ChatGPT

Sicherheits-Audit Bericht

Executive Summary

Der geprüfte Code des **BHB Invoice & Dunning Portal** weist mehrere schwerwiegende Sicherheitslücken und einige Verbesserungsmöglichkeiten in Bezug auf Code-Qualität auf. Insbesondere wurden **kritische Schwachstellen in der Zugriffskontrolle** identifiziert, die es einem authentifizierten Angreifer ermöglichen könnten, **unautorisierte Rechnungsdaten abzurufen**. Weiterhin gibt es **Konfigurationsschwächen** (z.B. ein fest codiertes Sitzungspasswort im Code) sowie fehlende Schutzmechanismen gegen gängige Angriffsarten wie **Brute-Force-Login-Versuche** und **Cross-Site Request Forgery (CSRF)**. Insgesamt wird das Sicherheitsniveau als **kritisch** eingestuft, da ein motivierter Angreifer mit wenig Aufwand sensible Daten kompromittieren oder erhöhte Rechte erlangen könnte. Positiv anzumerken ist, dass bewährte Verfahren wie Passwort-Hashing (bcrypt) und Input-Validierung (via Zod Schemas) bereits umgesetzt wurden. Die folgenden detaillierten Befunde erläutern die Schwachstellen und Code-Smells im Einzelnen, gefolgt von konkreten Verbesserungsvorschlägen.

Detaillierte Befunde

Schweregrad	Art	Ort (Datei:Zeile)	Beschreibung
Kritisch	Broken Access Control (IDOR)	`routes.ts` 1 2	**Unzureichende Zugriffskontrolle**: Authentifizierte Benutzer können über `/api/invoices/:id/pdf` **beliebige Rechnungs-PDFs** herunterladen, auch wenn sie nicht zu ihrem Konto gehören. Es fehlt eine Berechtigungsprüfung, ob die angeforderte Rechnung zum angemeldeten Benutzer gehört. Dies ermöglicht unautorisierte Einsicht in fremde Rechnungsdaten (Insecure Direct Object Reference).
Kritisch	Broken Access Control (IDOR)	`routes.ts` 3 4	**Unzureichende Zugriffskontrolle**: Ähnlich wie oben können Benutzer über `/api/customers/:id/statement-pdf` **Kontoauszüge anderer Kunden** abrufen. Die Route ist nur durch `isAuthenticated` geschützt, ohne zu prüfen, ob der angeforderte Kundenbericht dem aktuellen Benutzer gehört. Dadurch können vertrauliche Finanzdaten anderer Kunden offengelegt werden.

Schweregrad	Art	Ort (Datei:Zeile)	Beschreibung
Hoch	Unsicherer Standard (Default Credential)	`auth.ts` [5]	**Hardcoded Session Secret**: In der Session-Konfiguration wird ein Standard-Secret `"development-secret-change-in-production"` verwendet, falls die Umgebungsvariable `SESSION_SECRET` nicht gesetzt ist [5]. Sollte dieser Wert versehentlich in Produktion aktiv sein, wären **Session-Cookies vorhersagbar**, was Session Hijacking ermöglicht. Dieses fest codierte Secret stellt ein hohes Risiko dar, wenn es nicht überschrieben wird.
Hoch	Logikfehler (Race Condition)	`auth.ts` [6] [7]	**Erst-Admin Registrierung**: Die Registrierung erlaubt dem *allerersten* Benutzer, automatisch die Rolle **Admin** zu erhalten [8]. Ist die Anwendung öffentlich zugänglich, könnte ein Angreifer sich als Erster registrieren und Admin-Rechte erlangen. Zudem besteht ein Race-Condition-Risiko: Zwei gleichzeitige Registrierungen könnten beide als erster Admin durchgehen, da die Prüfung `hasAdmin` nicht atomar mit der Erstellung erfolgt [7]. Dies gefährdet die vorgesehenen Zugriffsrechte.
Mittel	Fehlender Schutz vor Brute-Force	`auth.ts` [9] [10]	**Keine Rate Limiting bei Login**: Die Login-Route (`/api/auth/login`) implementiert keine Begrenzung für Anmeldeversuche [9]. Ein Angreifer kann unbegrenzt Passwort-Versuche durchführen (Brute-Force/Credential Stuffing), da weder Captcha, noch Account-Sperren oder Verzögerungen bei wiederholten Fehlversuchen umgesetzt sind. Dies schwächt die Authentifizierungssicherheit.
Mittel	CSRF-Schutz fehlt	*Global*	**Kein CSRF-Schutz**: Trotz Verwendung von Cookies (Session-ID) fehlt ein CSRF-Token-Mechanismus. Zwar ist `sameSite` auf "lax" gesetzt [11], was CSRF-Risiken etwas mindert, jedoch können state-changing Requests (z.B. Logout, Einstellungen ändern) potentiell von fremden Sites ausgelöst werden, sofern der Benutzer auf einen präparierten Link klickt. Ein dedizierter CSRF-Schutz oder strikteres `sameSite=strict` wäre angebracht.

Schweregrad	Art	Ort (Datei:Zeile)	Beschreibung
Niedrig	Informationsleck (Logging)	`routes.ts` 12 13	**Sensitive Data Logging**: Debug-Endpunkte und PDF-Generierung loggen detaillierte Informationen, inkl. API-Schlüssel im Request-Body (`api_key`) und Endpunkt-URLs 13 . Diese Logs sind nur für Admins einsehbar, dennoch stellen sie ein Risiko dar, falls Log-Dateien kompromittiert werden. Geheimnisse sollten nicht im Klartext geloggt werden.
Info	Performance/ Code Smell	`storage.ts` 14 15	**Ineffiziente Datenabfrage**: Funktionen wie `getReceiptsForUser` laden in einer Schleife die Rechnungen für jeden Kunden einzeln 14 , statt einen JOIN/IN-Query zu nutzen. Dies ist kein direktes Sicherheitsproblem, kann aber bei vielen Kunden die Performance beeinträchtigen (möglicher DoS-Vektor durch hohe Last). Ebenso lädt `getCustomerByName` alle Kunden in den Speicher und filtert dann, was bei großem Datenbestand ineffizient ist.

Remediation (Lösungsvorschläge)

1. Zugriffskontrolle stärken (IDOR-Fix): Schützen Sie sensible Routen mit entsprechenden Berechtigungsprüfungen. Für die PDF-Download-Endpoints sollte verifiziert werden, ob die angeforderte Ressource zum angemeldeten Benutzer gehört. Zum Beispiel kann man für Rechnungs-PDFs entweder den Middleware-Guard `isInternal` erzwingen (falls nur interne Mitarbeiter Zugriff benötigen) **oder** eine Laufzeitprüfung implementieren. Ein sicheres Beispiel wäre, den Debitor der Rechnung mit den dem User zugeordneten Debitoren abzugleichen:

```
// Nur Admins oder zuständige Benutzer dürfen PDF abrufen
app.get("/api/invoices/:id/pdf", isAuthenticated, async (req, res) => {
  const invoice = await storage.getReceipt(req.params.id);
  if (!invoice) return res.status(404).json({ message: "Invoice not
found" });
  // Zugriff verweigern, wenn aktueller User nicht zugehörig
  if (req.session.role === "customer") {
    const userCustomers = await
storage.getCustomersForUser(req.session.userId!);
    const allowed = userCustomers.some(c => c.debtorPostingaccountNumber ===
invoice.debtorPostingaccountNumber);
    if (!allowed) {
      return res.status(403).json({ message: "Keine Berechtigung für diese
Rechnung" });
    }
```

```
  }
  // ... (PDF abrufen und zurücksenden)
});
```

Ähnlich sollte bei `/api/customers/:id/statement-pdf` verfahren werden, oder diese Routen auf interne Rollen beschränkt werden. Durch solche Prüfungen wird sichergestellt, dass **kein Benutzer auf fremde Daten** zugreifen kann.

2. Erst-Admin Registrierung absichern: Verhindern Sie, dass beliebige externe Nutzer den ersten Admin-Account erstellen. Mögliche Ansätze: - **Installationstoken:** Fordern Sie bei der ersten Registrierung einen speziellen einmaligen Setup-Code oder ein Admin-Passwort aus der Konfiguration, den nur berechtigte Personen kennen. Ohne diesen Code schlägt die Registrierung fehl. - **Transaktionale Prüfung:** Führen Sie die Prüfung auf vorhandene Admins und das Erstellen des neuen Users in einer **einzigen Datenbank-Transaktion** durch. So wird verhindert, dass zwei gleichzeitige Registrierungen beide keinen Admin vorfinden. Beispielsweise: `BEGIN; SELECT COUNT(*) ... FOR UPDATE; ... INSERT new user; COMMIT;`. Alternativ kann man die erste Registrierung komplett entfernen und stattdessen initiale Admin-Benutzer per SQL oder Konfiguration anlegen. - **Standard-Login deaktivieren:** Stellen Sie sicher, dass die Site nicht öffentlich erreichbar ist, bevor ein Admin existiert (z.B. durch einen Maintenance-Modus). Nach dem ersten Admin-Setup kann man die Registrierung wie vorgesehen sperren.

Diese Maßnahmen verhindern, dass sich unautorisierte Personen Admin-Rechte erschleichen.

3. Geheimnisse sicher handhaben: Entfernen Sie Hardcoded-Secrets aus dem Quellcode. Für Entwicklungsumgebungen kann man einen Warnhinweis oder einen generierten Wert nutzen, aber in Produktion **muss** `SESSION_SECRET` **gesetzt** sein. Ein sicherer Fix im Code könnte so aussehen:

```
session({
  // Keine feste Fallback-Phrase mehr nutzen
  secret: process.env.SESSION_SECRET || (() => {
    if (process.env.NODE_ENV !== "production") {
      console.warn("Using a temporary dev session secret");
      return require("crypto").randomBytes(32).toString("hex");
    }
    throw new Error("SESSION_SECRET is not set!");
  })(),
  // ... weitere Optionen ...
});
```

Hier wird in Produktion ein fehlendes Secret durch einen Fehler angeprangert (Abbruch), statt ein unsicheres Default zu verwenden. In Entwicklung wird zwar ein zufälliges Secret erzeugt, aber mit Hinweis im Log – so wird das Risiko minimiert, versehentlich mit dem Default-Secret zu arbeiten. Zusätzlich sollten keine sensiblen Konstanten (API-Schlüssel, Passwörter) im Repo-Text auftauchen, sondern immer in Umgebungsvariablen oder Secrets-Management liegen.

4. Bruteforce-Schutz einbauen: Implementieren Sie Mechanismen, um automatisierte Passwort-Angriffe zu erschweren: - **Rate Limiting:** Begrenzen Sie z.B. mittels Express-Middleware die Anzahl von Login-Versuchen pro IP/Account auf z.B. 5 pro 15 Minuten. Bei Überschreitung kann eine temporäre Sperre oder Captcha-Anforderung erfolgen. - **Account Lockout:** Sperren Sie einen Account nach X

fehlgeschlagenen Login-Versuchen zeitweilig und informieren den Benutzer oder Admin darüber. Dies verhindert das Ausprobieren vieler Passwörter. - **Logging & Monitoring:** Protokollieren Sie Fehlversuche und implementieren Sie ggf. Alerts, um Brute-Force-Versuche frühzeitig zu erkennen.

Durch diese Maßnahmen erhöhen Sie die Hürden für Angreifer erheblich, ohne die Benutzerfreundlichkeit stark zu beeinträchtigen.

5. CSRF-Schutz ergänzen: Obwohl `sameSite=lax` gesetzt ist, sollte für kritische POST/PUT/DELETE-Aktionen ein expliziter CSRF-Schutz implementiert werden: - **CSRF-Token:** Integrieren Sie ein eindeutiges Token, das der Client bei jeder state-changing Anfrage mitsendet (z.B. als Header oder verstecktes Formularfeld). Der Server prüft dessen Gültigkeit pro Session. Libraries wie `csurf` (für Express) können hier unterstützen. - **SameSite=Strict:** Ziehen Sie in Betracht, das Session-Cookie in Zukunft auf `SameSite=Strict` zu setzen, sofern die App keine berechtigten Cross-Site Anfragen benötigt. Dies unterbindet das Senden des Cookies bei sämtlichen Cross-Origin Requests, inklusive Navigationsaufrufen. - **CORS-Konfiguration:** Stellen Sie sicher, dass die API **nicht** per permissiven CORS-Headers für fremde Origins geöffnet ist (standardmäßig ist Express zwar restriktiv, aber explizite Einstellungen sollten geprüft werden). So kann kein externer Domain-Skript legitime API-Aufrufe im Namen des Nutzers absetzen.

Mit diesen Änderungen wird das Risiko von CSRF-Angriffen deutlich reduziert.

6. Logging & Error-Handling verbessern: Entfernen oder entschärfen Sie das Logging sensibler Daten: - Verwenden Sie für Debug-Zwecke Umgebungsflags (z.B. `DEBUG=true`), um ausführliche Logs nur in Entwicklung zu aktivieren. In Produktivumgebungen sollten API-Schlüssel oder Passwörter **nie in Logs erscheinen**. Zum Beispiel könnte man den geloggten Request-Body filtern (`{ ... requestBody, api_key: "***" }`), bevor er ausgegeben wird. - Fehlerberichte an den Client sollten allgemein gehalten sein. Der gegebene Code macht das bereits weitgehend richtig (generische Fehlermeldungen wie "Anmeldung fehlgeschlagen" [16] anstatt stack traces). Stellen Sie sicher, dass dies konsistent geschieht und keine Debug-Informationen versehentlich dem Endnutzer ausgegeben werden. - Implementieren Sie ggf. eine zentrale **Überwachung** für Fehler (z.B. Sentry), um sicherzustellen, dass sicherheitsrelevante Fehlerfälle erkannt und behoben werden können, ohne dem Angreifer Hinweise zu liefern.

7. Code-Qualität & Performance: Beheben Sie identifizierte Code Smells, um die Wartbarkeit zu verbessern und eventuelle DoS-Angriffsflächen zu verkleinern: - **Datenbankzugriffe optimieren:** Nutzen Sie set-basierte Abfragen statt Schleifen. Im Beispiel `getReceiptsForUser` könnte man z.B. mit **einer** Query alle relevanten Belege laden (WHERE debtorPostingaccountNumber IN (...)) anstatt N Einzelabfragen [14]. Dies reduziert Last und Komplexität. Gleiches gilt für `getCustomerByName`, wo eine LIKE-Query oder Volltextsuche in der DB die aktuelle vollständige Speicherung aller Datensätze im Speicher ersetzt. - **Asynchrone Aufrufe bündeln:** Wenn möglich, parallelisieren Sie unabhängige Aufrufe oder verwenden Sie Batch-Abfragen. Im Dashboard-Load werden z.B. alle Kunden und alle Regeln geladen [17], was okay ist – dennoch sollte man bei wachsendem Datenumfang überlegen, ob Pagination oder gezieltes Laden nötig ist. - **Security-Header:** Erwägen Sie den Einsatz von Middleware wie Helmet, um Standard-Sicherheitsheader zu setzen (Content Security Policy, XSS-Protection, etc.). Dies verbessert zwar hauptsächlich die Frontend-Sicherheit, rundet aber das Sicherheitsprofil der Anwendung ab.

Durch diese Refactorings wird der Code robuster, effizienter und sicherer gegen sowohl gezielte Angriffe als auch unabsichtliche Fehler. Insgesamt sollten die vorgeschlagenen Maßnahmen das Portal auf ein **höheres Sicherheitsniveau** heben und das Risiko erfolgreicher Angriffe erheblich reduzieren.

1 2 3 4 12 13 17 routes.ts

https://github.com/hi-its-lukas/bhb-invoice-portal/blob/9e0d99a6d1693567a6ff86da259f36c1d92a9835/server/routes.ts

5 6 7 8 9 10 11 16 auth.ts

https://github.com/hi-its-lukas/bhb-invoice-portal/blob/9e0d99a6d1693567a6ff86da259f36c1d92a9835/server/auth.ts

14 15 storage.ts

https://github.com/hi-its-lukas/bhb-invoice-portal/blob/9e0d99a6d1693567a6ff86da259f36c1d92a9835/server/storage.ts