Why I Wrote an AAD Canonicalization Spec

March 7, 2026

You know that sinking feeling when a perfectly working system suddenly starts rejecting valid data? Now imagine that system is your encryption layer, and the “invalid data” is actually legitimate ciphertext that your users are so desperately needing to decrypt.

Welcome to the world of Additional Authenticated Data (AAD) serialization bugs—where two systems can fully agree on *what* data should be authenticated, but somehow manage to disagree on *how* to represent that data as bytes, and silently your entire encryption pipeline goes down the drain.

The aforementioned is exactly why I chose to sit down one day (well, actually it was many days) and write my own formal specification for AAD canonicalization.

No more “just use JSON bro”, as that is about as helpful as “just use encryption” when it comes to actual security engineering.

Imagine That!

Picture this: a multi-tenant secrets management service. Each tenant’s secrets are encrypted with AEAD[1], and the AAD contains contextual binding—ya know, the tenant ID, resource path, purpose. Standard AAD material. The encryption process looked something like this:

```
aad = json.dumps({"tenant": "foobar_corp", "resource": "secrets/database/prod",
"purpose": "encryption-at-rest", "v": 1}, separators=(',', ': '))
ciphertext = aead_encrypt(key, nonce, plaintext, aad.encode('utf-8'))
```

Looks good? It’s a no-go, in Go.

Go constructed the *exact same* logical AAD context. Same tenant, same resource, same purpose, same version. But when it came time to decrypt secrets, every single decryption failed with an authentication error.

The Culprit: Serialization Non-Determinism

> *What a mouthful that is.* ***Serialization non-determinism*** *occurs when a serialization format permits multiple valid byte representations for the same logical data structure — meaning identical inputs can produce different outputs depending on the implementation, language, or even invocation order.*

Look here at what Python’s `json.dumps()` produced:

```
{"tenant": "foobar_corp", "resource": "secrets/database/prod", "purpose":
"encryption-at-rest", "v": 1}
```

And look at what Go's `json.Marshal()` yields for the same logical data:

```
{"purpose":"encryption-at-
rest","resource":"secrets/database/prod","tenant":"foobar_corp","v":1}
```

Spot the differences? Two things happened. First, Go's `encoding/json` sorted the map keys lexicographically—`purpose` before `resource` before `tenant`. Second, Python's default separators (`', ', ': '`) inserted spaces after every colon and comma, while Go produced compact output with no whitespace at all.

These are semantically identical JSON documents. Any JSON parser will produce the same data structure from either one. But AEAD doesn't care about semantics. It cares about bytes. And these byte sequences are completely different.

The authentication tag is computed over the exact byte sequence of the AAD. Change a single byte and the verification fails. This isn't a bug; it's exactly how AEAD is supposed to work.

Why This Matters More Than You Think

Okay, so why not just pick one serialization format?

In theory, it's simple — standardize on compact JSON, enforce it everywhere, call it a day. In '99, a similar kind of assumption turned the Mars Climate Orbiter into a $327 million lesson in what happens when two systems agree on *what* to exchange but not *how* to represent it.

The problem isn't choosing a format. It's enforcing that choice across every language, library, team, and service that touches your encryption pipeline — and ensuring that enforcement holds as those systems evolve independently. This breaks down in two specific ways.

The Multi-Language Problem

Modern systems are polyglot. Your encryption might happen in a Python pipeline, decryption in a Go microservice, and KMS in a Rust sidecar. Each language has its own JSON library with its own quirks…

Even within the same language, different JSON libraries behave differently. And we haven't even gotten to numbers yet![2]

The Multi-Tenant Risk

In a multi-tenant system, AAD is what prevents cross-tenant decryption. The tenant identifier in the AAD ensures that even if an attacker obtains a ciphertext from Tenant A, they can't trick the system into decrypting it under Tenant B's context. It's not a nice-to-have — it's a security boundary.

But serialization inconsistency quietly erodes that boundary. You encrypt with `{"tenant": "A", "resource": "data"}` and attempt to decrypt with `{"resource": "data", "tenant": "A"}`. Same fields, same values — authentication failure. And once that failure is baked into a production pipeline, every developer's favorite concept makes an appearance: "fallbacks."

If you have to start implementing "lenient AAD matching", you've fallen to the dark side. It's the cryptographic equivalent of disabling your seatbelt because it's uncomfortable.

The Canonicalization Solution

The answer to non-deterministic serialization is *canonicalization*—defining a single, unambiguous byte representation for any given data structure. This isn't new; it's exactly why RFC 8785 (JSON Canonicalization Scheme) exists.

JCS defines exactly how JSON serialization should and should not be implemented:

- Keys sorted lexicographically (the order they would appear in a dictionary) by UTF-16 code units
- No whitespace between tokens
- Strict rules for number formatting and string escaping

With JCS, both Python and Go will produce the exact same bytes for the same logical JSON object:

```
{"purpose":"encryption-at-
rest","resource":"secrets/database/prod","tenant":"foobar_corp","v":1}
```

So then what's the issue?

Why RFC 8785 Isn't Enough

JCS solves the *serialization* problem, but it doesn't tell you *what* exactly to serialize. For AAD specifically, we still need answers to:

What fields should be in the AAD?

Without standardization, one service might include `"tenant": "foobar_corp"` while another includes `"{"tenant_id": "foobar", "tenant_name": "Foobar Corp"}`. Both are "correct" but incompatible.

How do you handle versioning?

What if the corporation needs to add a new field?

How do existing ciphertexts remain decryptable?

How do you prevent downgrade attacks?

What about application-specific fields?

Different applications need different context. A secrets manager needs different AAD than a message queue. How do we allow extensibility without the risk of collisions?

You get the idea. Hence, I wrote a full specification.

The Specification Philosophy

The spec I wrote (and that this series will be based on) makes several opinionated design choices. Let's take a peek at the thought process behind a few of them:

Flat Structures

No nested objects. Ever.

```
// ✓ Good
{"purpose": "encryption", "resource": "secrets/db", "tenant": "acme", "v": 1}

// ✗ Bad
{"context": {"tenant": "acme"}, "purpose": "encryption"}
```

Why? Nested structures create additional serialization ambiguity (key ordering at each level) and make validation more complex. AAD should be *metadata*, not a data model. If you need complex structures, you're probably putting too much in your AAD[3].

Required Contextual Binding Fields

Every AAD object *must* contain:

- `v` (version): Schema version for forward compatibility
- `tenant`: Tenant or user identifier
- `resource`: What's being protected
- `purpose`: Why it's being protected

This isn't arbitrary. Each field prevents a specific attack vector, which will be covered in Part 2. The short version (if you're lazy or simply don't care): without `tenant`, you get cross-tenant decryption. Without `purpose`, you get context confusion. Without `resource`, you get substitution attacks.

Strict Type Constraints

All values are either strings (UTF-8, no null bytes) or unsigned 64-bit integers.

No arrays. No nulls. No booleans. No bullshit.

This eliminates entire categories of type confusion and canonicalization edge cases. It also makes validation straightforward, as each field can be checked independently.

Namespaced Extensions

Applications can add custom fields, but only with a specific prefix pattern: `x_<application>_<field>`. For example, `x_vault_cluster` or `x_payments_region`.

This prevents collisions between the core schema and application-specific fields, while still preserving extensibility.

Introducing the Specification

These principles are codified in the **AEAD Additional Authenticated Data (AAD) Canonicalization Specification**, currently at version 1.2.

From the abstract:

> *"This specification defines a canonical schema for Additional Authenticated Data (AAD) used with Authenticated Encryption with Associated Data (AEAD) algorithms. It leverages RFC 8785 (JSON Canonicalization Scheme) for deterministic serialization and defines required contextual binding fields to prevent confused deputy attacks and cross-context ciphertext reuse."*

In non-robotic terms: The spec addresses the three problems I outlined earlier–non-deterministic serialization across ecosystems, lack of standardized contextual binding fields, and ambiguous handling of edge cases like Unicode normalization and key ordering.

The spec is intentionally narrow—it defines AAD *contents* only. How AAD travels alongside ciphertext, nonce, and key identifiers is an envelope format concern, explicitly out of scope.

What's Next to Come

This series walks through the entire specification, end-to-end.

Part 2: Threat Modeling AAD covers the *why*; the attacks that motivated each design decision.

Part 3: The Devil in the Details covers the *how*; implementation pitfalls made in production systems.

Key Takeaways

1. **AAD is byte-sensitive:** Semantically identical data with different byte representations will cause authentication failures. This is a good thing.
2. **JSON serialization is non-deterministic:** Different languages, libraries, and even versions produce different bytes for the same logical structure.
3. **Canonicalization is necessary but not sufficient:** JCS solves serialization, but you still need a schema defining the *what* & *how* of serialization and validation.
4. **Cross-service encryption requires coordination:** If your encryption spans across multiple services, languages, or teams, you need a specification—not just documentation.
5. **Leniency in AAD handling is a security anti-pattern:** If the idea of adding fallback logic for your AAD mismatches crosses your mind, you've done it wrong. Fix the serialization instead.

In Part 2, we'll explore the threat model in depth: what attacks does properly constructed AAD actually prevent, and what happens when it's misconfigured?

References

Dworkin, M. (2007). *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D. Link

Google Cloud. (2024). *Additional authenticated data*. Google Cloud KMS Documentation. Link

McGrew, D. & Viega, J. (2004). *The Security and Performance of the Galois/Counter Mode (GCM) of Operation*. INDOCRYPT 2004. Link

Rescorla, E. (2008). *An Interface and Algorithms for Authenticated Encryption*. RFC 5116, IETF. Link

Rogaway, P. (2002). *Authenticated-encryption with associated-data*. Proceedings of the 9th ACM Conference on Computer and Communications Security. Link

Rundgren, A. (2020). *JSON Canonicalization Scheme (JCS)*. RFC 8785, IETF. Link

Albertini, A., et al. (2022). *How to Abuse and Fix Authenticated Encryption Without Key Commitment*. USENIX Security 2022. Link

Footnotes

1. AEAD (Authenticated Encryption with Associated Data) combines encryption with integrity verification. The “associated data” (AAD) is authenticated but not encrypted—meaning it’s protected against tampering but remains readable. Common AEAD algorithms include AES-GCM and ChaCha20-Poly1305. The key property relevant here is that decryption will fail if any input differs from encryption—including the AAD bytes. ↩
2. Number serialization in JSON is surprisingly complex. JavaScript’s `Number.MAX_SAFE_INTEGER` is 2^{53}-1, which is smaller than many systems’ native 64-bit integers. An integer like `9007199254740993` will round to `9007199254740992` in JavaScript but serialize correctly in Python or Go. JCS builds on ECMAScript’s IEEE 754 double-precision serialization rules, but you still need to decide whether your application permits integers that large. ↩
3. The “flat structure” rule has one significant implication: you cannot express hierarchical relationships within AAD. If your system genuinely needs to bind ciphertext to nested context (e.g., organization → team → project → resource), you have two options: (1) flatten the hierarchy into a path-like string (`"org/team/project/resource"`), or (2) include multiple fields (`org`, `team`, `project`, `resource`). The spec recommends option 2 when you need to query or filter on individual levels. ↩

---

https://blog.gtfo.dev/blog/aad-canonicalization-why-it-matters/