



End-to-End Message Encryption

A draft standard for message-level encryption across messaging platforms

Matt Pavlovich



Rob Parker



One message, many systems

An enterprise runs many brokers and protocols. A single message often crosses several of them between the application that sends it and the one that reads it.



Many systems and protocols

ActiveMQ, IBM MQ, Kafka, RabbitMQ, and cloud services, spoken over AMQP, MQTT, STOMP, OpenWire, and more.



Messages cross between them

Bridges, routes, and gateways move a message across brokers and protocols on its way from producer to consumer.



One standard travels with it

Encrypt at the producer, decrypt at the consumer. The protection rides in the message, independent of any broker or hop.

Transport security protects one hop at a time. An end-to-end standard protects the message across every system it touches, with one scheme every platform can implement.

01

How we encrypt today

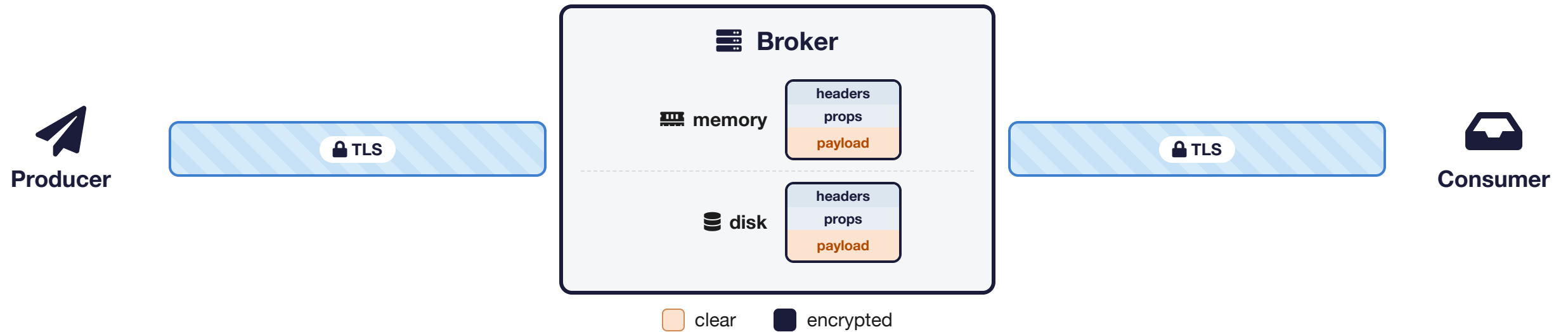
Anatomy of a message



- Headers** Protocol fields: destination, delivery mode, timestamps
- Custom properties** Application metadata the broker routes and filters on
- Payload** The application data - the part worth protecting

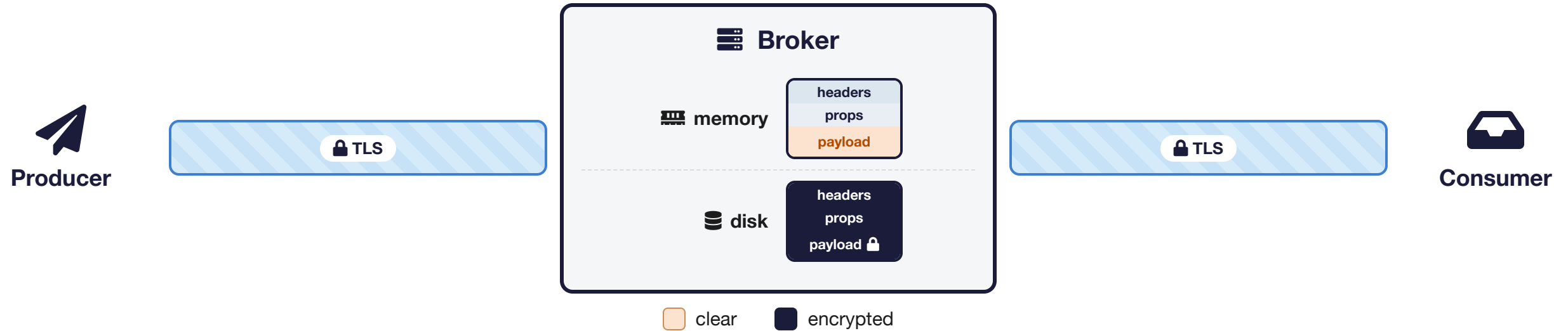
Some protocols combine headers and custom properties into a single segment. The token on the next slides tracks all three and shows where the payload is encrypted.

Transport encryption (TLS/SSL)



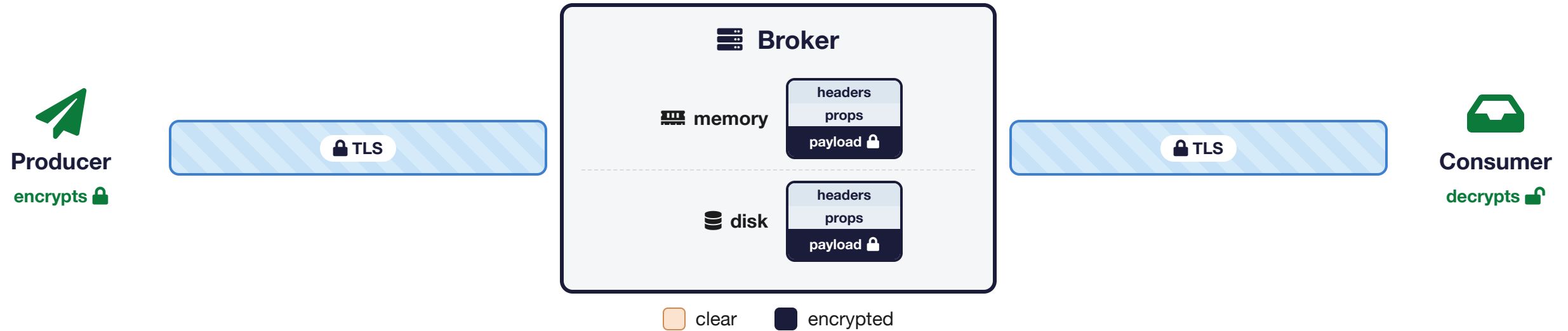
TLS encrypts the tunnel, not the message. The producer and consumer tunnels are separate, and the broker holds the message in the clear, in memory and on disk.

Broker-side encryption (data at rest)



Encrypting at rest protects the disk: physical theft, backup and restore, and the replication surface. The broker still processes the whole message in the clear in memory.

End-to-end message encryption



The producer encrypts the payload; only the consumer decrypts it. Headers and properties stay clear for routing, so the broker never sees the payload - in memory or on disk.

02

The cryptography

Classical vs post-quantum

Role	Classical	Post-quantum
Key exchange	RSA, ECDH, DH	ML-KEM (FIPS 203)
Signatures	RSA, ECDSA	ML-DSA (FIPS 204), SLH-DSA (FIPS 205)
Payload cipher (symmetric)	AES-256-GCM, ChaCha20-Poly1305	Unchanged - already PQC-safe
Integrity / hash	SHA-256, SHA-384, SHA-512	Unchanged - already PQC-safe

NIST finalized ML-KEM, ML-DSA, and SLH-DSA in August 2024.

What this means for message encryption



The payload uses symmetric crypto.

AES-256-GCM encrypts the body, and AES-256 is already post-quantum safe



The quantum-vulnerable part is key exchange.

Deriving or transporting the data key online is where ML-KEM matters



Pre-shared keys sidestep online key exchange.

A key referenced by id, provisioned out of band, removes the vulnerable step entirely



So a pre-shared key plus AES-256-GCM is post-quantum safe today

03

Toward a standard

Draft: what gets encrypted

-  **Properties stay clear by default** so routing, selectors, and observability keep working
-  **The payload is ciphertext** sealed producer to consumer
-  **Encrypted properties are optional** for sensitive metadata, carried alongside the clear ones
-  **A named header carries the non-secret encryption metadata** needed to decrypt

Draft: what the metadata carries

A small set of non-secret fields lets a consumer decrypt. Some are algorithm-specific, so the set must stay open.

Field	Why	Notes
kid	Which key to use	Pre-shared key, provisioned out of band; the hook for key rotation
alg	Cipher and mode	e.g. AES-256-GCM; lets the scheme evolve
iv	Per-message uniqueness	base64 or hex, told apart by length given alg
pack	How the payload is packed	Transform pipeline, e.g. gzip+aes+b64
v	Format version	Forward compatibility

Fields like `iv` are crypto-algorithm specific, so the schema keeps a small core and stays open to algorithm-specific fields.

Draft: four approaches to carry it

Approach	Metadata on the wire	Shape
No header (opaque)	None - the consumer knows everything ahead of time	payload only
Header fields	Readable key=value pairs	v=1; alg=...; kid=...
URI	One URI value	enc://<data>?kid=...
Token	One base64 JSON token	eyJ2Ijox...

Key rotation drives this: each message names its key (kid), so producers roll keys while consumers keep decrypting old and new. No-header forces lock-step rotation, out of band.

Option: header fields

A named header describes how to decrypt, carrying no secret material. The payload is ciphertext; the GCM authentication tag travels with it.

```
encryption: v=1; alg=AES-256-GCM; kid=orders-2026-q4; iv=<base64|hex>; pack=gzip+aes+b64
```



kid

Pre-shared key id. Names the key; the key itself is provisioned out of band



alg

Cipher and mode, e.g. AES-256-GCM. Lets the scheme evolve without guesswork



iv

Initialization vector / nonce. Non-secret, unique per message

Option: enc:// URI

A self-contained encrypted field: the scheme marks it, the host carries the base64url ciphertext, and every field is a query parameter.

```
enc://<base64url-ciphertext>?v=1&kid=orders-2026-q4&alg=AES-256-GCM&iv=<base64|hex>&pack=gzip+aes+b64
```



enc://

Scheme marks encrypted content and can version the format



host = ciphertext

The base64url payload; empty when the content is the message body, which travels separately



query = fields

v, kid, alg, iv, pack, and any extension, each a named parameter

Option: compact token (JSON + base64)

Like an OAuth2 / JWT token: the fields are a small JSON object, base64url-encoded into a single opaque header value.

```
encryption: eyJ2IjoxLCJhbGciOiJBdDTSImtpZCI6Im9yZGVycy0yMDI2  
LXE0IiwiaXYiOiJ4N0tMMm85UWlxWnIiLCJwYWNrIjoiz3ppcCthZXMrYjY0In0  
  
# base64url decodes to  
{"v":1,"alg":"AES-256-GCM","kid":"orders-2026-q4","iv":"x7KL2o9Qb1Zr",  
"pack":"gzip+aes+b64"}
```



One opaque value carries the whole descriptor, like a bearer token



Structured and extensible - add fields to the JSON without changing the grammar



Header-safe - base64url avoids delimiter and charset pitfalls in a single header value

Draft: packing

Packing is an ordered pipeline of transforms applied to the payload. The consumer reverses it. The wire message type is independent and auto-detected.

pack	Pipeline	Wire	Message type
aes	encrypt	bytes	BytesMessage
gzip+aes	compress, encrypt	bytes	BytesMessage
aes+b64	encrypt, base64	text	TextMessage
gzip+aes+b64	compress, encrypt, base64	text	TextMessage

The pack spec is authoritative. The consumer auto-detects BytesMessage vs TextMessage and reverses the pipeline, so a producer can choose the wire form for compatibility without changing how the message is unpacked.

04

Lab

Lab exercises



Round-trip an encrypted message through ActiveMQ on the demo site, mqsummit.hyte.io



Repeat against an unchanged IBM MQ broker over AMQP 1.0



Inspect the broker: confirm the payload is ciphertext and properties are clear



Encrypt a property and carry it alongside the clear routing properties



Review the draft points and capture next steps

Going further: encrypting properties

Clear (default)

routable and observable

region = us-east-1

Encrypt the value

value hidden; key still routes and filters

region =  enc://kR9xL2bQ8vYn

Encrypt key and value

key and value each sealed; not routable

 enc://pT1mA9cZ =  enc://kR9xL2bQ8vYn

An enc:// prefix marks an encrypted field, followed by base64url ciphertext - so each field stays a text-safe property and the consumer knows to decrypt it.

**Encrypt the value to protect sensitive metadata while the key still drives routing and selectors.
Encrypt the key too when even the property name is sensitive - giving up broker-side routing on it.**

Going further: asserting integrity

A hash over the ciphertext lets any hop verify the bytes without the key. A hash over the plaintext lets the consumer confirm the decrypted data end to end. Carry both as clear fields.

```
integrity: alg=HMAC-SHA-256; kid=orders-2026-q4; ct=<base64>; pt=<base64>
```



Hash of the ciphertext

ct covers the encrypted bytes. Any hop, including the broker, verifies the payload is intact without holding the key, catching tampering in transit and at rest.



Hash of the plaintext

pt covers the decrypted bytes. The consumer confirms end to end that what it decrypted is exactly what the producer sent, across every repack and hop.



Keyed, not bare

Compute both with HMAC keyed by kid, or a signature. A plain digest catches accidental change; only a key proves nothing was tampered with.

Small or low-entropy payloads: a bare pt digest invites a dictionary attack. A keyed MAC over pt closes it.

Key rotation, management, and policy



Key rotation

Each message names its key with `kid`, so producers roll to a new key while consumers keep decrypting old and new. No coordinated cutover.



Key management

Keys live in a registry or KMS, provisioned out of band and referenced by `kid`. Rotate, revoke, and scope keys per destination, tenant, or data class.



Policy enforcement

Set which destinations must be encrypted, which algorithms are allowed, and the minimum key strength. Enforce at the producer and audit from the clear `kid` and `alg`.

The clear `kid` is the hook for all three: it names the key per message, so rotation, management, and policy operate without ever reading the payload.

Performance: reuse the decrypt engine

The cipher is rarely the bottleneck; building its engine on every message is. Expanding the key is the costly step, so do it once per key, not once per message.



Engine per message

Build a fresh decrypt engine for each message.

- Pays the key schedule and allocation every message
- Simple and stateless, but adds latency and GC churn
- Throughput falls as message volume climbs



Engine per consumer

Initialize one engine per consumer and run N messages through it.

- Key is expanded once, only the per-message iv changes
- Cache engines by kid, so rotation just adds one
- One engine per consumer thread, no shared locking

Pub-sub: keys for many consumers

Point-to-point pairs one producer with one consumer. Pub-sub fans out to many, and they subscribe on demand, so the one encrypted copy must be readable by consumers the producer never named.



The challenge

One ciphertext, many subscribers. Every subscriber needs the key, and sharing one symmetric key across all of them makes distribution and revocation the hard part.



CA-anchored certificates

Release the key to any consumer presenting a valid certificate from a trusted CA. On-demand subscribers need a cert, not a pre-shared key, and revocation runs through the CA.



Stand-alone key pairs

Wrap the key to each consumer's own public key. Tight per-consumer control and revocation, but every consumer must be known ahead of time.

Standards governance

Messaging already has standards bodies. The wire protocols sit with OASIS; the JVM API sits with the Eclipse Foundation. An e2e encryption standard needs a home among them.



AMQP

OASIS AMQP 1.0, ratified as ISO/IEC 19464. The wire-level interoperability standard.



MQTT 5.0

OASIS MQTT 5.0. Its user properties carry the metadata; 3.1.1 has no message properties, so target 5.0.



Jakarta Messaging

Eclipse Foundation, Jakarta EE. Current release 3.1 (`jakarta.jms`), the messaging API for the JVM.

A message-level encryption scheme crosses all three. It fits best as a shared profile each body can adopt, so the header and `enc://` forms stay identical across protocols and APIs.

Discussion

Thank you.

**MQ
Summit**

mqsummit.com

Reference

The demo pipeline

Encrypt with openssl, send with HYTE pub, receive with pub, decrypt with openssl. Same message, unchanged brokers. Backs the hosted demo at mqsummit.hyte.io.

```
# Encrypt on stdin, send with pub (ActiveMQ, AMQP 1.0)
echo "secret order #42" \
  | openssl enc -aes-256-gcm -K "$KEY" -iv "$IV" -a \
  | pub send amqs://activemq:5671/queue/orders \
    -f - -H "encryption: v=1; alg=AES-256-GCM; kid=lab; iv=$IV"

# Same pipeline, unchanged IBM MQ – just a different AMQP URL
... | pub send amqs://ibmmq:5671/queue/DEV.QUEUE.1 -f - -H "encryption: ..."

# Receive body to stdout, decrypt with openssl
pub receive amqs://activemq:5671/queue/orders \
  | openssl enc -d -aes-256-gcm -K "$KEY" -iv "$IV" -a
```