`s` variant. The `Signature` accepts either at construction.

This is a defensive concern rather than the root cause for an active vulnerability in the codebase.

Impact

Replay deduplication keyed on raw signature bytes can be bypassed by submitting the high-`s` variant of an already seen signature.

No direct exploit path is established by this finding alone; the impact is determined by how individual call sites use the signature bytes downstream.

Recommendations

At construction, enforce low-s canonicalization.

Remediation

This issue has been acknowledged by Aave Labs.

3.10. The `BytesType` accepts a `Uint8Array` without copying, leaving the backing buffer mutable

Target	custom-types (BytesType)		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

At `packages/custom-types/src/shared-bases/bytes.type.ts:51`, `BytesType.parseValue` is the only path through which raw bytes enter a `BytesType` (or any of its subclasses such as `Signature`, `PrivateKey`, `AccountMasterKey`, `CredentialAuthToken`, etc.). When the input is a `Uint8Array`, the function returns the array verbatim:

```
const parseValue = (value: string | Uint8Array<ArrayBuffer>):
    Uint8Array<ArrayBuffer> => {
  if (typeof value === 'string') {
    if (!isHexString(value)) {
      throw new Error(`String supplied is not a hex string - ${value}`);
    }
    return arrayify(value);
  } else if (isUint8Array(value)) {
    return value;
  }
  throw new Error('Value supplied is not a valid hex string or uint8array');
};
```

The `Uint8Array` is a view onto a `SharedArrayBuffer` / `ArrayBuffer`; nothing prevents the caller from continuing to hold a reference to the same backing buffer and mutating it after the `BytesType` has been constructed. Because the `BytesType` instance stores the same `Uint8Array` reference in `_value`, a postconstruction mutation by the caller silently changes the supposedly immutable wrapped value. Since `validate()` ran once at construction time and never reruns, a now invalid value (incorrect length, zeroed bytes, attacker-controlled replacement) lives in a `BytesType` that still claims to be valid.

The string path is safe; `arrayify(value)` allocates a fresh `Uint8Array`. Only the `Uint8Array` branch shares state with the caller.

Impact

A caller that reuses an input buffer (a streaming decoder, an in-place mutation in a loop, a returned slice from `Buffer.subarray`) can land the `BytesType` in a state that violates its length and equality invariants without any subsequent revalidation.

Recommendations

Clone the input in the `Uint8Array` branch:

```
} else if (isUint8Array(value)) {
  return new Uint8Array(value);
}
```

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `11bb0071` ↗.

3.11. The `AppOrigin` localhost dev bypass tests `host` instead of `hostname`, rejecting `http://localhost/`

Target	custom-types (AppOrigin)		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `AppOrigin.validate()` method (`packages/custom-types/src/app-origin.type.ts:56`) carves out an exception for `http://localhost` origins to allow local development. The check uses the parsed URL's `host` field, which always includes the port if one is present.

The URL parser sets `host` to `'localhost:<port>'` when a port is supplied and to `'localhost'` (no trailing colon) when one is not. Therefore, `startsWith('localhost:')` matches only the port-bearing form. The same pattern is mirrored in the `get isLocalhost()` accessor (`app-origin.type.ts:99`).

The intended check uses the `hostname` field, which omits the port:

```
parsedUrlResult.value.hostname === 'localhost'
```

Impact

The check does not enable any privilege escalation patterns. This issue merely rejects a form of localhost it should accept.

Recommendations

Switch the comparison from `host` to `hostname`.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `c3ccfad6` ↗.

3.12. `BaseType.setValue` writes `_value` without rerunning `validate()`

Target	custom-types (BaseType)		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `BaseType.setValue` method (`packages/custom-types/src/shared-bases/base.type.ts:45`) assigns to the protected `_value` field directly and does not rerun `validate()`:

```
protected _value!: TValue;
protected constructor(value: TValue, byPassValidation = false) {
  this.setValue(value);
  if (!byPassValidation) {
    this.validate();
  }
}
// ...
protected setValue(value: TValue): void {
  this._value = value;
}

protected abstract validate(): void;
```

The constructor calls `setValue` and then explicitly calls `validate()` afterwards, so initial construction is safe. The issue is that `setValue` exists as a separate `protected` method, which advertises it as a legitimate mutation primitive. Any subclass that calls `this.setValue(...)` after construction or any subclass that reassigns `this._value` directly inside its own `validate()` (which `AppOrigin.validate()` does) bypasses validation entirely. There is no place in `BaseType` that revalidates after `setValue`.

Impact

A future subclass or refactor that calls `this.setValue(...)` outside the constructor lands the object in a state that has not run through `validate()`, breaking the invariant the type advertises.

Recommendations

Make `setValue` run validation, matching the constructor's contract.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `11bb0071` ↗.

3.13. Transaction proxies deny valid upstream content types

Target	server (REST transaction proxies)		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

Aave Labs identified that the two REST transaction proxies, `estimate-tx` and `analyze-tx`, both gate the upstream response on "`=== 'application/json'`" and forward the upstream `content-type` verbatim. The affected call sites are as follows:

Proxy	Location
`estimateTx`	`apps/server/src/rest/proxy/estimate-tx.ts:68`
`analyzeTx`	`apps/server/src/rest/proxy/tx-analyse.ts:81`

Both call sites look like this:

```
const contentType = response.headers.get('content-type');

if (!contentType && contentType !== 'application/json') {
  logger.error(`No content type received from estimate endpoint`);
  ...
}

res.setHeader('content-type', contentType);
// ... pipe upstream body to client
```

Two issues were noted here:

- The `&&` comparison should instead be `||` to check for content type being either undefined or an unaccepted value; and,
- Tenderly's actual `content-type` response header is `application/json; charset=utf-8`; a strict equality fix as noted in Finding 3.3. ↗ would fail

Impact

Response JSON would not be mishandled through character set smuggling, and upstream URLs are allow-listed based on transaction chain; therefore, identified impact is negligible, with minor availability loss, and this finding has been noted as *informational*.

Recommendations

As Aave Labs identified this finding, remediations were completed proactively. Remediation was performed by changing the comparison operator from `&&` to `||`, performing a `startsWith` comparison for the `content-type` response header, and not forwarding upstream `content-type` verbatim.

Remediation

This issue has been acknowledged by Aave Labs, and a fix was implemented in commit `fe81a427` ↗.

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. JWT implementation best practices

In line with the request for a key focus on the JWT implementation of the components raised by Aave Labs, Zellic observed several areas where the implementation could be further improved. These observations are informational in nature and do not impact the system at the time of writing. We provide here suggestions for best practice.

A notable concern is the use of a shared signing key for multiple JWT purposes, such as access tokens, refresh tokens, and temporary tokens. Even when each token contains different role or payload fields, signing them all with the same secret increases the risk of token confusion — a more pertinent concern due to the lack of schema validation of the JWT payload. A token intended for one flow may be accidentally accepted in another if a verifier is too permissive or if future code changes introduce overlapping claims. A safer approach is to use separate signing secrets for each token type. Validating token payloads by schema, rather than the current pattern of individually typecasting claims, would further alleviate this risk.

The verification logic should also explicitly enforce registered claim values, such as algorithms, issuer, audience, and token IDs. Calls to `jwt.verify(token, JWT_SECRET)` rely on library defaults and do not clearly communicate which algorithms are permitted or which issuer and audience are expected. The best practice is to specify the accepted algorithm and to validate `iss` and `aud` during verification. Refresh tokens are reusable until their expiry and do not include replay protection or server-side invalidation through a token identifier such as the aforementioned `jti` claim. Long-lived refresh tokens are high-value bearer credentials, so they should generally be rotated upon every use and invalidated after rotation.

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: Accounts Server

Description

The server application exposes a GraphQL API that is responsible for user management and authentication, transaction requests (such as classifying and approving well-known transactions), recovery flows, storing encrypted blobs, and integrating with third-party services such as CoinCover. The well-known transaction implementation classifies transactions, performs transaction simulations, and auto-approves transactions where possible. The server additionally exposes legacy REST API routes for backwards compatibility, allowing for `GET` requests for wallet information and transaction analysis/estimation.

Invariants

- GraphQL scalar inputs/outputs must have the expected type and schema.
- Authentication and recovery flows must have the required credential material, such as `passkeyInfo` and `passkeyCredentialId`.
- Request/session context and infrastructure operations must produce required values.
- Network and on-chain related code enforces that selected networks and chain identifiers are valid and supported.

Test coverage

Cases covered

- SQL injection and validity testing for query helpers (`ConditionalFormat`)
- Hashing correctness for `DeviceIdHash`, `Identity`, `IpAddressHash`, and `JwtHash`
- CoinCover status valid and invalid transitions
- Util helpers, such as `canonicalizeIP` and `requireSafeUrlValue` (largely unused)

Cases not covered

- API-level testing (routes and GraphQL handlers)
- Handling of external provider connections and outputs
- Well-known transaction flows (Test cases added post-review in commit f6760eefdc ↗)
- Token and authentication flows

- Malformed token handling and schema validation
- Load testing and concurrency issues

Attack surface

- The server exposes a GraphQL route for performing queries on the database.
- A proxy REST API exposes handlers for wallet functions, CoinCover flows, and RPC.
- User credentials are supplied using JSON Web Tokens (JWTs).
- HTTP request headers are validated and utilized by the server in service handlers, such as `User-Agent` strings for JWT expiry timing.
- Integration to third-party services and RPCs such as CoinCover, Alchemy, and Tenderly.

6. Assessment Results

During our assessment on the scoped Aave Accounts files, we discovered 13 findings. No critical issues were found. One finding was of high impact, one was of medium impact, five were of low impact, and the remaining findings were informational in nature.

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.