PIGEON LABS

MediBridge Security Assessment

Pentest Report

ENGAGEMENT

2028-04-10 - 2028-04-14

TARGET

https://api.medibridge.example

https://app.medibridge.example

CONFIDENTIAL

MediBridge Security Assessment

Executive Summary

MediBridge is a multi-tenant healthcare communication and telehealth platform connecting clinical practices to EHR systems like Epic, Cerner, and secure patient portals.

Pigeon Labs assessed https://api.medibridge.example / https://app.medibridge.example This pentest confirmed 21 findings (4 high, 4 medium, 4 low, 9 informational). Findings are ordered by severity below. Cross-tenant patient message, secure consultation, and SQL-ordering issues represent the highest demonstrated risk, potentially exposing sensitive Protected Health Information (PHI). Session and billing authorization gaps expand impact when combined with token exposure. Remediation guidance is included per finding.

0	4	4	4	9
CRITICAL	HIGH	MEDIUM	LOW	INFO

Table of Contents

Penetration test findings 21

1. HIGH SQL Injection Permits the Ability to Extract or Inject into Medibridge Database
2. HIGH Unscoped Message IDs Expose Other Clinics' Patient Consultations
3. HIGH Unscoped WebSocket Consultation IDs Expose Other Patients' Telehealth Sessions
4. HIGH Logout and Password Reset Leave Prior Bearer Access Usable
5. MEDIUM Provider-accessible Stripe Portal Sessions Bypass Clinic Billing Roles
6. MEDIUM Revoked Refresh Tokens Still Establish WebSocket Sessions
7. MEDIUM Script-readable Cookies Expose Reusable Session Tokens to Page Scripts
8. MEDIUM Provider Role Has Ability to Escalate Privileges via Invitation Feature
9. LOW Webhook Receipt Logging Copies Patient Messages into Operational Logs
10. LOW Plaintext Frontend Delivery Exposes Sessions over HTTP to Network Attackers
11. LOW Public Source Maps Expose Private Frontend Implementation
12. LOW AI Triage Assistant Surfaces Full System Instructions in the Provider UI
13. INFORMATIONAL Outdated Image Decoder May Compromise the API Worker
14. INFORMATIONAL Provider OAuth Callbacks Can Attach EHR Accounts Without Permission
15. INFORMATIONAL Bearer-only Profile Updates Permit Persistent Account Takeover
16. INFORMATIONAL Older Reset Links Can Override a Completed Password Recovery
17. INFORMATIONAL Public Verification Responses Reveal Registered Email Addresses
18. INFORMATIONAL Unbounded Consult Pages Permit Oversized Database Reads
19. INFORMATIONAL Public Readiness Checks Reveal Dependencies Outside API Throttling
20. INFORMATIONAL Missing Frame Restrictions Permit Clickjacking of Signed-in Users
21. INFORMATIONAL Unscoped WebSocket Consultation IDs Leak Viewer Presence on Protected Threads

Findings

Penetration test findings

Finding: SQL Injection Permits the Ability to Extract or Inject into Medibridge Database

HIGH

DESCRIPTION

Pigeon Labs engineers discovered that any authenticated clinical staff member can place a SQL expression in the patient message list's `orderBy` parameter. The expression reaches TypeORM's ordering clause rather than a bound value. Isolated PostgreSQL tests confirmed execution and a cross-table timing channel; one user-reported production timing pair was consistent with the issue but was not independently replayed. No production data extraction was demonstrated. TypeORM maintainers have documented this injection class in TypeORM/TypeORM#12333.

VULNERABLE ENDPOINTS

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/messages?
integrationId={integrationId}&orderBy={sortField}
```

STEPS TO REPRODUCE

Sign in as any active staff member of a controlled clinic with a connected EHR integration. The clinic Secure Inbox view uses this list API; the member can obtain their own `integrationId` from the clinic integrations response. Use staging for SQL timing tests.

1. Record a baseline response time:

```
curl -sS -o /dev/null -w '%{time_total}\n' \
  -H "Authorization: Bearer $ACCESS" \
  "https://api.medibridge.example/api/clinics/$SLUG/messages?
integrationId=$INTEGRATION_ID&page=1&pageSize=1&order=DESC&orderBy=portal_created_at"
```

2. Submit a non-destructive PostgreSQL time-delay expression:

```
curl -sS -o /dev/null -w '%{time_total}\n' \
  -H "Authorization: Bearer $ACCESS" \
  "https://api.medibridge.example/api/clinics/$SLUG/messages?
integrationId=$INTEGRATION_ID&page=1&pageSize=1&order=DESC&orderBy=portal_created_at%2
C(SELECT%201%20FROM%20pg_sleep(0.25))"
```

3. A local query-builder check against the locked `typeorm@0.3.31` emitted the following SQL fragment, confirming that the expression is not quoted or parameterized:

```
ORDER BY message.portal_created_at,(SELECT 1 FROM pg_sleep(3)) DESC
```

4. The same ORDER BY expression shape, with a shorter `pg_sleep(0.25)`, was executed against a disposable three-row table in PGlite 0.5.8 (PostgreSQL 18.3). Three baseline LIMIT 1 queries took 4, 1, and 1 ms; three queries with the sleep expression took 255, 252, and 251 ms. This confirms that the injected expression executes despite the LIMIT.

5. A user-supplied production transcript for clinic-alpha integration 801 showed a baseline HTTP 200 in 0.477035 seconds and the same endpoint with `pg_sleep(1.75)` in the `orderBy` value returning HTTP 200 in 3.345768 seconds. The 2.869-second gap is consistent with injected SQL execution, but a single timing pair does not exclude variable server latency or establish how many times the expression ran. The JWT expired before an independent replay; no sleep payload was sent to production by this review.

ROOT CAUSE ANALYSIS

In `messages.controller.ts`, it accepts an unrestricted query string and forwards it to `MessagesService.listMessages`:

```
@Query() orderBy?: string,
```

The EHR adapters pass that value to their repositories. `patientPortal.repository.ts` substitutes only two known names; every other expression remains caller-controlled:

```
const sortColumn =
orderBy === 'portalCreatedAt' || orderBy === 'portal_created_at'
? 'lastActivityAt'
: orderBy;
```

The same repository calls `parentQueryBuilder.orderBy(...)` with `sortColumn` inside the SQL ordering expression; the Epic and Cerner repositories follow the same pattern. SQL parameters cannot bind identifiers or arbitrary sort expressions. The isolated query-builder output above shows the injected subquery unquoted.

BUSINESS IMPACT

Any active clinic staff member needs only their own clinic slug and connected `integrationId`, both available through normal application access; no victim record ID is required. Their `orderBy` expression executes under the application's database role, creating a plausible route to infer data from other tables and consume database resources. Production table privileges and extraction were not established. However, this code analysis and dynamic tests give high confidence that the MediBridge's database can be extracted and manipulated. High reflects the reachable SQL execution boundary and potential ePHI exposure, not a claim that the full database was dumped.

RECOMMENDED REMEDIATION

In `messages.controller.ts`, replace the free-form `orderBy`?: string with a closed enum and reject all other values as 400. In each provider repository, map those enum members to fixed, code-owned column expressions before calling `orderBy`(); never concatenate the request value. Add HTTP integration tests across Patient Portal, Epic, and Cerner for commas, parentheses, comments, whitespace, and SQL keywords, plus a positive test for each allowed sort.

Finding: Unscoped Message IDs Expose Other Clinics' Patient Consultations

HIGH

DESCRIPTION

Pigeon Labs engineers discovered an IDOR that lets a provider in one clinic read another tenant's patient message and full consultation thread by pairing a known platform message ID with an EHR integration the member controls on the same provider. The service checks the supplied clinic and `integrationId` but loads the message globally by platform ID. Cross-tenant reads were reproduced on the deployed API.

An authorized clinic-alpha list response established that internal row 2040101, external message 998877665544332211, belongs to Athenahealth integration 801 (`dr_smith`). Dr. Jones's independently authorized Athenahealth integration is 805 (`dr_jones`). The Dr. Jones token received 403 Forbidden when used against the normal clinic-alpha integrations endpoint, confirming that the user lacks clinic access. The same Dr. Jones token then requested the message ID belonging to another clinic through clinic clinic-beta with `integrationId`=805; the server returned 200 OK with the same root patient message and one attached lab result. The reciprocal test was also successful: a Dr. Jones Epic message belonging to integration 910 was requested through clinic-alpha with its independently authorized Epic integration 912, and the server returned 200 OK with the medical inquiry, internal row ID, triage rationale, and workflow state belonging to another clinic. Correct behavior in both cases was 403 or 404. No write or provider-side action was tested.

VULNERABLE ENDPOINTS

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/messages/{messageId}?integrationId={integrationId}
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/messages/{messageId}/thread?integrationId={integrationId}
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/messages/{externalMessageId}/clinical-summary?integrationId={integrationId}
```

```
POST https://api.medibridge.example/api/clinics/{clinicSlug}/messages/{externalMessageId}/triage-reply
```

```
PUT https://api.medibridge.example/api/clinics/{clinicSlug}/messages/status
```

```
PUT https://api.medibridge.example/api/clinics/{clinicSlug}/messages/bulk/status
```

STEPS TO REPRODUCE

Sign in as an active staff member of a controlled attacker clinic and use an integration on the same EHR provider as a second controlled victim clinic. The Secure Inbox API exposes the member's own `integrationId` and message IDs. Obtain one victim external message ID from the victim test account; arbitrary victim-ID enumeration was not demonstrated.

1. Read that message using only the attacker clinic and attacker integration account:

   ```
   curl -i \
     -H "Authorization: Bearer $ATTACKER_ACCESS" \
     "https://api.medibridge.example/api/clinics/$ATTACKER_SLUG/messages/$VICTIM_MESSAGE_ID?integrationId=$ATTACKER_INTEGRATION_ID"
   ```

2. In staging only, test the source-predicted local status write against disposable messages:

   ```
   curl -i -X PUT \
     -H "Authorization: Bearer $ATTACKER_ACCESS" \
     -H 'Content-Type: application/json' \
     --data "{\"messageIds\":[\"$VICTIM_MESSAGE_ID\"],\"integrationId\":$ATTACKER_INTEGRATION_ID,\"isRead\":true}" \
     "https://api.medibridge.example/api/clinics/$ATTACKER_SLUG/messages/status"
   ```

   Confirm through the victim test tenant whether `isRead` changed. The production tests confirmed disclosure, not this write. Do not test provider-side response/delete actions on real patient records.

ROOT CAUSE ANALYSIS

In `messages.service.ts`, it verifies the caller's clinic and supplied integration ID, then passes the independently supplied path message ID to the provider adapter:

```
const integration = await this.securityService.verifyFullAccessChain(
userId,
request.clinicId,
request.integrationId
);
const adapter = getAdapter(integration.provider);
const message = await adapter.getMessage(
request.messageId,
integration,
request.regenerateMessageResponse
);
```

`epicMessage.repository.ts` selects by platform ID alone:

```
return await this.ormRepository.findOne({
where: { externalMessageId },
});
```

The Cerner and Athenahealth lookups similarly omit the verified integration ID. The status adapters operate on the globally found row.

BUSINESS IMPACT

An ordinary provider needs a valid integration on the same EHR and a specific platform message ID belonging to another clinic. The member's own integration ID is visible in normal application traffic; the victim ID was obtained from the victim test account, and broad enumeration was not proven. Production reads disclosed patient inquiries, medical history, triage rationale, and internal nursing notes outside the provider's tenant. This is a severe HIPAA violation and breaches patient confidentiality. Opening a thread may also clear a local new-message counter; other status, draft, and provider-side writes remain source-predicted. High reflects confirmed cross-tenant ePHI disclosure, tempered by the need for a known ID.

RECOMMENDED REMEDIATION

In MessagesService, pass the verified integration ID into every adapter lookup. Change the Epic, Cerner, and Athenahealth repository predicates to require both `externalMessageId` and `integrationId`, including inside update/delete statements; return 404 for an ID belonging to another clinic/row.

Finding: Unscoped WebSocket Consultation IDs Expose Other Patients' Telehealth Sessions

HIGH

DESCRIPTION

Pigeon Labs engineers discovered that any authenticated clinical staff member can read another patient's full Telehealth Consultation history over the WebSocket API if they know (or obtain) that consultation's numeric database id. The attack uses the normal `consultation:request` action with the attacker's own clinic slug and connected integration; no victim JWT or clinic membership is required.

REST inbox APIs tie each consultation thread to the caller's portal and integration account. The WebSocket handler checks that the user belongs to the clinic and integration they name, but then loads the consultation by global primary key only. That gap lets a provider of one tenant request a row that belongs to another.

On the deployed alpha API, a clinic-alpha session successfully retrieved a clinic-beta thread (consultation 770011) that REST denied under clinic-alpha, including patient symptoms and demographic metadata. Cross-tenant id guessing was not tested; the proof used an id harvested from an authorized victim-tenant session. Step-by-step requests and response fields are below.

VULNERABLE ENDPOINTS

`wss://api.medibridge.example/?token={accessJwt}, action consultation:request`

STEPS TO REPRODUCE

Sign in as an active staff member of controlled clinic clinic-alpha only. Open the Telehealth Inbox in the app and copy the `accessToken` cookie value, or use any valid bearer JWT for that membership. Obtain a numeric `consultationId` from a second controlled tenant you can sign into separately (example victim thread below); cross-tenant ID enumeration was not demonstrated.

1. Confirm REST denies the thread ID belonging to another clinic under the attacker clinic:

   ```
   export API='https://api.medibridge.example'
   export ATTACKER_ACCESS='<clinic-alpha access token>'
   curl -sS -i \
     -H "Authorization: Bearer $ATTACKER_ACCESS" \
     -H 'Accept: application/json' \
     "$API/api/clinics/clinic-alpha/departments/triage/consultations/770011?limit=5"
   ```

   Observed: no consultation payload for consultation 770011 under clinic-alpha REST (404 / not found). The same `consultationId` with a clinic-beta token returns 200 with the patient thread, establishing that 770011 is a victim-row id.

2. Connect to the WebSocket with the attacker token and request the victim id using attacker clinic context:

   ```
   export ATTACKER_INTEGRATION_ID='2481'
   export VICTIM_CONSULTATION_ID='770011'
   printf '%s\n' \
     "{\"action\":\"consultation:request\",\"payload\":{\"clinicSlug\":\"clinic-alpha\",
   \"integrationId\":$ATTACKER_INTEGRATION_ID,\"consultationId\":
   $VICTIM_CONSULTATION_ID}}" \
     vert{} websocat -1 "wss://api.medibridge.example/?token=$ATTACKER_ACCESS"
   ```

   Observed: server responds with action `consultation:details` and a payload including id 770011, `patientName` Elena, `profilePicture` `ws_99/patients/8888888/profile.webp`, and messages such as inbound text "I need a refill on my prescription for Lisinopril." (message id 1354658). Attacker tenant storage elsewhere uses prefix `ws_105/`, confirming cross-tenant data in the response. A secure implementation would reject the request rather than return IDs belonging to other clinic message bodies.

ROOT CAUSE ANALYSIS

The WebSocket schema accepts independent `clinicSlug`, `integrationId`, and `consultationId` fields.

Authorization on this path does not fully bind the requested object to the verified clinic context. `request-consultation.handler.ts` verifies the clinic/integration pair but does not require the loaded consultation to belong to that integration:

```
await securityService.verifyFullAccessChain(ws.userId, clinic.id, payload.integrationId);
```

The handler calls `findConsultationById(payload.consultationId)`. `telehealth-session.repository.ts` filters only by id, and the handler's response path returns patient metadata and clinical history. REST consultation routes apply thread access checks that this WebSocket path omits.

BUSINESS IMPACT

Any active staff member of any clinic who can open a WebSocket with a valid JWT needs their own `clinicSlug`, connected `integrationId` (visible in normal app traffic), and numeric `consultationId` belonging to another clinic. The victim id in this test was harvested from an authorized session on clinic-beta; guessing adjacent ids was not tested. The confirmed response exposes highly sensitive ePHI (patient name, medical requests, platform metadata, storage keys). That is a serious HIPAA breach for MediBridge's healthcare clients and their patients. High reflects demonstrated cross-tenant Consultation read, not a source-only prediction.

RECOMMENDED REMEDIATION

In `request-consultation.handler.ts`, retain the integration entity from verifyFullAccessChain and load the consultation with both `consultationId` and verified integration.id (or equivalent verifyConsultationAccess used by REST). Return an authorization-safe 404 on mismatch. Add two-tenant WebSocket integration tests for known IDs belonging to another clinics and for REST-denied ids. Do not embed live JWTs or presigned URLs in test artifacts.

Finding: Logout and Password Reset Leave Prior Bearer Access Usable

HIGH

DESCRIPTION

Pigeon Labs engineers found that logout does not immediately invalidate the old access JWT and that password reset leaves prior access and refresh tokens usable. Production tests confirmed the logout behavior. For a separate disposable account, the user reported completing a reset; the old access token still reached a protected endpoint and the old refresh token issued a new pair.

VULNERABLE ENDPOINTS

```
POST https://api.medibridge.example/api/auth/logout
```

```
POST https://api.medibridge.example/api/auth/update-password
```

```
POST https://api.medibridge.example/api/auth/refresh
```

```
GET https://api.medibridge.example/api/users/me (representative protected route)
```

STEPS TO REPRODUCE

1. Sign in to a disposable account through App.medibridge.example and retain its own access and refresh tokens. No clinic role is needed; the test concerns account sessions. Verify the access token on the profile route:

   ```
   curl -i -H "Authorization: Bearer $OLD_ACCESS" "https://api.medibridge.example/api/users/me"
   ```

2. Log out, then repeat the protected request:

```
curl -i -X POST -H 'Content-Type: application/json' \
  --data "{\"accessToken\":\"$OLD_ACCESS\"}" "https://api.medibridge.example/api/auth/logout"
curl -i -H "Authorization: Bearer $OLD_ACCESS" "https://api.medibridge.example/api/users/me"
```

   The production test used a supplied session for user 1024, before its access-token expiry. `GET /api/users/me` returned 200 before logout; `POST /api/auth/logout` returned 204; the same access token still returned 200 from `GET /api/users/me` after logout. `POST /api/auth/refresh` with the original paired refresh token returned 401. Only HTTP status codes were retained; token values and response bodies are omitted.

3. Separately, on a disposable account, reset its password and retry both the old access and refresh tokens:

```
curl -i -X POST -H 'Content-Type: application/json' \
  --data "{\"token\":\"$RESET_TOKEN\",\"password\":\"New-Test-Password-42!\",
\"confirmPassword\":\"New-Test-Password-42!\"}" \
  "https://api.medibridge.example/api/auth/update-password"
curl -i -H "Authorization: Bearer $OLD_ACCESS" "https://api.medibridge.example/api/users/me"
curl -i -X POST -H 'Content-Type: application/json' \
  --data "{\"refreshToken\":\"$OLD_REFRESH\"}" "https://api.medibridge.example/api/auth/refresh"
```

   For the production password-reset test, a fresh disposable account's access token returned 200 from `GET /api/users/me.` The user then reported completing a password reset, before the access token's expiry, the same access token again returned 200 from `GET /api/users/me`; the original refresh token returned 200 from `POST /api/auth/refresh`, demonstrating that it could issue a new token pair after the reported reset. The new pair was discarded, and the original refresh token was consumed by rotation. The reset request/response was not directly observed, so this result depends on the user's confirmation that the reset completed. The old access token had a 30-minute lifetime and the refresh token a one-day lifetime.

ROOT CAUSE ANALYSIS

`auth.service.ts` removes the token row:

```
await this.authRepository.removeToken(token);
return token.userId;
```

HTTP bearer validation checks signature, expiry, token type, and subject but not that database row. Thus the signed access JWT survives row deletion until its own expiry.

`auth.service.ts` updates the password and marks only the reset token used. It does not remove existing sessions; refresh validation still finds the old refresh-token row.

BUSINESS IMPACT

An attacker must first possess a provider's access or refresh bearer token; no clinic role or object ID is additionally required for /api/users/me. After logout, the stolen access token retains HTTP access for its remaining lifetime while its paired refresh token is rejected. After a user-confirmed password reset, both old token classes were accepted and the old refresh token issued a new pair. The tested access and refresh

lifetimes were 30 minutes and one day, respectively; renewal beyond one rotation was not tested. High reflects the practical failure of account recovery to terminate a potentially stolen session, risking medical staff accounts and their connected clinic workspaces.

RECOMMENDED REMEDIATION

Introduce a session ID/JTI or per-user token version in token issuance and verify current session state in HTTP authentication. Make logout revoke its session and close its sockets. After password reset, revoke all of that user's access and refresh sessions in the same transaction or via a version bump. Add tests for old-token rejection after logout and reset, refresh rotation, and socket closure.

Finding: Provider-accessible Stripe Portal Sessions Bypass Clinic Billing Roles

MEDIUM

DESCRIPTION

Pigeon Labs engineers discovered that an ordinary clinical provider can obtain a Stripe Billing Portal session URL even though the application treats billing as an administrative/medical director function and hides the Billing UI from standard staff. The deployed API returned a portal URL to a standard provider. Checkout-session creation and the actions available inside the resulting portal were not tested.

VULNERABLE ENDPOINTS

```
POST https://api.medibridge.example/api/clinics/{clinicSlug}/stripe/create-checkout-session
(source-confirmed only)
```

```
POST https://api.medibridge.example/api/clinics/{clinicSlug}/stripe/create-portal-session
(session creation confirmed)
```

STEPS TO REPRODUCE

Sign in as a standard provider/nurse in a controlled clinic with a Stripe customer and a subscription record whose `endedAt` is null. The clinic slug is available in the provider's clinic list; Billing is hidden in their UI, so use the API directly. A currently active subscription status is not required by the repository lookup.

1. Request a billing portal session:

   ```
   curl -i -X POST \
     -H "Authorization: Bearer $PROVIDER_ACCESS" \
     -H 'Content-Type: application/json' \
     --data '{"returnUrl":"https://app.medibridge.test/billing-return"}' \
     "https://api.medibridge.example/api/clinics/$SLUG/stripe/create-portal-session"
   ```

2. Confirm that the response contains a portal URL. Open it only for a disposable Stripe customer and with separate authorization if the available billing controls need verification.

3. `GET /api/clinics` returned clinic-alpha with role provider, clinic subscription status `manually_managed`, and subscription status canceled. A single `POST /api/clinics/clinic-alpha/stripe/create-portal-session` with that token and {"`returnUrl`":"https://app.medibridge.example/"} returned 200 with a url field. The URL was suppressed from test output. No billing change was made. The clinic's status shows that portal-session creation does not require a currently active subscription status.

ROOT CAUSE ANALYSIS

In `subscriptions.controller.ts`, it requires only clinic membership:

```
@Post('create-portal-session')
@Security('jwt', ['clinic'])
```

The checkout route uses the same scope. The role model describes billing as a medical director/admin function.

`subscriptions.service.ts` uses the clinic's Stripe customer ID to create and return a portal session. The subscription lookup filters on `endedAt`: null, not active subscription status.

BUSINESS IMPACT

Any active provider of a clinic with a Stripe customer and non-ended subscription record can request its portal URL using a known clinic slug; no Stripe customer ID is needed. The confirmed effect is access to a billing portal session despite the provider role. Depending on Stripe's portal configuration, that bearer URL may expose invoices, payment methods, or subscription controls, but those actions were not inspected. Medium reflects the proven role bypass and potential financial-data/service consequences while avoiding an unsupported claim of a billing change.

RECOMMENDED REMEDIATION

Add a `MANAGE_BILLING` permission to `clinic-permissions.ts` for the intended `admin/medical_director` roles. In both Stripe controller methods, enforce that permission before calling the service, and mirror it in service-level checks if other callers exist. Review Stripe portal configuration, revoke test-created sessions if appropriate, and add a role matrix asserting standard providers receive 403 for both routes.

Finding: Revoked Refresh Tokens Still Establish WebSocket Sessions

MEDIUM

DESCRIPTION

Pigeon Labs engineers discovered that the WebSocket upgrade accepts a refresh JWT even after logout has revoked its HTTP refresh session. The deployed server returned `101 Switching Protocols` for such a token while `POST /api/auth/refresh` returned 401. No WebSocket action frames were sent, so data access through that connection remains untested.

VULNERABLE ENDPOINTS

`wss://api.medibridge.example/?token={jwt}` (upgrade handler also accepts other paths)

STEPS TO REPRODUCE

1. Sign in to a disposable account through App.medibridge.example and retain its own refresh token. No clinic role is needed for the handshake. Log out of that session; before the refresh JWT's signed expiry, confirm the token is rejected by HTTP:

```
curl -sS -o /dev/null -w '%{http_code}\n' \
  -X POST -H 'Content-Type: application/json' \
  --data "{\"refreshToken\":\"$REVOKED_REFRESH\"}" \
  "https://api.medibridge.example/api/auth/refresh"
```

2. Attempt a WebSocket upgrade using the same revoked token. The curl connection may remain open until --max-time after it prints the response headers:

```
curl -sS --http1.1 --max-time 3 -D - -o /dev/null \
  -H 'Connection: Upgrade' -H 'Upgrade: websocket' \
  -H 'Sec-WebSocket-Version: 13' \
  -H 'Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==' \
  "https://api.medibridge.example/?token=$REVOKED_REFRESH"
```

ROOT CAUSE ANALYSIS

In `server.ts`, it reads a JWT from the URL query.

The same handler verifies only the signature, reads sub, and upgrades without checking the `type` or server-side token row:

```
const decoded = jwt.verify(token, config.jwt.secret);
const userId = decoded.sub;
```

The WebSocket upgrade validates the JWT signature but not the token class or revocation state that HTTP auth requires. It then calls `wss.handleUpgrade` without a type or token-row check. By contrast, HTTP JWT authentication rejects a non-access type, and refresh validation checks the token table.

BUSINESS IMPACT

Anyone holding a validly signed refresh JWT for an account can open a socket as that user until the JWT's expiry, even if its refresh session was revoked at logout. The tested token's signed lifetime was one day. No clinic ID or integration ID is needed for the handshake; individual actions may impose separate authorization. User/clinic notifications and handlers become reachable at the socket layer, but we did not proceed to exploitation past the proof of concept due to exploitability shown in other findings. Medium reflects the confirmed authentication bypass with a stolen-token prerequisite and limited demonstrated impact.

RECOMMENDED REMEDIATION

Make `server.ts` use the same token-class and session validation as HTTP: require an access token, numeric subject, and active session/JTI or token version before handleUpgrade. Restrict upgrades to the intended path, close associated sockets on logout/reset, and replace reusable JWT query strings with short-lived one-time tickets or secure session cookies. Test that valid access tokens connect while refresh, revoked, malformed, and expired tokens fail with 401.

Finding: Script-readable Cookies Expose Reusable Session Tokens to Page Scripts

MEDIUM

DESCRIPTION

Pigeon Labs engineers found that the production frontend stores both access and refresh JWTs in cookies readable by `page` JavaScript. Using Chrome’s developer tools shows `accessToken` and `refreshToken` on app.medibridge.example with empty `HttpOnly`, Secure, and `SameSite` columns. The frontend also constructs a WebSocket URL containing the access JWT. The tested production refresh JWTs had one-day signed lifetimes.

VULNERABLE ENDPOINTS

```
https://app.medibridge.example/login (frontend token creation)
```

```
POST https://api.medibridge.example/api/auth/login (tokens returned for browser storage)
```

```
wss://api.medibridge.example/?token={accessJwt}
```

STEPS TO REPRODUCE

Sign in through App.medibridge.example with a disposable account; any authenticated user is sufficient, and no clinic role or victim identifier is needed. Open Chrome DevTools > Application > Cookies for app.medibridge.example.

1. In the browser console, demonstrate script access:

   ```
   document.cookie
   ```

2. Observe that `accessToken` and `refreshToken` are present. The supplied Chrome Application-tab shows both production cookies with no `HttpOnly`, Secure, or explicit `SameSite` flag; token values are omitted from this report. In browser developer tools, inspect the WebSocket handshake URL to test the separate query-string claim.

3. Review a controlled reverse-proxy/APM log to determine whether that URL is retained in the deployed environment.

ROOT CAUSE ANALYSIS

In `identity.ts`, it stores both JWTs with js-cookie and later reads them in JavaScript:

```
Cookies.set(ACCESS_TOKEN_KEY, access.token);
Cookies.set(REFRESH_TOKEN_KEY, refresh.token);
```

JavaScript cannot set `HttpOnly`; this code also provides no explicit Secure or `SameSite` attributes.

`useWebSocket.tsx` reads the access token and appends it to a reusable socket URL:

```
const token = Cookies.get('accessToken');
const socketUrl = `${import.meta.env.VITE_WEBSOCKET_URL}?token=${token}`;
```

The backend consumes this URL parameter in `server.ts`. Whether proxies or telemetry retain it was not verified.

BUSINESS IMPACT

An attacker needs script execution in app.medibridge.example, a compromised included script, or access to a logging system that stores socket URLs; none was shown in this finding. Under that condition, any signed-in user's access and refresh tokens are available without a clinic-specific ID, allowing account and connected-clinic access for the token lifetime. The missing Secure flag also makes plaintext cookie transmission possible and is addressed in the separate transport finding. Medium reflects reusable bearer exposure in the browser and the one-day tested refresh lifetime, reduced by the unproven script/logging prerequisite.

RECOMMENDED REMEDIATION

Prefer a same-origin backend-for-frontend at app.medibridge.example: keep JWTs or an opaque session record on the server and give the browser only a host-only `HttpOnly`; Secure; `SameSite`=Lax (or Strict) session cookie. Proxy API calls through that backend so application JavaScript never receives the refresh

token. Revoke the server-side session on logout and password reset, and enforce CSRF or Origin checks on cookie-authenticated state changes. This follows the browser-app architecture in RFC 10017.
If retaining direct browser-to-API bearer calls, keep the short-lived access JWT in memory only. Move the refresh JWT to a server-issued, host-only `HttpOnly`; Secure; `SameSite`=Lax (or Strict) cookie on a first-party API host such as api.medibridge.example; rotate and revoke it server-side. On reload, use a credentialed refresh request to restore an access token. Configure Axios credentials and exact-origin credentialed CORS for App.medibridge.example the current production CORS setup does not enable credentialed responses. JavaScript's `Cookies.set()` cannot create an `HttpOnly` cookie.
Do not assume this cookie design will work unchanged on the current api.medibridge.example origin: it is cross-site from app.medibridge.example, so it would require `SameSite`=None; Secure, credentialed CORS, and browser acceptance of third-party cookies. Move the API behind a same-site subdomain or same-origin proxy before relying on refresh cookies.
Replace ?token={`accessToken`} in the WebSocket URL with a cookie-backed handshake that verifies Origin and session state, or a short-lived single-use ticket that is not a reusable bearer URL. Redact upgrade URLs at every proxy and telemetry layer. Test that document.cookie contains no bearer token, cookies have the intended attributes, refresh/logout/reset revoke sessions, and WebSocket URLs and logs contain no reusable credentials.
When making changes to authentication or authorization, make sure to thoroughly test because of its ability to cause outages.

Finding: Provider Role Has Ability to Escalate Privileges via Invitation Feature

MEDIUM

DESCRIPTION

Pigeon Labs engineers discovered that users with the standard provider role can view all invite links that are only supposed to be visible to admins/medical directors. An ordinary provider's invitation-list response contains raw invitation bearer tokens and acceptance URLs, including links for roles above provider. An API check shows this response; the role and network exchange were not independently replayed. The UI disables Copy for non-managers, but still delivers the secrets to their browser.

The user of the Clinic Alpha workspace's Invite Links `page` and its Network response shows active invitations returned with raw token and `inviteUrl` fields. A `lead_provider` invitation had `useCount`: 4 and `maxUses`: null, demonstrating that this link was reusable. The user reports that the session had the provider role. This does not establish an admin invitation, and the standard creation API only accepts provider, `lead_provider`, or `clinical_lead` roles. Live invitation secrets are intentionally omitted from this report.

VULNERABLE ENDPOINTS

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/invitations (disclosure)
```

```
POST https://api.medibridge.example/api/invitations/{token}/accept
```

STEPS TO REPRODUCE

1. Sign in as an ordinary provider of a controlled clinic and open Clinic Settings > Staff > Invite Links. The clinic slug is available in the provider's navigation. Create a disposable active invitation as an `admin/medical_director` beforehand, then list invitations as the provider:

   ```
   curl -i \
     -H "Authorization: Bearer $PROVIDER_ACCESS" \
     "https://api.medibridge.example/api/clinics/$SLUG/invitations"
   ```

2. Confirm that the response contains a token or `inviteUrl` for a disposable active invitation. Redact the secret from test logs and reports. This read-only result is sufficient to verify the disclosure.

3. Optionally, in a disposable clinic, authenticate as a second controlled account and accept a `lead_provider` or `clinical_lead` invitation to demonstrate impact:

   ```
   curl -i -X POST \
     -H "Authorization: Bearer $SECOND_ACCESS" \
     "https://api.medibridge.example/api/invitations/$DISCLOSED_TOKEN/accept"
   ```

4. Confirm that the second account receives the invitation's assigned role. The original provider cannot upgrade their existing membership through this route; acceptance returns their current role. Use only invitations and accounts created for the test.

ROOT CAUSE ANALYSIS

In `clinics.controller.ts`, it exposes the list route. `clinics.service.ts` checks only `VIEW_STAFF`, which the role mapping grants to ordinary providers.

The same service response includes both the raw secret and URL:

```
inv.token,
clinicName: clinic.name,
role: inv.role,
```

The service later sets `inviteUrl` from inv.token. The frontend `page` disables Copy for non-managers but cannot undo a network response already delivered. Standard invitation creation permits provider, `lead_provider`, and `clinical_lead`, not admin.

BUSINESS IMPACT

A provider needs only their own clinic slug and an unexpired invitation already created by an admin/medical_director; its token is directly present in the provider-visible response, so it need not be guessed. Sharing it with an outsider or redeeming it from a separate controlled account could create a `lead_provider` or `clinical_lead`, exceeding the original provider's privileges. The observed `clinical_lead` link had `useCount`: 4 and `maxUses`: null; this did not show acceptance or any admin-role invitation. Medium reflects exposure of reusable role-bearing secrets and potential unauthorized access, subject to an active invitation.

RECOMMENDED REMEDIATION

In `clinics.service.ts`, require `MANAGE_INVITATIONS` to list invitation metadata and stop returning token/`inviteUrl` after creation. Store token hashes, show plaintext only once to the authorized creator, and rotate links exposed during the assessment. Prefer short-lived, recipient-bound, single-use invitations for elevated roles, with atomic redemption; if multi-use links are required, constrain their role and lifetime.

Finding: Webhook Receipt Logging Copies Patient Messages into Operational Logs

LOW

DESCRIPTION

Pigeon Labs engineers found that the Epic and Cerner webhook handlers log full signed payload bodies, potentially copying patient portal messages, lab results, and consultations into operational logs. This path is source-confirmed; a controlled event was not traced into deployed logs, so actual log retention and reader permissions remain unverified.

VULNERABLE ENDPOINTS

```
POST https://api.medibridge.example/api/v1/webhook/epic
```

```
POST https://api.medibridge.example/api/v1/webhook/cerner
```

STEPS TO REPRODUCE

1. Use a staging Epic/Cerner integration with its test signing secret and application-log access; no MediBridge login is needed to submit a webhook, but a valid provider signature is required. A real patient can cause the provider to send such an event by sending a message to a connected clinic via their portal. Put a unique non-sensitive sentinel in the test message.
2. Sign and send the controlled webhook:

   ```
   BODY='{"object":"patient_portal","entry":[{"id":"test-clinic","messaging":[{"message":
   {"text":"LOG_TRACER_9a1b"}}]}]}'
   SIG=$(printf '%s' "$BODY" | openssl dgst -sha256 -hmac "$EPIC_APP_SECRET" -hex | awk
   '{print $2}')
   curl -i -X POST -H 'Content-Type: application/json' \
     -H "X-Hub-Signature-256: sha256=$SIG" \
     --data "$BODY" "https://api.medibridge.example/api/v1/webhook/epic"
   ```

3. Search the test deployment's application logs for `LOG_TRACER_9a1b`; source analysis predicts the complete body will be present.

ROOT CAUSE ANALYSIS

The Epic handler accepts a provider-signed payload. The Cerner handler similarly verifies the event and freshness.

`epic-webhook.controller.ts` includes the raw object for message events:

```
log.info('Webhook received', {
platform: object,
entryCount: entries.length,
body,
});
```

Cerner logs body and `patientId`. The production logger serializes metadata as JSON to console without recursive content redaction.

BUSINESS IMPACT

An external patient need only send a secure message or request a lab result to a connected clinic; the signed provider delivery then carries the content to this handler. No attacker-supplied API token or secret is needed to create ordinary patient content, but reading the resulting logs requires internal log access or a log-platform compromise. The copied messages and identifiers may be searchable, exported, or retained longer than the primary EHR data, broadening the company's privacy exposure and risking HIPAA non-compliance. Low reflects sensitive patient data crossing into a secondary store, conditional on deployed logging and its access controls.

RECOMMENDED REMEDIATION

In the Epic and Cerner receipt blocks, replace body with an allowlisted event record: provider, event type, entry count, request ID, and a keyed hash of the provider event ID. Add recursive redaction in `logger.ts`, restrict log-reader roles and retention, and review historical logs for sensitive ePHI content. Add a signed-sentinel staging test that asserts the message never appears in log output.

Finding: Plaintext Frontend Delivery Exposes Sessions over HTTP to Network Attackers

LOW

DESCRIPTION

Pigeon Labs engineers found that http://app.medibridge.example/ serves the frontend directly instead of redirecting to HTTPS, while its HTTPS response lacks HSTS. Both behaviors were observed passively on production. The frontend also creates access and refresh cookies without Secure, allowing a browser to send those bearer cookies on a plaintext request to the same host.

VULNERABLE ENDPOINTS

```
GET http://app.medibridge.example/
```

```
GET https://app.medibridge.example/
```

STEPS TO REPRODUCE

1. No login is needed for passive transport checks. Request the plaintext frontend from any client and observe whether it serves HTML rather than a redirect:

   ```
   curl -i http://app.medibridge.example/
   ```

   The observed production response on 2028-04-11 was `HTTP/1.1` 200 OK, not a 301 or 308 redirect.

2. Check the HTTPS response for HSTS:

   ```
   curl -sS -D - -o /dev/null https://app.medibridge.example/ | \
   ```

   rg -i '^`strict-transport-security`:'

3. The observed HTTPS response did not include the header. These passive checks are sufficient; do not perform man-in-the-middle testing against real users.

ROOT CAUSE ANALYSIS

The reviewed `Procfile` selects node `server.js`, not the checked-in nginx template.

`server.js` serves the app without a protocol redirect or HSTS header:

```
app.use(express.static('dist'));
```

The production 200 OK on HTTP confirms this behavior at the effective edge, regardless of the unused nginx redirect template.

`identity.ts` creates both bearer cookies without Secure. This makes them eligible for transmission on a same-host HTTP request.

BUSINESS IMPACT

An on-path attacker needs a victim to make an HTTP request to app.medibridge.example while signed in; no clinic ID or role is needed beyond the victim's account. They could read non-Secure access/refresh cookies sent with that request or replace the plaintext login/app response with hostile JavaScript, threatening the victim's account and connected clinical workspaces. The production configuration enables this path. Low reflects direct plaintext delivery plus bearer cookies eligible for transmission, a more concrete exposure than missing HSTS alone.

RECOMMENDED REMEDIATION

At the first trusted edge in front of `server.js`, redirect every HTTP request to HTTPS with 308 before serving any body. Immediately set Secure on credential cookies in `identity.ts`, then migrate to server-issued `HttpOnly`; Secure; `SameSite` sessions as described in the token-storage finding. Add `Strict-Transport-Security`: max-age=31536000; includeSubDomains only after inventorying HTTPS on all affected subdomains. If redirecting in Express, configure trusted proxy headers narrowly. Add deployment tests for HTTP redirect, cookie attributes, and HTTPS HSTS.

Finding: Public Source Maps Expose Private Frontend Implementation

LOW

DESCRIPTION

Pigeon Labs engineers discovered that the production frontend publishes source maps containing readable source from the private frontend repository. An anonymous download returned an approximately 10.3 MB map with embedded bodies for 1,258 sources, including 396 application files under src. The map did not contain the backend repository or any live credentials found in this review.

VULNERABLE ENDPOINTS

```
GET https://app.medibridge.example/assets/index-XyZ12AbC.js.map (hash changes by build)
```

```
GET https://app.medibridge.example/assets/{currentBundle}.js.map
```

STEPS TO REPRODUCE

1. No login, clinic role, or object ID is required. Inspect App.medibridge.example for its current JavaScript bundle, then request that bundle's .map asset. The command below uses the asset hash:

   ```
   curl -sS -o /tmp/medibridge-frontend.js.map \
     https://app.medibridge.example/assets/index-XyZ12AbC.js.map
   ```

2. Count the embedded sources and source bodies:

   ```
   jq '{sources: (.sources | length), embedded: (.sourcesContent | length)}' \
     /tmp/medibridge-frontend.js.map
   ```

3. The observed result contained 1,258 sources and 1,258 embedded source bodies. The asset hash will change between builds, so derive the current map URL from the deployed script when retesting.

ROOT CAUSE ANALYSIS

`vite.config.ts` enables maps:

```
true,
},
```

`server.js` serves the entire dist directory, including generated .map files. The observed map returned 200 OK and a content length of 10,345,441 bytes.

BUSINESS IMPACT

Any anonymous external visitor can download the map; no token, clinic slug, or guessing beyond the public bundle name is required. This reveals readable frontend routes, helpers, consultations, and client-side trust decisions, lowering the effort to study attack surfaces and disclosing private-repository source. It does not grant account access, reveal backend source, or expose a credential in the reviewed map. Low reflects a confirmed confidentiality/configuration issue with limited direct exploit impact.

RECOMMENDED REMEDIATION

Set build.sourcemap to false or a hidden/upload-only mode in `vite.config.ts`, upload maps directly to the authorized error-monitoring service if needed, and exclude *.map from `server.js` and CDN publication. Purge existing cached maps and add a release test that production map URLs return 404 while error-monitoring uploads still work.

Finding: AI Triage Assistant Surfaces Full System Instructions in the Provider UI

LOW

DESCRIPTION

Pigeon Labs engineers observed that the in-app AI triage assistant shows the full LLM system prompt and clinic instruction block to the provider in the browser before the suggested patient reply. This is a confidentiality issue for operational guidance, not a cross-tenant access bug: any provider who can use AI draft on a consultation they already may open can read those instructions.

On clinic-beta consultation 550022, the UI displayed a review banner and verbatim rules (tone, length, knowledge-base use, medical disclaimer policy, etc.) above the generated draft. The behavior was confirmed in the deployed web app; the exact API field or frontend component was not line-traced in this pass. Reproduction is UI-only, see steps below.

VULNERABLE ENDPOINTS

```
In-app Telehealth draft workflow on https://app.medibridge.example/c/{clinicSlug}/
departments/{departmentSlug}/consultations/{consultationId} (LLM-backed reply generation;
exact REST/WebSocket backing call was not isolated in this observation)
```

STEPS TO REPRODUCE

Sign in as a provider with access to clinic-beta and open consultation 550022 in the Inbox UI. Trigger the AI triage/reply assistant (same control used during manual testing that produced the leak). Observed: the UI lists verbatim system instructions and clinic context above the suggested patient reply, including rules such as professional tone, factuality constraints, and medical disclaimer policy. No cross-tenant access was involved; the issue is disclosure to authorized clinical staff on the thread.

Evidence: the UI displayed the following verbatim system instructions above the suggested patient reply.

```
You are helping a triage nurse draft a reply to a patient via the secure portal. Generate
a single reply message that the nurse can review and edit before sending.

Rules:
- Be professional, empathetic, and concise.
- Always include the standard medical disclaimer if symptoms are mentioned.
…
```

ROOT CAUSE ANALYSIS

The client or API response path includes the raw system prompt (or a debug-oriented wrapper) in material rendered to the provider user rather than keeping instructions server-side only. Exact file paths for the Triage LLM service and frontend component were not line-verified in this engagement pass; the defect is confirmed from deployed UI behavior.

BUSINESS IMPACT

Any provider who can draft replies on accessible consultations can read MediBridge's default AI system prompt and clinic-scoped instruction template. That lowers the bar for crafting prompt injection and leaks operational medical guidance that may be intended to stay internal to the model stack. Low reflects authorized-user disclosure without cross-clinic read.

RECOMMENDED REMEDIATION

Stop returning full system prompts to the browser; show only the model's proposed reply and optional short, non-sensitive rationale. If "review instructions" is a product requirement, gate it to admin/medical_director roles or redact meta-instructions. Add a UI/API test asserting system prompt fields are absent from provider-visible JSON.

Finding: Outdated Image Decoder May Compromise the API Worker

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers found that the triage-draft WebSocket path can fetch a caller-supplied attachment URL and decode its bytes with the backend's locked `sharp@0.33.5.` Maintainer advisories identify affected libvips and libheif builds, including possible code execution under specific Linux runtime conditions: `GHSA-f88m-g3jw-g9cj` and `GHSA-rgj7-g3m4-5g8c`. The deployed native-library versions, reachable file formats, and exploitability have not been dynamically confirmed.

VULNERABLE ENDPOINTS

```
wss://api.medibridge.example/?token={accessJwt}, action messages:generate with attachmentUrl
```

STEPS TO REPRODUCE

In isolated staging, sign in as a provider of a controlled clinic with a connected EHR integration and access to the Inbox draft workflow. Confirm the deployed sharp and native-library versions; the reviewed lockfile selects 0.33.5 but does not prove the running version.

1. Host a benign AVIF, TIFF, GIF, or VIPS test file on a controlled server.
2. In isolated staging, send a `messages:generate` action with a disposable account and `attachmentUrl` pointing to the benign test file:

   ```
   printf '%s\n' \
     "{\"action\":\"messages:generate\",\"payload\":{\"clinicSlug\":\"$SLUG\",
   \"integrationId\":$INTEGRATION_ID,\"messages\":[{\"platformMessageId\":\"img-
   test-99\",\"platformThreadId\":\"test-thread\",\"patientName\":\"tester\",\"message\":
   \"analyze image of wound\",\"attachmentUrl\":\"https://$CONTROLLED_HOST/
   test.avif\"}]}}" \
     | websocat -1 "wss://api.medibridge.example/?token=$ACCESS"
   ```

   Confirm the backend fetches and transforms the small benign image. Do not use a weaponized image outside an isolated test environment; reachability plus the affected locked version is sufficient to confirm exposure, not code execution.

ROOT CAUSE ANALYSIS

The WebSocket `messages:generate` payload carries caller-supplied message objects, including `attachmentUrl`, through `generate-messages.handler.ts`.

`triage-llm.service.ts` resolves the URL, fetches up to 15 MiB, and passes the buffer to native image decoding:

```
const response = await axios.get(imageUrl, {
responseType: 'arraybuffer',
timeout: 5000,
maxContentLength: 15 * 1024 * 1024,
});
let imageBuffer = Buffer.from(response.data);
```

The same function resolves `rawUrl` through S3, then invokes the native decoder:

```
const resizedBuffer = await sharp(imageBuffer)
.resize(768, 768, {
fit: 'inside',
withoutEnlargement: true,
})
.jpeg({ quality: 80 })
.toBuffer();
```

The reviewed lockfile selects `sharp@0.33.5.`

BUSINESS IMPACT

An authenticated provider needs their own clinic/integration IDs and a URL serving crafted image bytes; no victim object ID is required. If the deployed native decoders and runtime match the advisory conditions, a malicious image could crash the shared API worker or potentially execute code with access to application credentials and sensitive patient data. Neither outcome was demonstrated. Informational reflects a reachable decode path and affected locked version while withholding a higher rating until deployed formats, native libraries, and exploit conditions are established.

RECOMMENDED REMEDIATION

Upgrade `package-lock.js` on and the deployed native libraries to fixed versions, at least `sharp@0.35.4` for the cited libheif advisory. Until then, block affected loaders such as AVIF/HEIF as advised by the maintainer, and prevent `triage-llm.service.ts` from decoding arbitrary remote formats in a credentialed process. Add byte/pixel `limit` s, allowlisted provider hosts with redirect and resolved-IP checks, and sandboxed decoding. Verify the deployed library versions and harmless format reachability in staging; do not use weaponized images in production.

Finding: Provider OAuth Callbacks Can Attach EHR Accounts Without Permission

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers found that the Epic and Cerner OAuth callbacks permit a clinic provider to attach an external EHR account even though `CONNECT_EHR` belongs only to admins and medical directors. The controllers check clinic membership, not that permission, before exchanging a provider code. The WebSocket sync action has the same permission gap. This is source-confirmed; EHR connection and sync were deferred because no disposable EHR environment account was available.

VULNERABLE ENDPOINTS

```
POST https://api.medibridge.example/api/integrations/platforms/epic/oauth
```

```
POST https://api.medibridge.example/api/integrations/platforms/cerner/oauth
```

```
wss://api.medibridge.example/?token={accessJwt}, action sync:records
```

STEPS TO REPRODUCE

Sign in as a standard provider/nurse in a controlled clinic. The UI's Connect EHR controls are reserved for `admin/medical_director`, so the lower-role user must call the API directly. Obtain the clinic slug from the provider's clinic list.

1. Authorize a disposable Epic or Cerner account and capture a valid, unused OAuth code and matching redirect URI; a normal provider cannot fabricate these provider-issued values.
2. For Epic, submit it as the low-privilege provider:

```
curl -i -X POST \
  -H "Authorization: Bearer $PROVIDER_ACCESS" \
  -H 'Content-Type: application/json' \
  --data "{\"clinicSlug\":\"$SLUG\",\"flow\":\"user\",\"code\":\"$TEST_EPIC_CODE\",
\"redirectUri\":\"$REDIRECT_URI\"}" \
  "https://api.medibridge.example/api/integrations/platforms/epic/oauth"
A 202 from Epic proves only that the lower-role user passed the application permission
boundary and an EHR code was exchanged/queued. Poll GET https://
api.medibridge.example/api/integrations/platforms/epic/oauth/status/{jobId} with that
provider's token and confirm the job completed and a test account appeared in the
clinic integrations list. The user flow ingests patient lists asynchronously; the
system flow stores a connection row before its ingest job. Neither was exercised in
production.
```

3. For the separate Cerner path, use a disposable Cerner sandbox account and a matching unused code:

```
curl -i -X POST https://api.medibridge.example/api/integrations/platforms/cerner/oauth
\
  -H "Authorization: Bearer $PROVIDER_ACCESS" \
  -H 'Content-Type: application/json' \
  --data "{\"clinicSlug\":\"$SLUG\",\"authCode\":\"$TEST_CERNER_CODE\",
\"redirectUri\":\"$REDIRECT_URI\"}"
```

4. Verify account creation only in isolated staging. The WebSocket `sync:records` permission gap is source-confirmed but not included in this EHR-connection reproduction.

ROOT CAUSE ANALYSIS

`clinic-permissions.ts` grants `CONNECT_EHR` to `admin/medical_director`, not `provider/nurse`.

In `epic-oauth.controller.ts`, it accepts an OAuth code and calls:

```
await securityService.verifyUserClinicAccess(userId, clinic.id);
```

The Cerner controller makes the same membership-only check.

The Epic controller branches exchange the code and queue ingestion for `flow: "user"`, or store a system connection before queuing ingestion for `flow: "system"`. The Cerner controller calls its account-creation service after the membership check. `sync-records.handler.ts` likewise verifies membership and account ownership, but not `CONNECT_EHR`, before invoking sync.

BUSINESS IMPACT

A provider or nurse needs a valid session, their clinic slug, and a genuine EHR authorization code for an environment they control; these codes are obtained through OAuth, not guessed in the website. If the source-predicted flow succeeds in deployment, they can attach that integration or trigger sync despite the intended role boundary, importing patient records/messages and creating outbound clinical authority the medical director did not approve. Informational reflects the permission gap and business impact, constrained by EHR authorization and lack of a deployed test.

RECOMMENDED REMEDIATION

In `epic-oauth.controller.ts` and `cerner-oauth.controller.ts`, call verifyUserPermission(`userId`, clinic.id, Permission.`CONNECT_EHR`) before exchanging a code or creating a connection. Apply the same check in `sync-records.handler.ts` before starting sync. Add backend role-matrix tests with mocked EHR responses so no real sandbox account is needed.

Finding: Bearer-only Profile Updates Permit Persistent Account Takeover

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers discovered that an account's password and email can be updated using only its current access JWT, without the current password or recent reauthentication. A disposable account's password-only update returned 204 on production; a follow-up login was rate-limited, so successful login with the replacement password was not independently shown. Email replacement and retention of the verified flag are source-confirmed but were not dynamically tested.

VULNERABLE ENDPOINTS

```
PUT https://api.medibridge.example/api/users/me
```

STEPS TO REPRODUCE

Sign in to a controlled, verified account through App.medibridge.example and retain its access token. No clinic role or object ID is required; this is the Account/Personal Settings profile update path. Use disposable credentials and an email address you control.

1. Change both sensitive attributes without supplying the current password:

   ```
   curl -i -X PUT "https://api.medibridge.example/api/users/me" \
     -H "Authorization: Bearer $ACCESS" \
     -H 'Content-Type: application/json' \
     --data '{"email":"provider@medibridge.example.test","password":"New-Test-Password-42!"}'
   ```

2. Read the profile with the same access token:

   ```
   curl -i -H "Authorization: Bearer $ACCESS" "https://api.medibridge.example/api/users/me"
   ```

   In the production test, Codex sent `PUT /api/users/me` with only a new random password and an existing access token, without the current password. It returned 204, even though the account password had just been changed twice through reset links. No email address was changed. A login check with the new password was blocked by the login endpoint's 429 rate `limit`, so successful login with the replacement password was not independently verified at that time. Source analysis predicts that an email change would also be accepted while preserving the verified flag; that part was not exercised. Restore the test account after validation.

ROOT CAUSE ANALYSIS

In `users.controller.ts`, it requires only a valid JWT and passes body.email and body.password to the service. The DTO explicitly accepts both fields.

`users.service.ts` stores the new values without a current-password or recent-auth check:

```
email ?? existUser.email,
name: name ?? existUser.name,
password: password ? await encryptPassword(password) : existUser.password,
role: existUser.role,
isEmailVerified: existUser.isEmailVerified,
```

Preserving isEmailVerified when email changes is a separate source-predicted verification bypass.

BUSINESS IMPACT

An attacker needs a still-valid access JWT for a target user; the browser stores these JWTs in script-readable cookies, but theft was not demonstrated here. No clinic role, current password, victim record ID, or MFA proof is required by the update endpoint. A password change can extend temporary session theft into durable account access; an email change could redirect recovery and retain verified status without proving control of the new mailbox. The company risks clinical account takeover and control of connected healthcare assets. Informational reflects the sensitive write guarded only by a bearer token, while noting that getting access to a valid JWT is not trivial.

RECOMMENDED REMEDIATION

In `users.controller.ts` and `users.service.ts`, require and verify the current password or a recent step-up event before either sensitive change. Stage a new email separately, verify it out of band, then promote it and reset its verification state. Revoke existing sessions after a password change and notify the former email address. Add tests that a bearer token alone cannot change email/password, that an unverified new address stays unverified, and that old sessions fail after an authorized change.

Finding: Older Reset Links Can Override a Completed Password Recovery

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers discovered that issuing a newer password-reset link and completing recovery do not invalidate an older unused link. On a disposable production account, the older link changed the password after the user had used the newer one. Each reset link still has a 15-minute expiry; this finding concerns overlapping live tokens, not indefinite validity.

VULNERABLE ENDPOINTS

```
POST https://api.medibridge.example/api/auth/reset-password
```

```
POST https://api.medibridge.example/api/auth/update-password
```

```
POST https://api.medibridge.example/api/auth/reset-token-preview
```

STEPS TO REPRODUCE

For a controlled account whose mailbox you own, request two reset emails from App.medibridge.example (or the equivalent API request below) and retain token 1 and token 2. No logged-in session or clinic role is required to use a reset link; obtaining a victim's link would require mailbox access, interception, or another disclosure.

```
curl -i -X POST https://api.medibridge.example/api/auth/reset-password \
  -H 'Content-Type: application/json' \
  --data '{"email":"provider@medibridge.example.test"}'
```

Repeat once for the same controlled mailbox and identify which emailed token is newer.

1. Complete a password reset using token 2:

   ```
   curl -i -X POST https://api.medibridge.example/api/auth/update-password \
     -H 'Content-Type: application/json' \
     --data "{\"token\":\"$TOKEN_2\",\"password\":\"New-Test-Password-42!\",
   \"confirmPassword\":\"New-Test-Password-42!\"}"
   ```

2. Check token 1 without changing the password again:

   ```
   curl -i -X POST "https://api.medibridge.example/api/auth/reset-token-preview" \
     -H 'Content-Type: application/json' \
     --data "{\"token\":\"$TOKEN_1\"}"
   ```

3. With explicit authorization on the disposable account, submit token 1 with a different controlled test password:

```
curl -i -X POST https://api.medibridge.example/api/auth/update-password \
  -H 'Content-Type: application/json' \
  --data "{\"token\":\"$TOKEN_1\",\"password\":\"Another-Test-Password-43!\",
\"confirmPassword\":\"Another-Test-Password-43!\"}"
```

In the production test, two reset requests returned 200 around 14:22 UTC. The older token returned 200 from `reset-token-preview` at 14:30 UTC. At 14:32 UTC, the newer user-provided token returned 400 with Token has already been used, while the older token still returned 200 from the same preview endpoint. The user confirmed using the newer link for a password reset. Codex then submitted the older token with a randomly generated test password at 14:33 UTC; production returned 200. A subsequent preview of the older token returned 400 with Token has already been used, confirming the second password change. The reset tokens, account email, and test password are omitted.

ROOT CAUSE ANALYSIS

`auth.service.ts` creates a new hashed reset token with a 15-minute expiry but does not invalidate earlier records:

```
await this.authRepository.createResetToken(
new PasswordResetToken({
tokenHash,
userId: user.id,
expiresAt: new Date(Date.now() + 15 * 60 * 1000), // 15 minutes
})
);
```

`auth.service.ts` validates and marks only the presented token used after changing the password. The repository already offers `invalidateUnusedResetToken`sForUser, but this flow does not call it.

BUSINESS IMPACT

An attacker needs the victim's older reset token, which is a high-entropy bearer secret delivered by email and not enumerable from the website. If obtained before its 15-minute expiry, it remains effective after the victim completes a newer reset, allowing the attacker to replace the password and regain control. The production test demonstrated that sequence on a disposable account. Informational reflects a direct account-recovery bypass constrained by prior token possession and a short validity window.

RECOMMENDED REMEDIATION

In resetPassword, call the repository's `invalidateUnusedResetTokensForUser(user.id)` before creating a replacement token. In updatePassword, invalidate all remaining reset tokens for that user as part of an atomic password-change transaction. Preserve hashed storage and short expiry, and add tests that an older link fails both after a newer link is issued and after that newer link is redeemed.

Finding: Public Verification Responses Reveal Registered Email Addresses

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers discovered that the unauthenticated email-verification endpoint distinguishes registered from unregistered addresses. A single controlled production comparison returned 202 for a registered address with an invalid code and 404 User not found for a synthetic unregistered address. No address list or distributed enumeration was tested.

VULNERABLE ENDPOINTS

```
POST https://api.medibridge.example/api/users/verify
```

STEPS TO REPRODUCE

No login or clinic role is needed. From the signup/email-verification flow, choose one owned registered mailbox and one synthetic unregistered address. The candidate email is the only identifier the caller must know.

1. Submit the same invalid code for both:

   ```
   curl -i -X POST "https://api.medibridge.example/api/users/verify" \
     -H 'Content-Type: application/json' \
     --data '{"email":"known-controlled@example.test","code":"ABCDEF"}'
   curl -i -X POST "https://api.medibridge.example/api/users/verify" \
     -H 'Content-Type: application/json' \
     --data '{"email":"definitely-unregistered@example.test","code":"ABCDEF"}'
   ```

   Compare status codes, response bodies, and timing. In production, the synthetic unregistered address returned 404 with User not found; the disposable registered address returned 202. Both requests omitted authorization, used the same deliberately invalid six-character code, and were sent once. The registered address was obtained from its owner's authenticated `GET /api/users/me` response and is omitted here.

ROOT CAUSE ANALYSIS

In `users.controller.ts`, it accepts email and code without `@Security`.

`users.service.ts` throws before OTP verification for an absent account:

```
const existingUser = await this.userRepository.findUserByEmail(email);
if (!existingUser) {
throw new NotFoundError('User not found');
}
```

Existing accounts instead reach validateOTP, which maps provider errors to a canceled result. The user API `limit`er does not remove the semantic difference.

BUSINESS IMPACT

An unauthenticated external caller needs only a candidate email address, commonly obtainable from public hospital contacts or prior data sets. The distinct response reveals whether that address has a MediBridge account, aiding targeted phishing and privacy inference about medical staff. The test covered two controlled addresses, not bulk enumeration. Information reflects a real but limited disclosure rather than account access or patient-data exposure.

RECOMMENDED REMEDIATION

In `users.service.ts`, make absent and invalid-code cases return the same status, body schema, and approximately equivalent timing without revealing account existence. Apply a stricter endpoint-specific `limit`er in `app.ts` using IP and privacy-preserving per-address keys. Add tests for existing and absent addresses with the same invalid code, and retain detailed reasons only in protected telemetry.

Finding: Unbounded Consult Pages Permit Oversized Database Reads

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers found that authenticated list routes accept `page` sizes or limits without an application maximum. On the production message endpoint, a bounded comparison returned 101 records when `pageSize`=101, rather than enforcing the 100-item cap used elsewhere. Larger database and memory effects, and the other listed routes, remain source-predicted rather than load-tested.

VULNERABLE ENDPOINTS

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/patients?integrationId={integrationId}&pageSize={n}
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/messages?integrationId={integrationId}&pageSize={n} (cap bypass confirmed)
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/departments/{departmentSlug}/consultations?pageSize={n}
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/departments/{departmentSlug}/consultations/{id}?limit={n}
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/departments/{departmentSlug}/triage-agent/transcripts?limit={n}
```

```
GET https://api.medibridge.example/api/clinics/{clinicSlug}/departments/{departmentSlug}/triage-agent/transcripts/{consultationId}?limit={n}
```

STEPS TO REPRODUCE

Sign in as an active provider of a controlled clinic with more than 100 messages on one connected EHR account. Obtain the clinic slug and `integrationId` from the normal Inbox UI or integrations API. No victim data is needed. Use staging for broader cost or load measurements.

1. Compare a normal `page` size with one just above the intended maximum, without starting an LLM job:

   ```
   curl -i \
     -H "Authorization: Bearer $ACCESS" \
     "https://api.medibridge.example/api/clinics/$SLUG/messages?
   integrationId=$INTEGRATION_ID&page=1&pageSize=25&startLLMJob=false"
   curl -i \
     -H "Authorization: Bearer $ACCESS" \
     "https://api.medibridge.example/api/clinics/$SLUG/messages?
   integrationId=$INTEGRATION_ID&page=1&pageSize=101&startLLMJob=false"
   ```

   With a valid clinic-alpha access token, `GET /api/clinics/clinic-alpha/messages?integrationId=801&page=1&pageSize=25&startLLMJob=false` returned 25 records with total: 105. Changing only `pageSize` to 101 returned 101 records with total: 105. No message content was needed for this comparison.

2. Repeat the equivalent controlled request against consultations and agent transcript endpoints using `pageSize`=101 or `limit`=101 as applicable. Measure query counts, rows, latency, and memory only in a staging environment; do not use extreme values against production.

ROOT CAUSE ANALYSIS

In `messages.controller.ts`, it accepts optional `page` and `pageSize` without an upper bound. The patient, consultation, and agent transcript controllers also accept caller-controlled `limit`s.

In multi-account message listing, `messages.service.ts` asks each adapter for `page` * `pageSize` rows before flattening and sorting:

```
1,
pageSize: page * pageSize,
```

Patient repositories pass the size to ORM .take(); the consultation and transcript services likewise accept unbounded `limit`s. These effects were traced in source, not measured under heavy load.

BUSINESS IMPACT

An authenticated provider needs a clinic with enough stored messages/patients and their own clinic/integration IDs; all are available through normal UI/API access. A large `pageSize` can make the server read and sort more records than a normal `page`, potentially affecting other customers if repeated or combined across accounts. The deployed test proved only the 100-item cap bypass, not exhaustion or cross-tenant slowdown. Low reflects that bounded evidence and the need for larger data or sustained requests before a service-impact claim is justified.

RECOMMENDED REMEDIATION

In the message, patient, consultation, and agent schemas, require positive integers and a consistent maximum such as 100; enforce the same cap again in services/repositories. Replace the multi-account fetch with a bounded merge or keyset pagination.

Finding: Public Readiness Checks Reveal Dependencies Outside API Throttling

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers found that the unauthenticated `/ready` endpoint reports database and Redis status while sitting before the API's global limiter. One production request returned both component states without rate-`limit` headers. No sustained traffic or correlated dependency metrics were collected, so service exhaustion is a plausible source-predicted effect.

VULNERABLE ENDPOINTS

```
GET https://api.medibridge.example/ready
```

STEPS TO REPRODUCE

1. No login, clinic role, or identifier is required. From any external client, request the public health URL once; use only an authorized staging deployment for the following small rate-`limit` comparison:

   ```
   for i in $(seq 1 20); do
   curl -sS -o /dev/null -D - "https://api.medibridge.example/ready"
   done
   ```

2. Confirm that responses do not carry or enforce the application's normal rate-`limit` state.
3. Correlate the 20 requests with staging PostgreSQL query and Redis command metrics to confirm one probe of each dependency per request. Do not perform sustained testing against production.
4. On production, a single unauthenticated `GET /ready` returned 200 with {"status":"ok","db":"ok","redis":"ok"} and no R`ateLimit`-* headers. A single unauthenticated `GET /api/users/me` control returned 401 with `RateLimit-Limit`: 300, while `/health` returned 200 with {"status":"ok"} and no rate-`limit` headers. These observations confirm public reachability and component disclosure, not rate-`limit` exhaustion or measurable database/Redis amplification.

ROOT CAUSE ANALYSIS

In `app.ts`, it registers an unauthenticated route.

Each request performs a database query and Redis ping, then returns their separate states:

```
await db.connection?.query('SELECT 1');
checks.db = 'ok';
```

The Redis check similarly calls `client.ping()`.

`app.ts` registers `globalLimit`er after `/ready`, so it cannot govern this earlier route.

BUSINESS IMPACT

Any external caller can learn whether PostgreSQL or Redis is degraded and trigger one check of each per request without a user account. This may reveal incident timing and contribute to dependency load during an outage, but neither saturation nor metrics correlation was tested. Informational reflects confirmed public component disclosure and middleware ordering with limited demonstrated business harm.

RECOMMENDED REMEDIATION

Keep `/health` as a cheap public liveness route, but restrict `/ready` to platform health-check sources at the edge or authenticate it. If it must remain public, move a strict `limit`er ahead of it, cache probe results briefly, and return only a generic status externally. Add staging tests that unauthenticated public traffic cannot repeatedly drive PostgreSQL and Redis checks or reveal component names.

Finding: Missing Frame Restrictions Permit Clickjacking of Signed-in Users

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers discovered that the production frontend omits both CSP frame-ancestors and X-Frame-Options. Passive response inspection confirmed the missing headers. A hostile origin can attempt to frame the app and induce clicks, but a signed-in UI action was not tested and browser third-party cookie restrictions may `limit` exploitability.

VULNERABLE ENDPOINTS

```
GET https://app.medibridge.example/
```

```
GET https://app.medibridge.example/clinic/{clinicSlug}
```

STEPS TO REPRODUCE

1. No login is needed to inspect response headers. Request the public frontend and check for framing restrictions without changing application state:

   ```
   curl -sS -D - -o /dev/null https://app.medibridge.example/ | \
   ```

   rg -i '^(content-security-policylx-frame-options):'

2. For browser validation only, sign in to a disposable clinic account, then serve this file from a controlled HTTPS origin and open it in that same browser:

   ```
   <!doctype html>
   <title>Frame test</title>
   <iframe
   src="https://app.medibridge.example/"
   style="position:fixed;inset:0;width:100%;height:100%;opacity:.35">
   </iframe>
   ```

3. Confirm that the browser renders the application in the frame. Do not attempt billing, membership, or other state-changing clicks outside a controlled test account.

ROOT CAUSE ANALYSIS

The reviewed `Procfile` starts node `server.js`.

`server.js` serves static files and a fallback document without setting either framing header:

```
app.use(express.static('dist'));
app.get('/{*splat}', async (req, res) => {
```

The nginx template also omits them. Production HTTPS response inspection confirmed the absence at the effective edge.

BUSINESS IMPACT

An attacker needs to persuade a signed-in clinic user to visit a hostile `page` and interact with a disguised frame; they do not need an API token or target object ID. If the browser allows the application session in that frame, the user might click settings, membership, or billing controls unintentionally. No state-changing UI test was performed, and cookie/storage policy may block an authenticated frame. Informational reflects confirmed missing headers but conditional session exposure and required user interaction.

RECOMMENDED REMEDIATION

Set Content-Security-Policy: frame-ancestors 'none' at the outermost frontend response layer and X-Frame-Options: DENY as a legacy fallback. If legitimate embedding exists, use a narrow allowlist and keep privileged routes unframeable. Add a production-header regression check for both static assets and fallback `page`s, plus a controlled browser frame test.

Finding: Unscoped WebSocket Consultation IDs Leak Viewer Presence on Protected Threads

INFORMATIONAL

DESCRIPTION

Pigeon Labs engineers identified a WebSocket side channel: subscribing with `consultation:view` can broadcast `presence:update` events (who is viewing a thread) even when REST denies access to that consultation's messages. Only clinic membership is enforced for the view action not the same consultation-level checks used on HTTP inbox routes.

From a clinic-alpha session, consultation 550022 (a clinic-beta thread blocked in the UI and over REST) still yielded presence updates listing other viewers (for example an attending physician's user id and display name). No ePHI bodies appeared in this trace the issue is live viewing metadata, not full thread disclosure. Details and example frames are in the steps below.

VULNERABLE ENDPOINTS

```
wss://api.medibridge.example/?token={accessJwt} (server may broadcast presence:update)
action consultation:view
```

STEPS TO REPRODUCE

1. Sign in as a user with membership in clinic-alpha who cannot access consultation 550022 via REST (Clinic Beta triage thread). Confirm denial:

   ```
   curl -sS -i \
     -H "Authorization: Bearer $ACCESS" \
     "$API/api/clinics/clinic-alpha/departments/{departmentSlug}/consultations/550022?
   limit=5"
   ```

   Observed: 403 or 404 - no message/thread access for that JWT.

2. Open wss://api.medibridge.example/?token=`$ACCESS` and send:

```
{
"action": "consultation:view",
"payload": {
"clinicSlug": "clinic-alpha",
"consultationId": 550022
}
}
```

Observed: inbound `presence:update` for `consultationId` 550022 including viewers with clinical staff identity fields. Redact bearer tokens in exported material.

ROOT CAUSE ANALYSIS

The WebSocket registry treats `consultation:view` as a presence subscription with broadcast delivery to clinic members. Incomplete authorization: `consultation-presence.handler.ts` handles the view action without applying the same consultation access predicate REST uses before publishing presence. Sink: Broadcast `presence:update` exposes viewer lists for the supplied `consultationId` even when REST would deny reading that consultation.

BUSINESS IMPACT

An insider or compromised provider account learns who is actively viewing patient threads they cannot open in the app, enabling surveillance and social-engineering scenarios ("why is the medical director viewing this patient's lab results?"). Spoofed presence on inaccessible threads was not fully demonstrated. Informational, reflects clear authorization bypass on a side channel without ePHI body disclosure in the captured response, however the impact to patient privacy requires attention.

RECOMMENDED REMEDIATION

Before handling `consultation:view`, enforce shared verifyConsultationAccess (or equivalent) for `consultationId` and `clinicSlug`. Reject unauthorized ids with an error frame and do not emit `presence:update` with viewer lists. Optionally `limit` viewer detail returned to clients that pass ACL. Add a REST-deny / WS-deny parity test for a forbidden consultation id.

MediBridge Pentest Appendix

Application Overview

MediBridge is a multi-tenant healthcare communications platform. Clinical practices organize work into clinics and departments, connect Epic, Cerner, Athenahealth, and Patient Portal accounts, and use the web application to manage secure patient messages, run clinical workflow automations, and prepare AI-assisted triage responses. Clinic membership and roles govern access to connected integrations, invitations, and billing functions.

The React/TypeScript frontend communicates with a TypeScript/Express HTTP API and a WebSocket service for real-time telehealth activity. The backend also runs background workers and uses PostgreSQL, Redis, and S3-backed media storage, with integrations including EHR platform APIs and Stripe. The central security boundaries are between clinical staff and patient records, between a clinic and its connected EHR integrations, and between MediBridge's backend and external vendor services. Patient messages and provider-linked actions pass through these boundaries, making authorization, HIPAA compliance, and session handling especially critical.

Scope and Boundaries

This pentest confirmed **21 findings** concerning the MediBridge web application. The assessment drew on the private Backend and Frontend source packages and on targeted observation of these deployed interfaces:

- **Web application:** https://app.medibridge.example
- **HTTP API and WebSocket service:** https://api.medibridge.example (`/api` for the HTTP API)

The reviewed surface included authentication and session handling, clinic and EHR-account authorization, patient messages and secure telehealth consultations, billing integration, invitations, browser security controls, and selected workflow and AI-assisted triage features. The findings identify the specific routes and roles examined.

Third-party EHR platforms, Stripe, cloud-provider control planes, and other vendor infrastructure were considered only where they intersected with MediBridge's application behavior. They were not independently assessed. Tests that could change real patient data, send provider messages, extract ePHI in bulk, or impose sustained production load were limited or deferred as described in the individual findings.

Methodology

The assessment combined code review and dynamic application testing, guided by the OWASP Web Security Testing Guide (WSTG) V4.2. AI-assisted agents supported source-code review and performed a limited number of low-impact, non-destructive dynamic checks. Relevant WSTG areas included information gathering, configuration, authentication, authorization, session management, input validation, business logic, client-side security, and API testing. The guide informed test selection; this was not a claim that every WSTG test case was executed.

1. **Map the application:** Review browser behavior, HTTP requests and responses, cookies, security headers, WebSocket actions, role-based UI paths, and the corresponding frontend and backend routes.
2. **Trace code paths:** follow untrusted inputs through controllers or socket handlers, services, adapters, repositories, and external integrations to sensitive reads, writes, queries, and browser sinks. Compare each path's authentication and authorization checks with the object or action ultimately used.
3. **Develop and execute business-logic cases:** Define expected outcomes from MediBridge's own clinic and integration ownership rules, role permissions, invitation visibility, billing access, and session lifecycle. Test applicable cases against the application with controlled accounts and data, comparing what each role or tenant should be allowed to do with the observed HTTP, WebSocket, and browser behavior.
4. **Validate technical behavior:** Use controlled accounts, clinics, and disposable data for targeted HTTP, WebSocket, and browser checks. Compare permitted and denied requests, roles, tenants, and token states where feasible. Isolated local tests were used for some query-builder and dependency behavior.
5. **Corroborate assisted analysis:** Agent-generated hypotheses and test results were checked against the source code or observed application responses before inclusion. Agent output alone was not treated as proof of a vulnerability.
6. **Record limits:** Each finding distinguishes deployed observations from source-based predictions and isolated tests. Where a production result was supplied by the user rather than independently replayed, or where a provider-side effect was not observed, the finding says so.

Severity Ratings

Ratings are qualitative and engagement-specific. They consider the likely attacker and required access, how easily identifiers or secrets can be obtained, reliability of exploitation, the sensitivity and scale of the affected data or action, and the resulting business impact. This follows the likelihood-and-impact principle in the OWASP Risk Rating Methodology; it is not a numerical OWASP or CVSS calculation. Evidence confidence is separate from severity: a source-only finding is not automatically harmless, and an unproven worst-case outcome does not justify a high rating.

Rating	Meaning
Critical	A practical path to widespread compromise, major unauthorized control, or comparably severe business harm with few meaningful prerequisites. Requires immediate attention. No finding in this report is rated Critical.
High	A credible path to substantial ePHI exposure, account takeover, privilege escalation, or serious service compromise. Prioritize urgently, even if some prerequisites apply.
Medium	Meaningful unauthorized access, modification, or disruption, but with bounded impact or material exploitation constraints. Prioritize for remediation.
Low	Limited or conditional impact, or a path with restrictive prerequisites that reduces its realistic risk. Address through planned hardening.
Informational	A security-relevant weakness or hardening observation for which practical impact is minimal or not yet established. Investigate and address as appropriate; this label does not mean the behavior is desirable.

Evidence and Limitations

The report combines deployed-application tests, manual source review, user-provided test evidence, and isolated local checks. These evidence types are not interchangeable. A source-level authorization gap does not prove that a third-party EHR provider accepted a request, and a successful WebSocket handshake does not by itself prove that a protected action succeeded. Individual findings state those distinctions where they matter.

The assessment is a point-in-time review. Source snapshots and deployed behavior can change independently; retesting is needed to confirm that a fix is present in the running application. Credentials, live invitation links, and other bearer secrets are omitted from reproduction examples, which use placeholders instead.