auth0 / **nextjs-auth0** Public

Code | Issues 8 | Pull requests 20 | Discussions | Actions | Security and

Commit 98c36dc

nandan-bhat authored 9 hours ago · 9 / 10 · Verified

```
fix: DPoP nonce retry race issue (#2580)
```

main (#2580) · v4.18.0

1 parent b06d30f commit 98c36dc

6 files changed +243 -235 lines changed

Top

Filter files…

- src/server
  - auth-client-provider.test.ts
  - auth-client-provider.ts
  - auth-client.test.ts
  - auth-client.ts
  - mcd.integration.test.ts
  - proxy-handler.test.ts

6 files changed +243 -235 lines changed

Search within code

src/server/auth-client-provider.test.ts

```
@@ -503,105 +503,6 @@ describe("AuthClientProvider", () => {
503 503       });
504 504     });
505 505
506     -   describe("proxy fetchers", () => {
507     -     it("should cache proxy fetcher by key", async () => {
508     -       const provider = new AuthClientProvider({
```

```
-         domain: "example.auth0.com",
-         createAuthClient: createAuthClientMock
-       });
-
-       const fetcher1 = { mock: "fetcher1" };
-       const fetcher2 = { mock: "fetcher2" };
-
-       const factory1 = vi.fn().mockResolvedValue(fetcher1);
-       const factory2 = vi.fn().mockResolvedValue(fetcher2);
-
-       const result1a = await provider.getProxyFetcher("key1", factory1);
-       const result1b = await provider.getProxyFetcher("key1", factory1);
-       const result2 = await provider.getProxyFetcher("key2", factory2);
-
-       expect(result1a).toBe(fetcher1);
-       expect(result1b).toBe(fetcher1); // Same fetcher from cache
-       expect(result1a).toBe(result1b); // From cache
-       expect(result2).toBe(fetcher2);
-       expect(factory1).toHaveBeenCalledTimes(1); // Only called once
-       expect(factory2).toHaveBeenCalledTimes(1);
-     });
-
-     it("should not call factory if fetcher is cached", async () => {
-       const provider = new AuthClientProvider({
-         domain: "example.auth0.com",
-         createAuthClient: createAuthClientMock
-       });
-
-       const factory = vi.fn().mockResolvedValue({ mock: "fetcher" });
-
-       await provider.getProxyFetcher("key", factory);
-       await provider.getProxyFetcher("key", factory);
-       await provider.getProxyFetcher("key", factory);
-
-       expect(factory).toHaveBeenCalledTimes(1);
-     });
-
-     it("should use LRU behavior for proxy fetchers", async () => {
-       const provider = new AuthClientProvider({
-         domain: "example.auth0.com",
```

```
549 -         createAuthClient: createAuthClientMock
550 -       });
551 -
552 -       // Fill cache to near capacity (MAX_PROXY_FETCHERS is 100)
553 -       const factories: { [key: string]: any } = {};
554 -       const fetchers: { [key: string]: any } = {};
555 -
556 -       // Create 99 fetchers to fill most of the cache
557 -       for (let i = 0; i < 99; i++) {
558 -         const key = `key${i}`;
559 -         const factory = vi.fn().mockResolvedValue({ id: `fetcher-${i}` });
560 -         factories[key] = factory;
561 -         fetchers[key] = await provider.getProxyFetcher(key, factory);
562 -       }
563 -
564 -       // Now we have 99 cached fetchers
565 -       // Add fetcher A (will be candidate for eviction if we exceed limit)
566 -       const factoryA = vi.fn().mockResolvedValue({ id: "fetcher-a" });
567 -       const fetcherA = await provider.getProxyFetcher("keyA", factoryA);
568 -       expect(fetcherA.id).toBe("fetcher-a");
569 -       expect(factoryA).toHaveBeenCalledTimes(1);
570 -
571 -       // We now have 100 cached fetchers (at max)
572 -
573 -       // Access A again - should promote it to end
574 -       const fetcherAAgain = await provider.getProxyFetcher("keyA", factoryA);
575 -       expect(fetcherAAgain).toBe(fetcherA);
576 -       expect(factoryA).toHaveBeenCalledTimes(1); // Still 1 call
577 -
578 -       // Add 2 more fetchers to trigger eviction
579 -       // This should evict the oldest entries (key0 and key1)
580 -       const factoryB = vi.fn().mockResolvedValue({ id: "fetcher-b" });
581 -       const factoryC = vi.fn().mockResolvedValue({ id: "fetcher-c" });
582 -       await provider.getProxyFetcher("keyB", factoryB);
583 -       await provider.getProxyFetcher("keyC", factoryC);
584 -
585 -       // Now key0 should be evicted (it was the oldest)
586 -       const factory0New = vi.fn().mockResolvedValue({ id: "fetcher-0-new" });
587 -       const _fetcher0Again = await provider.getProxyFetcher(
588 -         "key0",
```

```
589     -         factory0New
590     -       );
591     -       // key0 should have been evicted and factory should be called
592     -       expect(factory0New).toHaveBeenCalledTimes(1);
593     -
594     -       // A should still be cached (it was promoted)
595     -       const factoryANew = vi.fn().mockResolvedValue({ id: "fetcher-a-new" });
596     -       const fetcherAAgain2 = await provider.getProxyFetcher(
597     -         "keyA",
598     -         factoryANew
599     -       );
600     -       expect(fetcherAAgain2).toBe(fetcherA); // Same instance
601     -       expect(factoryANew).not.toHaveBeenCalled(); // Factory not called - was
        cached
602     -     });
603     -   });
604     -
605 506     describe("mode detection", () => {
606 507       it("should detect static mode", () => {
607 508         const provider = new AuthClientProvider({
```

src/server/auth-client-provider.ts

```
        @@ -34,13 +34,6 @@ interface AuthClientProviderOptions {
34  34     */
35  35    const MAX_DOMAIN_CLIENTS = 100;
36  36
37      - /**
38      -  * Maximum number of proxy fetchers to cache.
39      -  * This prevents unbounded memory growth when many unique audience/domain
40      -  * combinations are used. Uses LRU eviction matching the domain clients
        pattern.
41      -  */
42      - const MAX_PROXY_FETCHERS = 100;
43      -
44  37    /**
45  38     * AuthClientProvider manages creating and caching AuthClient instances for MCD
        mode.
46  39     *
        @@ -61,7 +54,6 @@ export class AuthClientProvider {
```

```
61  54      private domainClients: LruMap<string, AuthClient>;
62  55
63  56      private createAuthClientFactory: (domain: string) => AuthClient;
64      -   private proxyFetchers: LruMap<string, any>;
65  57
66  58      /**
67  59       * Creates a new AuthClientProvider instance.
        @@ -76,7 +68,6 @@ export class AuthClientProvider {
76  68
77  69        // Initialize LRU caches
78  70        this.domainClients = new LruMap(MAX_DOMAIN_CLIENTS);
79      -     this.proxyFetchers = new LruMap(MAX_PROXY_FETCHERS);
80  71
81  72        // Detect mode and validate configuration
82  73        if (typeof options.domain === "string") {
        @@ -192,33 +183,6 @@ export class AuthClientProvider {
192 183       return newClient;
193 184     }
194 185
195     -   /**
196     -    * Gets a proxy fetcher from cache or creates one via the provided factory.
197     -    *
198     -    * Proxy fetchers are cached per key to avoid creating multiple instances
199     -    * for the same audience or configuration.
200     -    *
201     -    * @param key - A unique key for the fetcher (e.g., "domain:audience")
202     -    * @param factory - Factory function to create the fetcher if not cached
203     -    * @returns The cached or newly created fetcher
204     -    *
205     -    * @internal
206     -    */
207     -   async getProxyFetcher(
208     -     key: string,
209     -     factory: () => Promise<any>
210     -   ): Promise<any> {
211     -     // Check cache first (LruMap.get() handles promotion)
212     -     const fetcher = this.proxyFetchers.get(key);
213     -     if (fetcher) {
214     -       return fetcher;
```

```
215        -     }
216        -     // Create new fetcher (LruMap.set() handles eviction)
217        -     const newFetcher = await factory();
218        -     this.proxyFetchers.set(key, newFetcher);
219        -     return newFetcher;
220        -   }
221        -
222   186      /**
223   187       * Resolves the domain from request headers using the resolver function.
224   188       *
```

src/server/auth-client.test.ts

```
           @@ -10039,6 +10039,66 @@
           ykwV8CV22wKDubrDje1vchfTL/ygX6p27RKpJm8eAH7k3EwVeg3NDfNVzQ==
10039  10039          expect(authClient["clientMetadata"][oauth.clockSkew]).toBe(15);
10040  10040          expect(authClient["clientMetadata"][oauth.clockTolerance]).toBe(30);
10041  10041        });
       10042  +
       10043  +     it("should ignore a provided dpopHandle when DPoP is disabled on the
              client", async () => {
       10044  +       const secret = await generateSecret(32);
       10045  +       const transactionStore = new TransactionStore({ secret });
       10046  +       const sessionStore = new StatelessSessionStore({ secret });
       10047  +       const dpopHandle = { privateKey: "test", publicKey: "test" } as any;
       10048  +
       10049  +       const authClient = new AuthClient({
       10050  +         transactionStore,
       10051  +         sessionStore,
       10052  +         domain: DEFAULT.domain,
       10053  +         clientId: DEFAULT.clientId,
       10054  +         clientSecret: DEFAULT.clientSecret,
       10055  +         secret,
       10056  +         appBaseUrl: DEFAULT.appBaseUrl,
       10057  +         routes: getDefaultRoutes(),
       10058  +         useDPoP: false,
       10059  +         fetch: getMockAuthorizationServer()
       10060  +       });
       10061  +
       10062  +       const fetcher = await authClient.fetcherFactory({
```

```
10063 +         getAccessToken: vi.fn().mockResolvedValue("at_123"),
10064 +         dpopHandle
10065 +       });
10066 +
10067 +       expect((fetcher as any).config.dpopHandle).toBeUndefined();
10068 +       expect((fetcher as any).hooks.isDpopEnabled()).toBe(false);
10069 +     });
10070 +
10071 +     it("should ignore a provided dpopHandle when DPoP is disabled for the
      fetcher", async () => {
10072 +       const secret = await generateSecret(32);
10073 +       const transactionStore = new TransactionStore({ secret });
10074 +       const sessionStore = new StatelessSessionStore({ secret });
10075 +       const { generateDpopKeyPair } = await
      import("../utils/dpopRetry.js");
10076 +       const dpopKeyPair = await generateDpopKeyPair();
10077 +       const dpopHandle = { privateKey: "test", publicKey: "test" } as any;
10078 +
10079 +       const authClient = new AuthClient({
10080 +         transactionStore,
10081 +         sessionStore,
10082 +         domain: DEFAULT.domain,
10083 +         clientId: DEFAULT.clientId,
10084 +         clientSecret: DEFAULT.clientSecret,
10085 +         secret,
10086 +         appBaseUrl: DEFAULT.appBaseUrl,
10087 +         routes: getDefaultRoutes(),
10088 +         useDPoP: true,
10089 +         dpopKeyPair,
10090 +         fetch: getMockAuthorizationServer()
10091 +       });
10092 +
10093 +       const fetcher = await authClient.fetcherFactory({
10094 +         getAccessToken: vi.fn().mockResolvedValue("at_123"),
10095 +         useDPoP: false,
10096 +         dpopHandle
10097 +       });
10098 +
10099 +       expect((fetcher as any).config.dpopHandle).toBeUndefined();
10100 +       expect((fetcher as any).hooks.isDpopEnabled()).toBe(false);
```

```
                10101 +     });
10042           10102       });
10043           10103     });
10044           10104
```

src/server/auth-client.ts

```
                      @@ -331,7 +331,7 @@ export class AuthClient {
331             331
332             332       private readonly mfaTokenTtl: number;
333             333
334               -   private proxyFetchers: { [audience: string]: Fetcher<Response> } = {};
                334 +   private proxyDpopHandles: { [audience: string]: oauth.DPoPHandle } = {};
335             335
336             336       /**
337             337        * Maximum allowed response body size (1 MB). Responses exceeding this
                      limit
                      @@ -2948,12 +2948,14 @@ export class AuthClient {
2948            2948          throw discoveryError;
2949            2949        }
2950            2950
                2951 +     const shouldUseDpop = this.useDPoP && (options.useDPoP ?? true);
                2952 +
2951            2953        const fetcherConfig: FetcherConfig<TOutput> = {
2952            2954          // Fetcher-scoped DPoP handle and nonce management
2953              -       dpopHandle:
2954              -         this.useDPoP && (options.useDPoP ?? true)
2955              -           ? oauth.DPoP(this.clientMetadata, this.dpopKeyPair!)
2956              -           : undefined,
                2955 +       dpopHandle: shouldUseDpop
                2956 +         ? (options.dpopHandle ??
                2957 +           oauth.DPoP(this.clientMetadata, this.dpopKeyPair!))
                2958 +         : undefined,
2957            2959          httpOptions: this.httpOptions,
2958            2960          allowInsecureRequests: this.allowInsecureRequests,
2959            2961          retryConfig: this.dpopOptions?.retry,
                      @@ -2964,7 +2966,7 @@ export class AuthClient {
2964            2966
2965            2967        const fetcherHooks: FetcherHooks = {
```

```
2966  2968          getAccessToken: options.getAccessToken,
2967        -       isDpopEnabled: () => options.useDPoP ?? this.useDPoP ?? false
      2969  +       isDpopEnabled: () => shouldUseDpop
2968  2970        };
2969  2971
2970  2972        return new Fetcher<TOutput>(fetcherConfig, fetcherHooks);
                  @@ -3508,38 +3510,24 @@ export class AuthClient {
3508  3510          return tokenSetResponse.tokenSet;
3509  3511        };
3510  3512
3511        -     // Get/create fetcher instance with domain-aware caching
3512        -     // In MCD mode, use provider.getProxyFetcher for shared caching across
                  AuthClient instances
3513        -     // In static mode, use local proxyFetchers cache
3514        -     const cacheKey = this.provider
3515        -       ? `${this.domain}:${options.audience}`
3516        -       : options.audience;
3517        -
3518        -     let fetcher: Fetcher<Response>;
3519        -     if (this.provider) {
3520        -       fetcher = await this.provider.getProxyFetcher(cacheKey, async () => {
3521        -         return this.fetcherFactory({
3522        -           useDPoP: this.useDPoP,
3523        -           fetch: this.fetch,
3524        -           getAccessToken: getAccessToken
3525        -         });
3526        -       });
3527        -     } else {
3528        -       // Static mode or no provider: use local cache
3529        -       fetcher = this.proxyFetchers[options.audience];
3530        -       if (!fetcher) {
3531        -         fetcher = await this.fetcherFactory({
3532        -           useDPoP: this.useDPoP,
3533        -           fetch: this.fetch,
3534        -           getAccessToken: getAccessToken
3535        -         });
3536        -         this.proxyFetchers[options.audience] = fetcher;
3537        -       }
3538        -     }
```

```
                3513 +     // Cache only the DPoP handle so nonce state is shared across proxied
                3514 +     // requests for the same audience on this AuthClient instance. Always
                     create
                3515 +     // a fresh request-bound fetcher so token resolution remains scoped to
                     the
                3516 +     // current session instead of being shared or mutated.
                3517 +     const dpopHandle =
                3518 +       this.useDPoP && this.dpopKeyPair
                3519 +         ? (this.proxyDpopHandles[options.audience] ??= oauth.DPoP(
                3520 +             this.clientMetadata,
                3521 +             this.dpopKeyPair
                3522 +           ))
                3523 +         : undefined;
3539            3524
3540            -     // Override getAccessToken for the current request
3541            -     // @ts-expect-error Override fetcher's getAccessToken to capture token
                     set side effects
3542            -     fetcher.getAccessToken = getAccessToken;
                3525 +     const fetcher = await this.fetcherFactory({
                3526 +       useDPoP: this.useDPoP,
                3527 +       fetch: this.fetch,
                3528 +       getAccessToken,
                3529 +       dpopHandle
                3530 +     });
3543            3531
3544            3532       try {
3545            3533         const response = await fetcher.fetchWithAuth(
                     @@ -3891,6 +3879,7 @@ type GetTokenSetResponse = {
3891            3879   export type FetcherFactoryOptions<TOutput extends Response> = {
3892            3880     useDPoP?: boolean;
3893            3881     getAccessToken: AccessTokenFactory;
                3882 +   dpopHandle?: oauth.DPoPHandle;
3894            3883   } & FetcherMinimalConfig<TOutput>;
3895            3884
3896            3885   /**
```

src/server/mcd.integration.test.ts

```
                     @@ -732,57 +732,6 @@ describe("MCD Integration Tests (Units 6-12)", () => {
```

```
732  732
733  733         expect(client.domain).toBe("example.com");
734  734       });
735       -
736       -     it("U11-9: Proxy fetcher cached by key", async () => {
737       -       const createAuthClient = vi.fn(
738       -         () =>
739       -           ({
740       -             domain: "example.com"
741       -           }) as any
742       -       );
743       -
744       -       const provider = new AuthClientProvider({
745       -         domain: "example.com",
746       -         createAuthClient
747       -       });
748       -
749       -       const factory = vi.fn(async () => ({ fetch: vi.fn() }) as any);
750       -       const fetcher1 = await provider.getProxyFetcher(
751       -         "domain1:audience1",
752       -         factory
753       -       );
754       -       const fetcher2 = await provider.getProxyFetcher(
755       -         "domain1:audience1",
756       -         factory
757       -       );
758       -
759       -       expect(fetcher1).toBe(fetcher2);
760       -       expect(factory).toHaveBeenCalledTimes(1);
761       -     });
762       -
763       -     it("U11-10: Different keys have separate fetchers", async () => {
764       -       const createAuthClient = vi.fn(
765       -         () =>
766       -           ({
767       -             domain: "example.com"
768       -           }) as any
769       -       );
770       -
771       -       const provider = new AuthClientProvider({
```

```
772        -         domain: "example.com",
773        -         createAuthClient
774        -       });
775        -
776        -       const factory1 = vi.fn(async () => ({ fetch: "fetcher1" }) as any);
777        -       const factory2 = vi.fn(async () => ({ fetch: "fetcher2" }) as any);
778        -
779        -       const fetcher1 = await provider.getProxyFetcher("key1", factory1);
780        -       const fetcher2 = await provider.getProxyFetcher("key2", factory2);
781        -
782        -       expect(fetcher1).not.toBe(fetcher2);
783        -       expect(factory1).toHaveBeenCalledTimes(1);
784        -       expect(factory2).toHaveBeenCalledTimes(1);
785        -     });
786   735       });
787   736
788   737       // ===== Unit 12 Tests: openid Scope Enforcement =====
```

src/server/proxy-handler.test.ts

```
          @@ -1481,21 +1481,21 @@ describe("Authentication Client - Custom Proxy Handler", async () => {
1481  1481        * CRITICAL TEST: Validates that tokenSetSideEffect is properly captured on each proxy call.
1482  1482        *
1483  1483        * PROBLEM:
1484       -    * - Fetchers are cached per audience to reuse DPoP handles
          1484 +    * - Proxy requests reuse a DPoP handle per audience
1485  1485        * - Each proxy call creates a new `getAccessToken` closure that captures `tokenSetSideEffect`
1486       -    * - When a fetcher is reused, if we don't override its `getAccessToken`, it uses the STALE
1487       -    *   closure from the first call, which references the OLD `tokenSetSideEffect` variable
          1486 +    * - If the same fetcher instance were reused directly, it would keep the STALE closure from
          1487 +    *   the first call, which references the OLD `tokenSetSideEffect` variable
1488  1488        * - This causes the second token refresh to update the WRONG tokenSetSideEffect variable,
```

```
1489  1489          *   leading to the session not being updated on the second call
1490  1490          *
1491  1491          * SOLUTION:
1492        -       * - Override `fetcher.getAccessToken` on every proxy call to capture fresh `tokenSetSideEffect`
1493        -       * - See auth-client.ts line ~2367: `fetcher.getAccessToken = getAccessToken;`
      1492  +       * - Create a fresh request-bound fetcher on every proxy call while reusing the cached
      1493  +       *   DPoP handle per audience
1494  1494          *
1495  1495          * This test validates that BOTH proxy calls properly update their sessions after token refresh,
1496  1496          * which would fail if the tokenSetSideEffect closure is stale.
1497  1497          */
1498        -     it("8.3 should update session on BOTH calls when fetcher is reused for same audience", async () => {
      1498  +     it("8.3 should update session on BOTH calls when DPoP state is reused for same audience", async () => {
1499  1499          const now = Math.floor(Date.now() / 1000);
1500  1500
1501  1501          // Track how many times token endpoint is called
      ⇣⇡          @@ -1580,8 +1580,8 @@ describe("Authentication Client - Custom Proxy Handler", async () => {
1580  1580            `Bearer ${refreshedTokens[0]}`
1581  1581          );
1582  1582
1583        -       // ===== SECOND REQUEST (reusing fetcher for same audience) =====
1584        -       // Key point: This will reuse the cached fetcher from the first request
      1583  +       // ===== SECOND REQUEST (reusing DPoP state for same audience) =====
      1584  +       // Key point: This will reuse the cached DPoP handle from the first request
1585  1585          // If the getAccessToken closure is stale, tokenSetSideEffect won't be updated
1586  1586
1587  1587          // Simulate passage of time - token expires again
      ⇣⇡          @@ -1611,7 +1611,7 @@ describe("Authentication Client - Custom Proxy Handler", async () => {
1611  1611          expect(response2.status).toBe(200);
1612  1612
```

```
1613  1613          // CRITICAL ASSERTION: Verify second session was ALSO updated
1614        -       // BUG SCENARIO: If tokenSetSideEffect closure is stale from the cached fetcher,
      1614  +       // BUG SCENARIO: If tokenSetSideEffect closure were stale from a reused fetcher,
1615  1615          // the second token refresh would populate the OLD tokenSetSideEffect variable
1616  1616          // from the first call, which is no longer in scope. This would cause:
1617  1617          // 1. tokenSetSideEffect to remain undefined in the second call
      @@ -1634,9 +1634,9 @@ describe("Authentication Client - Custom Proxy Handler", async () => {
1634  1634          // Verify the two session cookies are different (proving both were independently updated)
1635  1635          expect(setCookie2).not.toBe(setCookie1);
1636  1636
1637        -       // Summary: This test passes because auth-client.ts overrides fetcher.getAccessToken
1638        -       // on reuse (line ~2367). Without that override, this test would FAIL because the
1639        -       // second call's tokenSetSideEffect wouldn't be captured, preventing session updates.
      1637  +       // Summary: This test passes because the proxy path creates a fresh fetcher per request
      1638  +       // while reusing only the cached DPoP handle. Without that, the second call's
      1639  +       // tokenSetSideEffect would be stale and the session would not update.
1640  1640        });
1641  1641      });
1642  1642
      @@ -1914,6 +1914,151 @@ describe("Authentication Client - Custom Proxy Handler", async () => {
1914  1914          // All three concurrent requests should have been processed
1915  1915          expect(meCallCount).toBe(3);
1916  1916        });
      1917  +
      1918  +     it("10.4 should keep DPoP nonce retry bound to the original session", async () => {
      1919  +       const now = Math.floor(Date.now() / 1000);
      1920  +       const nonceRetryDelayMs = 200;
      1921  +       const requestBDuringRetryDelayMs = 10;
```

```
1922 +       const createProxySession = (params: {
1923 +         accessToken: string;
1924 +         sid: string;
1925 +         sub: string;
1926 +       }) =>
1927 +         createInitialSessionData({
1928 +           user: {
1929 +             sub: params.sub
1930 +           },
1931 +           internal: {
1932 +             sid: params.sid,
1933 +             createdAt: now
1934 +           },
1935 +           tokenSet: {
1936 +             accessToken: params.accessToken,
1937 +             refreshToken: DEFAULT.refreshToken,
1938 +             expiresAt: now + 3600,
1939 +             scope: "read:data",
1940 +             audience: DEFAULT.audience,
1941 +             token_type: "DPoP"
1942 +           }
1943 +         });
1944 +
1945 +       const raceClient = new AuthClient({
1946 +         domain: DEFAULT.domain,
1947 +         clientId: DEFAULT.clientId,
1948 +         clientSecret: DEFAULT.clientSecret,
1949 +         appBaseUrl: DEFAULT.appBaseUrl,
1950 +         routes: getDefaultRoutes(),
1951 +         secret,
1952 +         sessionStore: new StatelessSessionStore({ secret }),
1953 +         transactionStore: new TransactionStore({ secret }),
1954 +         useDPoP: true,
1955 +         dpopKeyPair,
1956 +         dpopOptions: {
1957 +           retry: {
1958 +             delay: nonceRetryDelayMs,
1959 +             jitter: false
1960 +           }
1961 +         },
```

```
1962 +          fetch: (url, init) =>
1963 +            fetch(url, { ...init, ...(init?.body ? { duplex: "half" } : {}) })
1964 +        });
1965 +
1966 +        const sessionA = createProxySession({
1967 +          accessToken: "tokenA",
1968 +          sid: "sid-a",
1969 +          sub: "user-a"
1970 +        });
1971 +        const sessionB = createProxySession({
1972 +          accessToken: "tokenB",
1973 +          sid: "sid-b",
1974 +          sub: "user-b"
1975 +        });
1976 +
1977 +        const cookieA = await createSessionCookie(sessionA, secret);
1978 +        const cookieB = await createSessionCookie(sessionB, secret);
1979 +
1980 +        const upstreamRequests: Array<{
1981 +          authorization: string;
1982 +          nonce?: string;
1983 +        }> = [];
1984 +        let notifyNonceChallenge = () => {};
1985 +        const nonceChallengeSeen = new Promise<void>((resolve) => {
1986 +          notifyNonceChallenge = resolve;
1987 +        });
1988 +
1989 +        server.use(
1990 +          http.get(`${DEFAULT.upstreamBaseUrl}/data`, ({ request }) => {
1991 +            upstreamRequests.push({
1992 +              authorization: request.headers.get("authorization") || "",
1993 +              nonce: extractDPoPInfo(request.headers.get("dpop")).nonce
1994 +            });
1995 +
1996 +            if (upstreamRequests.length === 1) {
1997 +              notifyNonceChallenge();
1998 +              return new Response(
1999 +                JSON.stringify({
2000 +                  error: "use_dpop_nonce",
2001 +                  error_description: "DPoP nonce is required"
```

```
2002 +              }),
2003 +              {
2004 +                status: 401,
2005 +                headers: {
2006 +                  "www-authenticate": 'DPoP error="use_dpop_nonce"',
2007 +                  "dpop-nonce": "server_nonce_123",
2008 +                  "content-type": "application/json"
2009 +                }
2010 +              }
2011 +            );
2012 +          }
2013 +
2014 +          return HttpResponse.json({ success: true });
2015 +        })
2016 +      );
2017 +
2018 +      const requestA = new NextRequest(
2019 +        new URL(`${DEFAULT.proxyPath}/data`, DEFAULT.appBaseUrl),
2020 +        {
2021 +          method: "GET",
2022 +          headers: { cookie: cookieA }
2023 +        }
2024 +      );
2025 +      const requestB = new NextRequest(
2026 +        new URL(`${DEFAULT.proxyPath}/data`, DEFAULT.appBaseUrl),
2027 +        {
2028 +          method: "GET",
2029 +          headers: { cookie: cookieB }
2030 +        }
2031 +      );
2032 +
2033 +      const responseAPromise = raceClient.handler(requestA);
2034 +      await nonceChallengeSeen;
2035 +      await new Promise((resolve) =>
2036 +        setTimeout(resolve, requestBDuringRetryDelayMs)
2037 +      );
2038 +      const responseBPromise = raceClient.handler(requestB);
2039 +
2040 +      const [responseA, responseB] = await Promise.all([
2041 +        responseAPromise,
```

```
2042 +         responseBPromise
2043 +       ]);
2044 +
2045 +       expect(responseA.status).toBe(200);
2046 +       expect(responseB.status).toBe(200);
2047 +       expect(upstreamRequests).toEqual([
2048 +         {
2049 +           authorization: "DPoP tokenA",
2050 +           nonce: undefined
2051 +         },
2052 +         {
2053 +           authorization: "DPoP tokenB",
2054 +           nonce: "server_nonce_123"
2055 +         },
2056 +         {
2057 +           authorization: "DPoP tokenA",
2058 +           nonce: "server_nonce_123"
2059 +         }
2060 +       ]);
2061 +     });
1917 2062     });
1918 2063
1919 2064     describe("Category 11: CORS Handling", () => {
```

Comments 0

Please sign in to comment.