027-Schwachstellen-API-Haertung

IT-Sicherheitsaufgabe
Anforderungsniveau: Schwer

1 Ziel der Aufgabe

Sie erhalten unten den vollständigen Quelltext einer absichtlich verwundbaren REST-API für ein Wartungsportal. Sie übernehmen ihn, bringen ihn zum Laufen, auditieren ihn, belegen jeden Fund reproduzierbar, beheben ihn und sichern die Behebung durch Regressionstests ab. Der Reiz liegt in der Reihenfolge: erst der Beleg, dann der Test, dann der Fix – ein Fix ohne vorher rot gelaufenen Test ist wertlos, weil niemand prüfen kann, ob er greift.

Veranschlagte Bearbeitungszeit: 5–7 Stunden

Verwendete Techniken: Node.js mit TypeScript, `express` und PostgreSQL über `pg`; JWT mit `jsonwebtoken`, Passwort-Hashing mit `argon2` oder `bcrypt`, Validierung mit `zod`; Schutzschichten mit `helmet`, `cors` und `express-rate-limit`; Docker Compose für Anwendung und Datenbank; Tests mit `vitest` (oder `jest`) und `supertest`; Klassifikation nach OWASP API Security Top 10 (2023). Alternativ ist Rust mit `axum` und `sqlx` zulässig; die Schwachstellenklassen müssen dann sinngemäß dieselben sein.

2 Aufgabenbeschreibung

1. **Ausgangszustand herstellen**

   Zwei Tabellen, vier Endpunkte, zwei Rollen – mehr ist es nicht. Legen Sie das Schema an und übernehmen Sie den Quelltext *mit allen Fehlern.* Committen Sie diesen Stand unter dem Tag `v0-verwundbar`. Rollen sind `kunde` (darf nur Tickets der eigenen `kunden_nr` sehen und nur das eigene Konto ändern) und `admin` (darf alles).

```
nutzer(id, email, passwort_hash, rolle, kunden_nr, erstellt_am)
ticket(id, kunden_nr, titel, beschreibung, status, erstellt_am)
```

```
// app.ts -- der komplette verwundbare Ausgangszustand
const GEHEIMNIS = "geheim123";
app.use(cors({ origin: (o, cb) => cb(null, true), credentials: true }));
app.use(express.json());

app.post("/auth/login", async (req, res) => {
  const { rows } = await db.query("SELECT * FROM nutzer WHERE email = $1",
                                  [req.body.email]);
  if (!rows[0]) return res.status(404).json({ fehler: "Konto unbekannt" });
  if (!(await argon2.verify(rows[0].passwort_hash, req.body.passwort)))
    return res.status(401).json({ fehler: "Passwort falsch" });
  res.json({ token: jwt.sign({ sub: rows[0].id, rolle: rows[0].rolle,
                               kunden_nr: rows[0].kunden_nr }, GEHEIMNIS) });
});

app.use("/tickets", (req, res, next) => {
  req.nutzer = jwt.decode((req.headers.authorization || "")
                          .replace("Bearer ", ""));
  next();
```

```
});

app.get("/tickets", async (req, res) => {
  res.json((await db.query(`SELECT * FROM ticket
      ORDER BY ${req.query.sortieren || "erstellt_am"}
               ${req.query.richtung || "DESC"}
      LIMIT ${req.query.limit || 50}`)).rows);
});

app.get("/tickets/:id", async (req, res) =>
  res.json((await db.query("SELECT * FROM ticket WHERE id = $1",
                           [req.params.id])).rows[0]));

app.patch("/nutzer/:id", async (req, res) => {
  const felder = Object.keys(req.body);
  const zuweisung = felder.map((f, i) => `${f} = $${i + 2}`).join(", ");
  await db.query(`UPDATE nutzer SET ${zuweisung} WHERE id = $1`,
                 [req.params.id, ...felder.map((f) => req.body[f])]);
  res.json((await db.query("SELECT * FROM nutzer WHERE id = $1",
                           [req.params.id])).rows[0]);
});

app.use((fehler, req, res, next) =>
  res.status(500).json({ fehler: fehler.message, stack: fehler.stack }));
// keine Ratenbegrenzung, helmet ist nicht eingebunden
```

2. **Angriffsfläche erfassen**

   Legen Sie `SICHERHEIT.md` an und beschreiben Sie zuerst das Ziel, bevor Sie darauf schießen: eine Tabelle aller vier Endpunkte mit erwarteter Rolle, Datensichtbarkeit und Vertrauensgrenze. Diese Tabelle ist Ihr Maßstab – eine Schwachstelle ist eine Abweichung davon, kein Bauchgefühl.

3. **Schwachstellen finden und belegen**

   In den vierzig Zeilen oben stecken mehr als sechs Schwachstellen. Die folgende Liste nennt die Klassen, nicht die Fundstellen.

   - **Zugriffskontrolle** – fremde Datensätze über die ID erreichbar (BOLA/IDOR), kein Endpunkt prüft, wem das Objekt gehört
   - **Token-Prüfung** – Signatur nicht verifiziert, Algorithmus und Ablauf ungeprüft, Geheimnis im Quelltext, fehlendes Token ohne `401`
   - **Massenzuweisung und Datenpreisgabe** – `rolle` oder `kunden_nr` über den Rumpf setzbar; Passwort-Hashes und Felder anderer Mandanten in Antworten
   - **Einschleusung** – Verkettung in `ORDER BY` und `LIMIT`; auch der Spaltenname in `PATCH /nutzer/:id` stammt ungeprüft aus dem Rumpf
   - **Ressourcenverbrauch, Preisgabe, Fehlkonfiguration** – keine Obergrenze für `limit`, keine Ratenbegrenzung, unterscheidbare Anmeldefehler (Kontenaufzählung), Stacktrace in der Antwort, CORS spiegelt jeden Ursprung und erlaubt zugleich Anmeldedaten

   Jeder Befund kommt in `SICHERHEIT.md` mit Nummer, Endpunkt, OWASP-Klasse, Auswirkung in einem Satz, reproduzierbarem Beleg samt beobachteter Antwort und Verweis auf den Regressionstest.

```
# Beleg 01 -- Kunde liest fremdes Ticket
curl -s -H "Authorization: Bearer $TOKEN_KUNDE_A" \
```

```
    http://localhost:3000/tickets/4711 | jq '.kunden_nr, .titel'
# erwartet 403 -- beobachtet 200 mit dem Ticket von Kunde B
```

Ein Befund ohne Beleg zählt nicht; ein Beleg, der nur „geht nicht“ zeigt, ebenso wenig.

4. **Regressionstests zuerst schreiben**

   Für jeden Befund mindestens ein Test, der den Angriff über HTTP nachstellt. Er muss gegen `v0-verwundbar` *rot* laufen; halten Sie die Ausgabe dieses Laufs als `tests/ausgangslauf.txt` fest. Führen Sie die Tests gegen den Ausgangszustand aus, indem Sie die verwundbaren Quellen mit `git checkout v0-verwundbar -- src/` vorübergehend zurückholen.

```
it("verweigert Zugriff auf ein fremdes Ticket", async () => {
  const antwort = await request(app).get(`/tickets/${ticketVonKundeB}`)
    .set("Authorization", `Bearer ${tokenKundeA}`);
  expect(antwort.status).toBe(403);
  expect(JSON.stringify(antwort.body)).not.toContain("Kessel undicht");
});
```

   Prüfen Sie nicht nur den Statuscode, sondern auch, dass die Nutzlast die geschützten Werte nicht enthält – ein `403` mit dem Datensatz im Rumpf ist schon vorgekommen.

5. **Schwachstellen beheben**

   Ein Commit je Befund, Befundnummer in der Commit-Nachricht. Dabei gilt:

   - Zugriffskontrolle wird an der Datenzugriffsschicht erzwungen, nicht durch Ausblenden im Ergebnis: die Abfrage lädt gar nicht erst, was der Aufrufer nicht sehen darf. Tokens prüft `jwt.verify` mit festem `algorithms` und Ablauf; das Geheimnis kommt aus der Umgebung, fehlt es, startet die Anwendung nicht.
   - Jede Eingabe passiert ein `zod`-Schema; für Sortierfeld, Richtung und änderbare Spalten gilt eine Positivliste statt Ausfiltern verbotener Zeichen. Antworten entstehen aus einem Ausgabeschema (Positivliste der Felder), nicht durch Löschen einzelner Felder.
   - `helmet`, CORS-Positivliste konkreter Ursprünge, `express-rate-limit` an `/auth/*`, Obergrenze für `limit`, Fehler-Handler ohne Stacktrace nach außen. Die Anmeldung antwortet gleich, ob das Konto existiert oder nicht – in Text, Statuscode und näherungsweise Laufzeit.

6. **Nachweisen, dass die Tests greifen**

   Ein grüner Test, der nie rot werden kann, sichert nichts. Machen Sie für mindestens zwei Befunde die Gegenprobe – Fix zurücknehmen, Testlauf zeigen, Fix wiederherstellen – und halten Sie fest, welcher Test mit welcher Meldung rot wurde. Da die Behebung bereits committet ist, genügt `git stash` dafür nicht – nehmen Sie den Fix mit `git checkout v0-verwundbar -- src/app.ts` zurück und stellen Sie ihn danach mit `git checkout HEAD -- src/app.ts` wieder her.

7. **Funktionsfähigkeit und Betrieb**

   Härtung, die die API unbrauchbar macht, ist keine Lösung: mindestens vier Tests für den erlaubten Weg – Anmeldung, eigene Ticketliste lesen, Liste sortiert abrufen, eigenes Konto ändern –, grün vor und nach der Härtung. Eine `docker-compose.yml` liegt bei, die Konfiguration kommt nur aus Umgebungsvariablen, eine `.env.beispiel` mit erfundenen Werten ist eingecheckt, die echte nicht.

3 Anforderungen

- Der Ausgangszustand ist als Tag `v0-verwundbar` erreichbar. Mindestens sechs dokumentierte Befunde, jeder mit Klasse, Auswirkung, Beleg und zugeordnetem Regressionstest.
- Jeder Sicherheitstest war gegen `v0-verwundbar` nachweislich rot und ist im Endstand grün; der Ausgangslauf liegt im Repository, für mindestens zwei Befunde auch die Gegenprobe. Ein Commit pro Befund mit Befundnummer in der Nachricht.
- Zugriffskontrolle serverseitig durchgesetzt, kein Endpunkt glaubt dem Aufrufer seine Rolle. Keine Zeichenkettenverkettung in SQL, Sortierfelder und änderbare Spalten über Positivliste. JWT-Prüfung mit festem Algorithmus und Ablaufprüfung, Geheimnis nur aus der Umgebung.
- Antwortschemata sind Positivlisten; Passwort-Hashes und Datensätze fremder Mandanten erscheinen in keiner Antwort an die Rolle `kunde`. Fehlermeldungen ohne Stacktrace oder SQL-Fragmente.
- **Keine echten Geheimnisse als Testdaten**: erfundene Werte (`hunter2`, `example.invalid`) und der Nachweis, dass auch die *Historie* frei davon ist, nicht nur der Arbeitsbaum.
- **Kein Formatierer ohne Projektkonfiguration**: eine `.prettierrc` oder ESLint-Konfiguration wird mit eingecheckt, sonst bleibt die Formatierung unangetastet – ein Sicherheits-Commit, der nebenbei 400 Zeilen umformatiert, ist nicht prüfbar.

4 Abgabe

- Git-Repository mit nachvollziehbarer Historie und `README.md` mit Startanleitung in dieser Form:

```
cp .env.beispiel .env && docker compose up -d datenbank
npm install && npm run migrieren && npm run seed
npm run dev    # API auf http://localhost:3000 -- npm test für alle Tests
```

- `SICHERHEIT.md` als eigentlicher Bericht: Angriffsfläche, Befundtabelle, je Befund Beleg und Behebung, die Gegenproben und ein Abschnitt „offen geblieben“ mit begründeten Punkten, die Sie erkannt, aber bewusst nicht behoben haben.
- Die Belege als ausführbare Sammlung (`belege/*.sh` oder Datei für `hurl`); Seed-Daten mit zwei Mandanten und beiden Rollen.

5 Bewertungskriterien

- **Vollständigkeit** – alle Fehlerklassen gefunden oder nur die auffällige Einschleusung? Auch die stillen wie Kontenaufzählung und CORS-Spiegelung?
- **Belege** – reproduzierbar ohne Rückfrage, und zeigt der Beleg die Auswirkung statt nur einen Statuscode?
- **Art der Behebung** – Ursache beseitigt (Positivlisten, Parameterbindung, Durchsetzung an der Datenzugriffsschicht) oder Symptom kaschiert (Zeichen herausfiltern, Felder nachträglich löschen)?
- **Keine Regression und Bericht** – läuft die API nach der Härtung vollständig, und ist `SICHERHEIT.md` für eine Kundin lesbar, die kein TypeScript liest?