Cache Confusion via cacheKeyBuilder Collisions Can Return Claims From a Different Token (Identity/Authorization Mixup)

Critical **antoatta85** published **GHSA-rp9m-7r4c-75qg** last week

Package

fast-jwt (npm)

Affected versions

>= 0.0.1 < 6.1.0

Patched versions

6.2.0

Description

NOTE: While the library exposes a mechanism which could introduce the vulnerability, this issue is created by developer-supplied code and not by the library itself. We will add a warning and some education for users around the possible issues however since the defaults work we will not be updating the library beyond that for this advisory.

Impact

Setting up a custom cacheKeyBuilder method which does not properly create unique keys for different tokens can lead to cache collisions. This could cause tokens to be mis-identified during the verification process leading to:

- Valid tokens returning claims from different valid tokens
- Users being mis-identified as other users based on the wrong token

This could result in:

- User impersonation - UserB receives UserA's identity and permissions
- Privilege escalation - Low-privilege users inherit admin-level access
- Cross-tenant data access - Users gain access to other tenants' resources
- Authorization bypass - Security decisions made on wrong user identity

Affected Configurations

This vulnerability ONLY affects applications that BOTH:

1. Enable caching using the cache option
2. Use custom cacheKeyBuilder functions that can produce collisions

VULNERABLE examples:

```
// Collision-prone: same audience = same cache key
cacheKeyBuilder: (token) => {
  const { aud } = parseToken(token)
  return `aud=${aud}`
}

// Collision-prone: grouping by user type
cacheKeyBuilder: (token) => {
  const { aud } = parseToken(token)
  return aud.includes('admin') ? 'admin-users' : 'regular-users'
}

// Collision-prone: tenant + service grouping
cacheKeyBuilder: (token) => {
  const { iss, aud } = parseToken(token)
  return `${iss}-${aud}`
}
```

SAFE examples:

```
// Default hash-based (recommended)
createVerifier({ cache: true })  // Uses secure default

// Include unique user identifier
cacheKeyBuilder: (token) => {
  const { sub, aud, iat } = parseToken(token)
  return `${sub}-${aud}-${iat}`
}

// No caching (always safe)
createVerifier({ cache: false })
```

Not Affected

- Applications using **default caching**
- Applications with **caching disabled**

Assessment Guide

To determine if you're affected:

1. Check if caching is enabled: Look for cache: true or cache: in verifier configuration
2. Check for custom cache key builders: Look for cacheKeyBuilder function in configuration
3. Analyze collision potential: Review if your cacheKeyBuilder can produce identical keys for different users/tokens
4. If no custom cacheKeyBuilder: You are NOT affected (default is safe)

Mitigations

Mitigations include:

- Ensure uniqueness of keys produced in cacheKeyBuilder
- Remove custom cacheKeyBuilder method
- Disable caching

---

fast-jwt allows enabling a verification cache through the cache option.
The cache key is derived from the token via cacheKeyBuilder.

When a custom cacheKeyBuilder produces collisions between different tokens, the verifier may return the cached payload of a previous token instead of validating and returning the payload of the current token.

This results in cross-token payload reuse and identity confusion.

Two distinct valid JWTs can be verified successfully but mapped to the same cached entry, causing the verifier to return claims belonging to a different token.

This affects authentication and authorization decisions when applications trust the returned payload.

Affected component

src/verifier.js

Relevant logic:

cache enabled via createCache

cache population via cacheSet

lookup based on cacheKeyBuilder(token)

cached payload returned without re-verification

Impact

Identity / authorization confusion via cache collision.

If two tokens generate the same cache key:

token A is verified → payload stored in cache

token B is verified → cache hit occurs

verifier returns payload from token A instead of B

Observed effect:

subject mismatch

claim mismatch

authorization decision performed on wrong identity

Potential real-world consequences:

user impersonation (logical)

privilege confusion

incorrect RBAC evaluation

gateway / middleware auth inconsistencies

This is especially dangerous when:

cache is enabled (recommended for performance)

custom cacheKeyBuilder is used

identity claims (sub / aud / iss) drive authorization

Root cause

The verifier assumes the cache key uniquely identifies the token and its claims.

However:

cacheKeyBuilder is user-controlled

collisions are not detected

cache entries store decoded payload

cached payload is returned without binding validation

This creates a trust boundary break between:

token → cache key → cached payload

Proof of concept

Environment:

fast-jwt: 6.1.0

Node.js: v24.13.1

PoC:

```
const { createSigner, createVerifier } = require('fast-jwt')

const sign = createSigner({ key: 'secret' })

// Two distinct tokens
const t1 = sign({ sub: 'userA', aud: 'admin' })
const t2 = sign({ sub: 'userB', aud: 'admin' })

// Deliberately unsafe cache key builder (collision)
const verify = createVerifier({
key: 'secret',
cache: true,
cacheKeyBuilder: () => 'static-key'
})

console.log('verify t1')
const p1 = verify(t1)
console.log('t1 PASS sub=', p1.sub)

console.log('verify t2')
const p2 = verify(t2)
console.log('t2 PASS sub=', p2.sub)

console.log('verify t2 again')
const p3 = verify(t2)
console.log('t2-again PASS sub=', p3.sub)

console.log('verify t1 again')
const p4 = verify(t1)
console.log('t1-again PASS sub=', p4.sub)
```

Observed output:

```
verify t1
t1 PASS sub= userA
```

verify t2
t2 PASS sub= userA

verify t2 again
t2-again PASS sub= userA

verify t1 again
t1-again PASS sub= userA

The verifier returns payload from userA when verifying userB.

Expected behavior

Cache must not allow returning claims from a different token.

Verification must remain bound to the actual token being validated.

Even if cache collisions occur, the verifier should:

revalidate signature

re-decode payload

or invalidate cache entry

Why this is not “just misuse”

This is not merely a user mistake.

Reasons:

fast-jwt explicitly exposes cacheKeyBuilder as an extension point.

The documentation suggests performance tuning via custom key builders.

No safeguards exist against collisions.

No verification binding is performed between:

cached payload

original token

The verifier trusts cache output as authoritative identity.

This creates a security-sensitive invariant:

"cache key uniqueness"

which is neither enforced nor validated.

Security-critical libraries must assume extension hooks can be misused and implement defensive checks, especially when identity decisions are derived from cached values.

Security classification

logical authorization flaw

cache confusion vulnerability

identity boundary break

Closest CWE:

CWE-440 — Expected Behavior Violation

Suggested fix (minimal and safe)

Bind cache entries to token integrity.

Option A — safest:

Store token hash along with payload and verify match before returning cache.

Conceptual patch:

```
const tokenHash = hashToken(token)

cache.set(key, { tokenHash, payload })

...

const entry = cache.get(key)

if (entry && entry.tokenHash === hashToken(token)) {
return entry.payload
}
```

Option B — simpler:

Disable cache usage when custom cacheKeyBuilder is provided.

Option C — defensive:

Always re-validate signature when cache hit occurs.

Notes

Default cacheKeyBuilder is safe (hash-based).

Issue appears when custom builders are used — a documented and supported feature.

Impact increases in:

API gateways

auth middleware

RBAC layers relying on payload.sub / payload.aud

This vulnerability is independent from:

RegExp statefulness issue

ReDoS claim validation issue

It is a separate flaw in cache design and trust model.

PoC did on my computer:

```
'use strict'

const fs = require('node:fs')
const path = require('node:path')
const { createSigner, createVerifier } = require('./src')

function nowSec() {
return Math.floor(Date.now() / 1000)
}

const sign = createSigner({ key: 'secret' })
const t1 = sign({ sub: 'userA', aud: 'admin', iat: nowSec() })
const t2 = sign({ sub: 'userB', aud: 'admin', iat: nowSec() })

function badKeyBuilder() {
return 'aud=admin'
}

const verify = createVerifier({
key: 'secret',
cache: true,
cacheTTL: 60000,
cacheKeyBuilder: badKeyBuilder
})

function run(tok) {
try {
const out = verify(tok)
return { ok: true, sub: out.sub, aud: out.aud }
} catch (e) {
return { ok: false, code: e.code || String(e), message: e.message }
}
}
```

```
const results = []
results.push({ step: 'verify(t1)', token: 't1', result: run(t1) })
results.push({ step: 'verify(t2)', token: 't2', result: run(t2) })
results.push({ step: 'verify(t2) again', token: 't2', result: run(t2) })
results.push({ step: 'verify(t1) again', token: 't1', result: run(t1) })

const evidence = {
title: 'fast-jwt cache confusion when cacheKeyBuilder collisions occur',
environment: {
node: process.version,
fastJwt: require('./package.json').version
},
config: {
cache: true,
cacheTTL: 60000,
cacheKeyBuilder: "returns constant key 'aud=admin' (realistic collision pattern)"
},
tokens: {
t1: { claims: { sub: 'userA', aud: 'admin' }, jwt: t1 },
t2: { claims: { sub: 'userB', aud: 'admin' }, jwt: t2 }
},
observed: results
}

const outPath = path.join(process.cwd(), 'evidence-cache-keybuilder-confusion.json')
fs.writeFileSync(outPath, JSON.stringify(evidence, null, 2))
console.log('Wrote evidence to:', outPath)

for (const r of results) {
console.log(r.step, '=>', r.result.ok ? `PASS sub=${r.result.sub}` : `FAIL ${r.result.code}`)
}
```

Output:

```
PS C:\Users\Franciny Rojas\Desktop\crypto-research\fast-jwt> node
poc_cache_keybuilder_confusion_evidence.js
Wrote evidence to: C:\Users\Franciny Rojas\Desktop\crypto-research\fast-jwt\evidence-cache-
keybuilder-confusion.json
verify(t1) => PASS sub=userA
verify(t2) => PASS sub=userA
verify(t2) again => PASS sub=userA
verify(t1) again => PASS sub=userA
PS C:\Users\Franciny Rojas\Desktop\crypto-research\fast-jwt>
```

Severity

Critical 9.1 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	None

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

CVE ID

CVE-2026-35039

Weaknesses

No CWEs

Credits

 **fasrm** Reporter

 **SociableSteve** Analyst