

You are a software engineer with web security expertise. How can I implement Signed Double Submit Cookie pattern to prevent cookie injection and CSRF attacks in a NestJs application?

Below is a **practical, security-oriented design** for implementing the **Signed Double Submit Cookie (SDSC) pattern** in a NestJS application. This approach mitigates **CSRF** while also preventing **cookie injection / fixation** attacks that affect naïve double-submit implementations.

1. Threat Model and Rationale

Why not plain Double Submit Cookies?

The classic pattern assumes:

- Attacker cannot read cookies (true under SOP)
- But **attacker can set cookies** via:
 - subdomain takeover
 - response splitting
 - malicious browser extensions
 - poorly scoped Domain attributes

If the CSRF token is unsigned, an attacker can:

1. Inject a cookie with an arbitrary CSRF token
2. Reflect the same value in a request header

3. Bypass CSRF protection

Signed Double Submit Cookie (SDSC)

We harden the pattern by:

- Cryptographically **signing** the CSRF token
 - Verifying integrity server-side
 - Avoiding server-side storage (still stateless)
-

2. High-Level Flow

1. Server generates:

```
ini
```

```
csrfToken = random  
signature = HMAC(csrfToken, serverSecret)
```

 Copy code

2. Server sets cookie:

```
ini
```

```
csrf = csrfToken.signature
```

 Copy code

3. Client:

- Reads cookie value
- Sends it back in a custom header (or body)

4. Server:

- Parses token + signature

- Recomputes HMAC
- Compares using constant-time equality

3. Cryptographic Requirements

Property	Choice
MAC	HMAC-SHA256
Token size	≥ 128 bits
Secret	Server-only, rotated
Comparison	timing-safe

4. NestJS Implementation

4.1 CSRF Utility (Token Generation & Verification)

```
ts

// csrf.util.ts
import * as crypto from 'crypto';

const CSRF_SECRET = process.env.CSRF_SECRET!;

export function generateCsrfToken(): string {
```

```
const token = crypto.randomBytes(32).toString('hex');
const signature = sign(token);
return `${token}.${signature}`;
}

export function verifyCsrfToken(value: string): boolean {
  const [token, signature] = value.split('.');
  if (!token || !signature) return false;

  const expected = sign(token);
  return timingSafeEqual(signature, expected);
}

function sign(token: string): string {
  return crypto
    .createHmac('sha256', CSRF_SECRET)
    .update(token)
    .digest('hex');
}

function timingSafeEqual(a: string, b: string): boolean {
  const bufA = Buffer.from(a);
  const bufB = Buffer.from(b);
  return (
    bufA.length === bufB.length &&
    crypto.timingSafeEqual(bufA, bufB)
  );
}
```

 Copy code

5. Setting the CSRF Cookie

5.1 Middleware

ts

```
// csrf-cookie.middleware.ts
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Response, Request } from 'express';
import { generateCsrfToken } from './csrf.util';

@Injectable()
export class CsrfCookieMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: () => void) {
    if (!req.cookies?.csrf) {
      const csrf = generateCsrfToken();

      res.cookie('csrf', csrf, {
        httpOnly: false,          // must be readable by JS
        secure: true,
        sameSite: 'strict',
        path: '/',
      });
    }
    next();
  }
}
```

 Copy code

5.2 Register Middleware

ts

```
app.use(cookieParser());  
app.use(new CsrfCookieMiddleware().use);
```

 Copy code

6. CSRF Validation Guard

6.1 Guard Implementation

ts

```
// csrf.guard.ts  
import {  
  CanActivate,  
  ExecutionContext,  
  ForbiddenException,  
  Injectable,  
} from '@nestjs/common';  
import { verifyCsrfToken } from './csrf.util';  
  
@Injectable()  
export class CsrfGuard implements CanActivate {  
  canActivate(ctx: ExecutionContext): boolean {  
    const req = ctx.switchToHttp().getRequest();  
  
    const cookieToken = req.cookies?.csrf;  
    const headerToken = req.headers['x-csrf-token'];
```

```
    if (!cookieToken || !headerToken) {  
      throw new ForbiddenException('Missing CSRF token');  
    }  
  
    if (cookieToken !== headerToken) {  
      throw new ForbiddenException('CSRF token mismatch');  
    }  
  
    if (!verifyCsrfToken(cookieToken)) {  
      throw new ForbiddenException('Invalid CSRF token');  
    }  
  
    return true;  
  }  
}
```

 Copy code

6.2 Apply Guard

ts

```
@UseGuards(CsrfGuard)  
@Post('/transfer')  
transferFunds() {  
  ...  
}
```

 Copy code

7. Client-Side Handling

ts

```
const csrf = document.cookie
  .split('; ')
  .find(c => c.startsWith('csrf='))
  ?.split('=')[1];

fetch('/api/transfer', {
  method: 'POST',
  headers: {
    'X-CSRF-Token': csrf,
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data),
});
```

 Copy code

8. Cookie Hardening Checklist

Attribute	Value	Rationale
Secure	true	HTTPS only
SameSite	Strict	Best CSRF protection
HttpOnly	false	Needed for JS access
Domain	not set	Prevent subdomain injection

Attribute	Value	Rationale
Path	/	Consistency

9. Rotation and Expiration (Optional but Recommended)

Enhancements:

- Embed timestamp in token
- Reject tokens older than N minutes
- Rotate `CSRF_SECRET` with overlap window

Example token:

```
pgsql
```

```
token.timestamp.signature
```

 Copy code

10. Comparison With Alternatives

Approach	Pros	Cons
Synchronizer Token	Strong	Server state
Plain Double Submit	Stateless	Cookie injection vulnerable
Signed Double Submit	Stateless + secure	Slight crypto overhead

Approach	Pros	Cons
SameSite Only	Simple	Not sufficient alone

11. When You Still Need This

Even with:

- SameSite=Strict
- Modern browsers

You still want SDSC if:

- You serve legacy browsers
- You have subdomains
- You accept cross-origin redirects
- You want defense-in-depth

Final Recommendation

For NestJS APIs with cookie-based auth:

Signed Double Submit Cookie + SameSite + HTTPS

is the strongest *state/less* CSRF defense available today.

If you want, I can:

- Adapt this to **Fastify**

- Add **timestamped** rotation
- Integrate with **JWT sessions**
- Compare against `@nestjs/csrf` or `csrf` trade-offs



Add timestamped rotation and provide an implementation solution that uses 'csrf-csrf' package. But before that, please provide a test files examples for `CsrfGuard`, `CsrfCookieMiddleware` and the utility functions in `csrf.util.ts` file

Understood. Below are **focused, production-grade Jest test examples** for the three components as **previously defined**:

- `csrf.util.ts`
- `CsrfCookieMiddleware`
- `CsrfGuard`

These tests emphasize **security correctness**, **edge cases**, and **tamper resistance**, not just happy paths.

Assumptions:

- Testing framework: **Jest**
- HTTP adapter: **Express**
- Cookies parsed via `cookie-parser`
- Environment variable `CSRF_SECRET` is defined in test setup

1. Tests for csrf.util.ts

1.1 csrf.util.spec.ts

ts

```
import * as crypto from 'crypto';
import {
  generateCsrfToken,
  verifyCsrfToken,
} from './csrf.util';

describe('csrf.util', () => {
  beforeAll(() => {
    process.env.CSRF_SECRET = 'test-secret';
  });

  describe('generateCsrfToken', () => {
    it('should generate a token with a valid format', () => {
      const token = generateCsrfToken();

      const parts = token.split('.');
      expect(parts).toHaveLength(2);
      expect(parts[0]).toMatch(/^[a-f0-9]{64}$/);
      expect(parts[1]).toMatch(/^[a-f0-9]{64}$/);
    });

    it('should generate unique tokens', () => {
      const t1 = generateCsrfToken();
```

```
    const t2 = generateCsrfToken();
    expect(t1).not.toEqual(t2);
  });
});

describe('verifyCsrfToken', () => {
  it('should validate a correctly signed token', () => {
    const token = generateCsrfToken();
    expect(verifyCsrfToken(token)).toBe(true);
  });

  it('should reject a token with a modified payload', () => {
    const token = generateCsrfToken();
    const [payload, sig] = token.split('.');
    const tampered = `deadbeef${payload.slice(8)}${sig}`;

    expect(verifyCsrfToken(tampered)).toBe(false);
  });

  it('should reject a token with a modified signature', () => {
    const token = generateCsrfToken();
    const [payload] = token.split('.');
    const badSig = crypto.randomBytes(32).toString('hex');

    expect(verifyCsrfToken(`${payload}.${badSig}`)).toBe(false);
  });

  it('should reject malformed tokens', () => {
    expect(verifyCsrfToken('')).toBe(false);
  });
});
```

```
    expect(verifyCsrfToken('abc')).toBe(false);  
    expect(verifyCsrfToken('a.b.c')).toBe(false);  
  });  
});  
});
```

 Copy code

2. Tests for CsrfCookieMiddleware

2.1 csrf-cookie.middleware.spec.ts

ts

```
import { CsrfCookieMiddleware } from './csrf-cookie.middleware';  
import { generateCsrfToken } from './csrf.util';  
  
jest.mock('./csrf.util');  
  
describe('CsrfCookieMiddleware', () => {  
  let middleware: CsrfCookieMiddleware;  
  
  beforeEach(() => {  
    middleware = new CsrfCookieMiddleware();  
    (generateCsrfToken as jest.Mock).mockReturnValue('token.signature');  
  });  
  
  it('should set csrf cookie if not present', () => {  
    const req: any = { cookies: {} };  
    const res: any = {
```

```
      cookie: jest.fn(),
    };
    const next = jest.fn();

    middleware.use(req, res, next);

    expect(res.cookie).toHaveBeenCalledWith(
      'csrf',
      'token.signature',
      expect.objectContaining({
        httpOnly: false,
        secure: true,
        sameSite: 'strict',
        path: '/',
      }),
    );

    expect(next).toHaveBeenCalled();
  });

  it('should not overwrite existing csrf cookie', () => {
    const req: any = {
      cookies: { csrf: 'existing.token' },
    };
    const res: any = {
      cookie: jest.fn(),
    };
    const next = jest.fn();
```

```
middleware.use(req, res, next);

expect(res.cookie).not.toHaveBeenCalled();
expect(next).toHaveBeenCalled();
});
});
```

 Copy code

Security-relevant behavior verified:

- No regeneration if cookie already exists (prevents fixation via churn)
- Cookie attributes are enforced

3. Tests for Csrfguard

3.1 csrf.guard.spec.ts

ts

```
import { Csrfguard } from './csrf.guard';
import { verifyCsrftoken } from './csrf.util';
import { ForbiddenException } from '@nestjs/common';

jest.mock('./csrf.util');

describe('Csrfguard', () => {
  let guard: Csrfguard;

  beforeEach(() => {
    guard = new Csrfguard();
```



```
});

function mockExecutionContext(req: any) {
  return {
    switchToHttp: () => ({
      getRequest: () => req,
    }),
  } as any;
}

it('should allow request with valid matching tokens', () => {
  (verifyCsrfToken as jest.Mock).mockReturnValue(true);

  const req = {
    cookies: { csrf: 'token.signature' },
    headers: { 'x-csrf-token': 'token.signature' },
  };

  const ctx = mockExecutionContext(req);
  expect(guard.canActivate(ctx)).toBe(true);
});

it('should reject if cookie token is missing', () => {
  const req = {
    cookies: {},
    headers: { 'x-csrf-token': 'token.signature' },
  };

  const ctx = mockExecutionContext(req);
```

```
expect(() => guard.canActivate(ctx)).toThrow(ForbiddenException);
});

it('should reject if header token is missing', () => {
  const req = {
    cookies: { csrf: 'token.signature' },
    headers: {},
  };

  const ctx = mockExecutionContext(req);
  expect(() => guard.canActivate(ctx)).toThrow(ForbiddenException);
});

it('should reject if tokens do not match', () => {
  const req = {
    cookies: { csrf: 'token.signature' },
    headers: { 'x-csrf-token': 'other.token' },
  };

  const ctx = mockExecutionContext(req);
  expect(() => guard.canActivate(ctx)).toThrow(ForbiddenException);
});

it('should reject if signature verification fails', () => {
  (verifyCsrfToken as jest.Mock).mockReturnValue(false);

  const req = {
    cookies: { csrf: 'token.signature' },
    headers: { 'x-csrf-token': 'token.signature' },
  };

```

```
};  
  
    const ctx = mockExecutionContext(req);  
    expect(() => guard.canActivate(ctx)).toThrow(ForbiddenException);  
  });  
});
```

 Copy code

Security guarantees verified:

- **Presence** enforcement
- **Equality** check (double-submit)
- **Cryptographic integrity** enforcement
- No reliance on client trust

What Comes Next

In the next step, I will:

1. **Add timestamped rotation** (token expiry + leeway window)
2. Replace the custom implementation with `csrf-csrf`
3. Show:
 - Secure configuration
 - NestJS middleware + guard integration
 - How `csrf-csrf` internally solves cookie injection
 - Migration considerations from your current code

Proceed when ready.



proceed

Below is a **complete, production-grade solution** that:

1. Adds **timestamped rotation** (expiration + leeway)
2. Replaces the custom crypto with the `csrf-csrf` **package**
3. Preserves the **Signed Double Submit Cookie** security properties
4. Integrates cleanly into **NestJS (Express)**

I will proceed in a **strictly layered** manner so the security guarantees are explicit.

1. Why `csrf-csrf` Is the Correct Tool

`csrf-csrf` implements a hardened **Signed Double Submit Cookie** pattern with:

- HMAC-signed tokens
- Timestamp embedding
- Automatic expiration
- Stateless verification
- Built-in cookie injection resistance

Internally, its token structure is conceptually:

SCSS

```
base64(random) . base64(timestamp) . HMAC(secret, random + timestamp)
```

 Copy code

This is exactly what we want—without re-implementing cryptography.

2. Security Properties We Achieve

Threat	Mitigated
CSRF	Yes
Cookie injection	Yes (signature)
Token replay	Yes (expiry)
Subdomain cookie poisoning	Yes (no Domain attr)
Timing attacks	Yes
Server state	None

3. Installation

```
bash

npm install csrf-csrf
```

 Copy code

4. Central CSRF Configuration (Timestamped Rotation)

4.1 csrf.config.ts

ts

```
import { doubleCsrf } from 'csrf-csrf';

export const csrf = doubleCsrf({
  getSecret: () => process.env.CSRF_SECRET!,
  cookieName: 'csrf',
  cookieOptions: {
    httpOnly: false,
    secure: true,
    sameSite: 'strict',
    path: '/',
  },

  size: 32, // 256-bit random value
  getTokenFromRequest: (req) =>
    req.headers['x-csrf-token'],

  ignoredMethods: ['GET', 'HEAD', 'OPTIONS'],

  // Timestamped rotation
  expiresIn: 60 * 15, // 15 minutes
});
```

 Copy code

Key Security Notes

- `expiresIn` enforces token freshness
- Expired tokens are **cryptographically invalid**

- No server-side token storage
 - HMAC secret rotation is supported by process restart
-

5. CSRF Cookie Middleware (NestJS)

`csrf-csrf` exposes a middleware-compatible function.

5.1 `csrf-cookie.middleware.ts`

ts

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { csrf } from './csrf.config';

@Injectable()
export class CsrfCookieMiddleware implements NestMiddleware {
  use(req: any, res: any, next: () => void) {
    // Generates cookie if missing or expired
    csrf.generateToken(req, res);
    next();
  }
}
```

 Copy code

Important Detail

- Token is **rotated automatically** if expired
- Cookie regeneration is **idempotent**
- No fixation risk (signature changes)

6. CSRF Guard Using csrf-csrf

6.1 csrf.guard.ts

ts

```
import {
  CanActivate,
  ExecutionContext,
  ForbiddenException,
  Injectable,
} from '@nestjs/common';
import { csrf } from './csrf.config';

@Injectable()
export class CsrfGuard implements CanActivate {
  canActivate(ctx: ExecutionContext): boolean {
    const req = ctx.switchToHttp().getRequest();
    const res = ctx.switchToHttp().getResponse();

    try {
      csrf.validateRequest(req, res);
      return true;
    } catch {
      throw new ForbiddenException('Invalid CSRF token');
    }
  }
}
```

 Copy code

What validateRequest() Enforces

- Cookie exists
 - Header exists
 - Values match (double submit)
 - Signature is valid
 - Timestamp is within expiresIn
-

7. Application Bootstrap

7.1 main.ts

ts

```
import cookieParser from 'cookie-parser';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(cookieParser());
  app.use(new CsrfsCookieMiddleware().use);

  await app.listen(3000);
}
```

 Copy code

8. Applying the Guard

Option A: Per-route

ts

```
@UseGuards(CsrfGuard)
@Post('/transfer')
transferFunds() {
  ...
}
```

 Copy code

Option B: Global (Recommended)

ts

```
app.useGlobalGuards(new CsrfGuard());
```

 Copy code

With `ignoredMethods`, safe verbs remain unaffected.

9. Client-Side Usage (Unchanged)

ts

```
const csrf = document.cookie
  .split('; ')
  .find(c => c.startsWith('csrf='))
  ?.split('=')[1];

fetch('/api/transfer', {
  method: 'POST',
```

```
headers: {
  'X-CSRF-Token': csrf,
  'Content-Type': 'application/json',
},
body: JSON.stringify(data),
});
```

 Copy code

10. Timestamped Rotation Behavior (Explicit)

Scenario	Result
Token < 15 min old	Accepted
Token expired	Rejected
Expired cookie present	Regenerated
Replay attack	Rejected
Injected cookie	Signature invalid
Header-only attack	Rejected

11. Secret Rotation Strategy

Recommended:

env

```
CSRF_SECRET=current-secret
```

 Copy code

Rotation:

- Deploy with new secret
- Old tokens naturally expire
- No coordination required

If you need overlapping secrets, I can show a multi-secret verifier.

12. Comparison With Your Previous Custom Code

Aspect	Custom	csrf-csrf
HMAC signing	Yes	Yes
Timestamped expiry	Manual	Built-in
Middleware support	Manual	Native
Maintenance risk	Medium	Low
Audited library	No	Yes

Final Recommendation

For NestJS applications using cookie-based authentication:

`csrf-csrf` with timestamped rotation is the best stateless CSRF defense available today.

If you want next steps, I can:

- Add **Fastify** support
- Show **JWT + CSRF** coexistence
- Provide **E2E tests**
- Design **multi-secret rotation**
- Analyze **SameSite edge cases** (OAuth, redirects)



Provide unit tests for "csrf.guard.ts" and "csrf-cookie.middleware.ts", and then provide e2e tests.

Below is a **complete, security-focused test suite** covering:

1. Unit tests

- `csrf.guard.ts`
- `csrf-cookie.middleware.ts`

2. End-to-end (E2E) tests

- Full NestJS app
- Real `csrf-csrf` behavior
- Cookie + header validation
- Expiration behavior

The tests are structured to validate **actual CSRF guarantees**, not just NestJS wiring.

1. Unit Tests

Unit tests **mock** `csrf-csrf` to:

- Isolate NestJS behavior
 - Ensure correct control flow
 - Verify exception handling
-

1.1 Unit Tests — `csrf.guard.spec.ts`

ts

```
import { Csrfguard } from './csrf.guard';
import { ForbiddenException } from '@nestjs/common';
import { csrf } from './csrf.config';
```

```
jest.mock('./csrf.config', () => ({
  csrf: {
    validateRequest: jest.fn(),
  },
}));
```

```
describe('Csrfguard', () => {
  let guard: Csrfguard;
```

```
beforeEach(() => {
  guard = new CsrfGuard();
  jest.clearAllMocks();
});

function mockContext(req = {}, res = {}) {
  return {
    switchToHttp: () => ({
      getRequest: () => req,
      getResponse: () => res,
    }),
  } as any;
}

it('should allow request when validateRequest succeeds', () => {
  (csrf.validateRequest as jest.Mock).mockImplementation(() => {});

  const ctx = mockContext();
  expect(guard.canActivate(ctx)).toBe(true);
  expect(csrf.validateRequest).toHaveBeenCalled();
});

it('should throw ForbiddenException when validateRequest throws', () => {
  (csrf.validateRequest as jest.Mock).mockImplementation(() => {
    throw new Error('CSRF invalid');
  });

  const ctx = mockContext();
```

```
    expect(() => guard.canActivate(ctx)).toThrow(ForbiddenException);  
    expect(csrf.validateRequest).toHaveBeenCalled();  
  });  
});
```

 Copy code

What this verifies

- Guard delegates all validation to `csrf-csrf`
 - Any failure results in **HTTP 403**
 - No false positives
-

1.2 Unit Tests — `csrf-cookie.middleware.spec.ts`

ts

```
import { CsrfCookieMiddleware } from './csrf-cookie.middleware';  
import { csrf } from './csrf.config';  
  
jest.mock('./csrf.config', () => ({  
  csrf: {  
    generateToken: jest.fn(),  
  },  
}));  
  
describe('CsrfCookieMiddleware', () => {  
  let middleware: CsrfCookieMiddleware;  
  
  beforeEach(() => {
```



```
middleware = new CsrfCookieMiddleware();
jest.clearAllMocks();
});

it('should call generateToken and continue', () => {
  const req = {};
  const res = {};
  const next = jest.fn();

  middleware.use(req as any, res as any, next);

  expect(csrf.generateToken).toHaveBeenCalledWith(req, res);
  expect(next).toHaveBeenCalled();
});

it('should still call next if generateToken throws', () => {
  (csrf.generateToken as jest.Mock).mockImplementation(() => {
    throw new Error('unexpected');
  });

  const req = {};
  const res = {};
  const next = jest.fn();

  expect(() =>
    middleware.use(req as any, res as any, next),
  ).toThrow();

  // If you prefer resilience, you can catch and log instead
```

```
    expect(next).not.toHaveBeenCalled();  
  });  
});
```

 Copy code

Design note

Failing hard on CSRF token generation is **acceptable**:

- App cannot safely proceed
 - Better than silently disabling CSRF
-

2. End-to-End (E2E) Tests

These tests:

- Use **real** `csrf-csrf`
 - Use **Supertest**
 - Validate **cookie + header + expiry**
 - Run against a real NestJS HTTP server
-

2.1 E2E Test App

`test/app.e2e-spec.ts`

`ts`

```
import { Test } from '@nestjs/testing';  
import {
```

```
    Controller,
    Get,
    Post,
    UseGuards,
  } from '@nestjs/common';
import { INestApplication } from '@nestjs/common';
import * as request from 'supertest';
import cookieParser from 'cookie-parser';

import { Csrfguard } from '../src/csrfguard';
import { CsrfguardMiddleware } from '../src/csrfguard-middleware';

@Controller()
class TestController {
  @Get('/public')
  getPublic() {
    return { ok: true };
  }

  @UseGuards(Csrfguard)
  @Post('/protected')
  postProtected() {
    return { success: true };
  }
}

describe('CSRF E2E', () => {
  let app: INestApplication;
```

```
beforeAll(async () => {
  process.env.CSRF_SECRET = 'e2e-secret';

  const moduleRef = await Test.createTestingModule({
    controllers: [TestController],
  }).compile();

  app = moduleRef.createNestApplication();
  app.use(cookieParser());
  app.use(new CsrfCookieMiddleware().use);

  await app.init();
});

afterAll(async () => {
  await app.close();
});

it('should set csrf cookie on GET', async () => {
  const res = await request(app.getHttpServer())
    .get('/public')
    .expect(200);

  const cookies = res.headers['set-cookie'];
  expect(cookies).toBeDefined();
  expect(cookies.some(c => c.startsWith('csrf='))).toBe(true);
});

it('should reject POST without CSRF header', async () => {
```

```
const agent = request.agent(app.getHttpServer());

await agent.get('/public');

await agent
  .post('/protected')
  .expect(403);
});

it('should allow POST with valid csrf cookie + header', async () => {
  const agent = request.agent(app.getHttpServer());

  const getRes = await agent.get('/public');

  const csrfCookie = getRes.headers['set-cookie']
    .find(c => c.startsWith('csrf='))
    .split(';')[0]
    .split('=')[1];

  await agent
    .post('/protected')
    .set('X-CSRF-Token', csrfCookie)
    .expect(201)
    .expect({ success: true });
});

it('should reject POST with mismatched csrf header', async () => {
  const agent = request.agent(app.getHttpServer());
```

```
    await agent.get('/public');

    await agent
      .post('/protected')
      .set('X-CSRF-Token', 'invalid.token')
      .expect(403);
  });
});
```

 Copy code

2.2 Expiration / Rotation E2E Test

To test expiration deterministically, configure a **short TTL**:

`csrf.config.ts` (test override)

ts

```
expiresIn: 1, // 1 second (TEST ONLY)
```

 Copy code

E2E Test

ts

```
it('should reject expired csrf token', async () => {
  const agent = request.agent(app.getHttpServer());

  const res = await agent.get('/public');
```

```
const csrfCookie = res.headers['set-cookie']
  .find(c => c.startsWith('csrf='))
  .split(';')[0]
  .split('=')[1];

await new Promise(r => setTimeout(r, 1100));

await agent
  .post('/protected')
  .set('X-CSRF-Token', csrfCookie)
  .expect(403);
});
```

Copy code

ChatGPT

Get Plus

Share

3. Security Coverage Matrix

Scenario	Unit	E2E
Missing cookie	—	Yes
Missing header	—	Yes
Token mismatch	—	Yes
Invalid signature	Unit (mock)	Yes
Token expiration	—	Yes
Cookie generation	Unit	Yes

4. Final Notes

- Unit tests **mock crypto** to isolate NestJS behavior
- E2E tests validate **real cryptographic enforcement**
- This setup protects against:
 - CSRF
 - Cookie injection
 - Token replay
 - Fixation

If you want next-level coverage, I can:

- Add **Fastify E2E**
- Add **multi-secret rotation tests**
- Add **SameSite downgrade tests**
- Add **OAuth redirect edge cases**

Just say how far you want to take it.



Thank you, you've been of great help

You are welcome. I am glad the material was useful and that it addressed your requirements with the level of rigor you were looking for.

If you revisit this area in the future—whether to harden the setup further (multi-secret rotation, Fastify, OAuth edge cases), review it from an audit perspective, or integrate it into a broader security architecture—feel free to reach out.

