
Protecting secrets

Key takeaways

1. A bearer credential lets its holder act as a principal. Protecting it requires privacy from readers and integrity against replacement.
2. Authenticated encryption protects secret bytes and binds visible record fields to them. The consumer must reject a record whose check fails.
3. Encryption narrows trust to keyholders and plaintext consumers. Any process that can see either one remains able to disclose or misuse the secret.

Common misunderstandings of secret protection

Which claim would you defend right now? Choose before the evidence.

“Encryption prevents an attacker from replacing a stored credential.”

“A valid authentication tag proves which holder of the shared key created the record.”

“Once a service token is encrypted, a process that decrypts it no longer needs special trust.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

The stored deployment token

A deployment worker saves a service token now and another process uses it later

- The token
 - A bearer credential accepted by the inventory service.
- The stored record
 - The deployment worker writes bytes that a scheduled consumer will read.
- The adversary
 - Can read and replace the stored file, but does not know the secret key.



The record must conceal the token and make unauthorized replacement fail.

Instapoll

Instapoll · Review · Oct 22 A

A service encrypts an API token before storing it. Someone who cannot decrypt the token can still replace bytes in the stored file. What must the consumer check before using the decrypted value?

- A. The encryption key has remained private
- B. The stored file still has the original name
- C. An authenticated integrity check covering the protected data succeeds
- D. The ciphertext differs from the plaintext

Privacy and integrity protect different losses

The same token record can fail in two independent ways

PRIVACY

Blocked reading

PUBLIC FILE

🛡️ **Stored token record**

public ciphertext = "Qx7kP9..."



Unauthorized reader

Reads the file; has no secret key



Token value stays unknown

Ciphertext conceals the bearer credential

INTEGRITY

Detected replacement

SAME FILE THREAT

Attacker replaces bytes

inventory → billing; retains ciphertext + public tag



Consumer checks the record

Secret key + protected bytes



REJECT

Changed bytes do not match the tag

Privacy blocks disclosure; authenticated integrity detects changed protected bytes.

Trust names a function and a possible loss

The deployment worker is trusted only for what its failure can damage



- Function
 - Create a token record that the later consumer can use.
- Required behavior
 - Keep the token private and bind the intended record fields.
- Loss
 - Disclosure lets another party use the token. Replacement can redirect the consumer.

Trust becomes concrete when we can name the function and the loss caused by failure.

Systems make decisions about principals

Authentication identifies a principal; authorization decides what it may do



- **Principal**
 - An identity the system can distinguish, such as a person, service, or process.
- **Authentication**
 - Establish which principal is making the request.
- **Authorization**
 - Apply policy to decide whether that principal may perform the requested action.

A bearer token stands in for its holder

The inventory service accepts possession of the token as authority to call its API

```
Authorization: Bearer eyJhbGciOi...
```

```
request + accepted token
```

```
-> authenticate the token's principal
```

```
-> apply that principal's permissions
```

- Any party that possesses a bearer token can use it, subject to the token's scope and the service's policy.
- The stored token carries the principal's effective reach. Reading it can become impersonation.
- Encryption protects the stored representation. Scope, expiration, and revocation limit what a stolen token can do.

Encryption hides plaintext from a reader without the key

The nonce may be public; the secret key and plaintext stay inside the trusted boundary

```
ciphertext = Encrypt(secret_key, nonce, token)

store:    nonce, ciphertext
recover: Decrypt(secret_key, nonce, ciphertext) -> token
```

- The secret key controls recovery of the token and must remain unavailable to the file attacker.
- The nonce follows the encryption algorithm's rule. Reusing a nonce with the same key can destroy privacy for common modes.
- Encryption by itself does not tell the consumer whether stored bytes were replaced.

A MAC detects unauthorized change

Verification recomputes a keyed tag and rejects a mismatch

Illustrative: unchanged record

✓ **Public protected bytes + tag**

inventory | $42 \cdot \text{tag} = \text{"b79e..."}$

🔑 **Verify tag**

Secret MAC key

ACCEPT

Unchanged bytes

Illustrative: changed record

✗ **Changed bytes + same public tag**

billing | $42 \cdot \text{tag} = \text{"b79e..."}$

🔑 **Verify tag**

Same secret MAC key

REJECT

Changed bytes; old tag

Limits: no secrecy · any MAC-key holder can mint tags · replay needs a separate freshness check.

The tag is public; only a MAC-key holder can mint a tag for different protected bytes.

AEAD protects secret bytes and authenticates context

Authenticated encryption binds ciphertext and visible associated data under one key

```
ciphertext, tag = Seal(key, nonce, token, associated_data)
```

```
Open(key, nonce, ciphertext, associated_data, tag)
```

```
    verification failure -> reject
```

```
    success                -> release token
```

- The token becomes ciphertext. Associated data stays visible but any change to it makes verification fail.
- The consumer must verify before using released plaintext and must treat failure as a hard rejection.
- AEAD does not add freshness by itself. A protocol needs a sequence number, timestamp, or stored state when replay matters.

The tag binds metadata to the token

PUBLIC MESSAGE COVERED BY THE TAG

Associated data · public

format_version = 2 · service = inventory ·
principal_id = svc-17

Nonce · public

8xJ2...

Ciphertext · public bytes

5f2a... · contents = secret token

Public tag (illustrative)

tag = "b79e..." · beside covered
message

Change a visible field

service: inventory → billing

REJECT

The same tag no longer verifies

Associated data lets the consumer inspect routing metadata
without accepting a changed meaning.

An approval is an authenticated record with freshness

Digest the exact intent a human saw; bind approver, expiry, and a nonce to that digest

The intent: the object a human approves

```
{  
  "operation": "publish_makeup_quiz",  
  "course_id": 1451193,  
  "quiz_id": 9912,  
  "student_ids": [44301],  
  "opens_at": "2026-10-14T17:40:00-05:00",  
  "closes_at": "2026-10-14T18:30:00-05:00",  
  "questions_sha256": "b21...",  
  "preconditions": {"attempts": 0, "published": false}  
}
```

An approval is an authenticated record with freshness

The approval: bound to the digest, and it expires

```
approval = {  
  "intent_sha256": digest(intent),  
  "approver": authenticated_user,  
  "issued_at": now,  
  "expires_at": now + minutes(15),  
  "nonce": token_hex(16),  
}  
require(now < approval["expires_at"])  
require(digest(current_intent)  
        == approval["intent_sha256"])  
require(nonce_used_once(approval["nonce"]))
```

- “Approve deployment” binds nothing. The record names targets, values, timing, content hash, and preconditions; the approval dies on expiry, drift, or reuse.

Key possession remains a trust boundary

OUTSIDE KEY BOUNDARY

File attacker

Attacker

Reads, replaces, or replays the file; no key

🛡️ Encrypted record

Read: ciphertext

Replace:
verification fails

🔄 Old valid record

May replay
without freshness

KEY + PLAINTEXT

Trusted keyholder + consumer



Consumer + plaintext token

Secret key decrypts and creates valid
records

📦 Inventory API

Intended
token use



Build log copy

Bearer token
persists
outside workflow

Key and plaintext holders remain trusted; the boundary is an assumption, not OS isolation.

Key Takeaways

1. A bearer token lets its holder act as a principal. Protect the stored record against disclosure and undetected replacement, then limit the token's authority at the service.
2. Use authenticated encryption for secret bytes and bind visible fields as associated data. Reject verification failure before using plaintext, and add freshness checks when replay matters.
3. State the threat model. Cryptography protects the record from parties without the key; keyholders and plaintext consumers remain inside the trust boundary.

Design an encrypted service-token record

10 min

- A build service stores an encrypted API token for a deployment consumer. A file attacker can read, replace, and replay stored bytes but does not know the secret key.
- List the record fields. Mark each one public or secret, and state which fields must be covered by authenticated integrity.
- Explain what the consumer must do before using plaintext, including any freshness state needed to reject an old valid record.
- Explain what the record's cryptography cannot prevent if the consumer discloses its key or plaintext token.