Entwicklung einer unsicheren API

Belegarbeit im Rahmen des Moduls “Softwaretechnik-Projekt IT-Sicherheit” des B.Sc. Studiengangs Angewandte Informatik IT-Sicherheit an der Hochschule Mittweida

Vorgelegt von
Noah Hiller [59057], Franz Sander [58739], Hannes Wilhelm Lange [58744], Michael Primke [59102], Jonas Germann [58740], Jon Lee Römmling [58931], Jakob Hindemith[58742]

Betreuer: M.Sc. Philipp Engler

5.Semester
Abgabedatum: 28.02.2025

Inhaltsverzeichnis

1. Team- und Aufgabenvorstellung

Unser Team, bestehend aus Studierenden des Bachelor-Studiengangs Angewandte Informatik mit einer Vertiefung in IT-Sicherheit, entwickelte im Rahmen des Moduls Softwaretechnik-IT-Sicherheit eine bewusst unsichere Webanwendung. Ziel war es, Sicherheitslücken in die API einzubauen, um sie später zu identifizieren, zu dokumentieren und mögliche Schutzmaßnahmen zu erarbeiten. Diese unsichere API soll zudem als Lernplattform verwendet werden, um Studierenden oder angehenden Sicherheitsexperten die Möglichkeit zu geben, realistische Schwachstellen in Webanwendungen zu untersuchen und Abwehrmechanismen zu entwickeln. Um eine effiziente Umsetzung des Projekts sicherzustellen, haben wir die Arbeit in folgende Schwerpunkte aufgeteilt:

Hannes L. , Franz S. , Jakob H. , Jon R .	Programmierung API & DB
Michael P. , Jonas G.	Programmierung Weboberfläche
Noah H.	Dokumentation
Franz S. , Jon R.	Beleg

Unser Team hat sich grundsätzlich durch regelmäßige Meetings und eine klare Aufgabenverteilung koordiniert und somit hat jeder Beitrag zur erfolgreichen Umsetzung der unsicheren API, zur Erstellung einer funktionsfähigen Webanwendung und der entsprechenden Dokumentation beigetragen.

2. Motivation

Die Sicherheit von Webanwendungen zählt grundsätzlich zu den größten Herausforderungen in der modernen Softwareentwicklung. Mit der zunehmenden Digitalisierung und der Verlagerung vieler Geschäftsprozesse ins Internet wird die Entwicklung sicherer Anwendungen nicht nur zu einer technischen, sondern auch zu einer gesellschaftlichen Verantwortung. Dennoch zeigen Berichte über Cyberangriffe immer wieder, dass viele Webanwendungen grundlegende Schwachstellen aufweisen. Zu den häufigsten Angriffstechniken gehören Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), schwache Authentifizierung und somit insbesondere unsichere API-Implementierungen. Solche Schwachstellen können schwerwiegende Folgen haben, darunter finanzielle Schäden oder ein irreparabler Imageverlust für Unternehmen, sowie ein enormer Datenverlust. Um diese Risiken nahbarer zu machen und ein tieferes Verständnis für Sicherheitsmaßnahmen zu fördern, wurde dieses Projekt umgesetzt.

Das Ziel des Projekts besteht darin, eine bewusst unsichere Webanwendung zu entwickeln, um typische Schwachstellen zu demonstrieren und deren Auswirkungen zu veranschaulichen. Darüber hinaus soll die Anwendung als Lehrbeispiel dienen, um Entwickler für die Wichtigkeit sicherer Software-Praktiken zu sensibilisieren. Eine solche Plattform soll es ermöglichen, Schwachstellen wie unverschlüsselte Passwörter oder Cross-Site Request Forgery (CSRF) in der Praxis zu untersuchen, die Mechanismen hinter diesen Angriffen zu verstehen und schließlich wirksame Gegenmaßnahmen zu entwickeln.

Ein besonderer Schwerpunkt des Projekts liegt, wie bereits erwähnt, auf der Implementierung einer unsicheren API. APIs sind das Rückgrat moderner Webanwendungen und ermöglichen die Kommunikation zwischen Frontend und Backend. Sie sind jedoch aufgrund ihrer oft öffentlichen Zugänglichkeit ein häufiges Ziel für Angriffe. Selbst kleine Fehler in der Implementierung können schwerwiegende Konsequenzen haben, da sie einem Angreifer den Zugriff auf sensible Daten oder geschützte Funktionen ermöglichen. Durch die bewusste Implementierung von Schwachstellen kann gezeigt werden, wie leicht solche Fehler

ausgenutzt werden können und welche Auswirkungen sie auf die Sicherheit einer Anwendung haben.

Das vorliegende Szenario eines Online-Shops wurde bewusst gewählt, da es alltäglich und leicht verständlich ist. In einem solchen Kontext bietet sich eine Vielzahl von Anwendungsfällen, die potenzielle Angriffspunkte aufzeigen. So sind Funktionen wie die Registrierung und der Login von Benutzern, die Anzeige von Produkten und deren Bewertung sowie das Verwalten eines Warenkorbs und das Tätigen von Bestellungen typische Bestandteile eines Online-Shops. Durch die Nachbildung eines solchen Systems werden die Schwachstellen in einem realitätsnahen Umfeld präsentiert, was den Lerneffekt verstärkt.

Langfristig geht dieses Projekt somit über die reine Implementierung einer unsicheren API hinaus. Es bietet eine Grundlage für Schulungen, welche die Entwickler für die Wichtigkeit von Sicherheitsmaßnahmen sensibilisieren sollen. Darüber hinaus kann die Anwendung als Plattform genutzt werden, um Angriffe zu analysieren, neue Sicherheitsmaßnahmen zu testen und Best Practices für die API-Entwicklung zu fördern.

3. Grundlagen

Die Entwicklung der unsicheren Webanwendung erforderte ein solides Verständnis moderner Technologien und ein gezieltes Einbauen typischer Schwachstellen. Dieses Kapitel beschreibt die theoretischen und technischen Grundlagen, die für das Projekt relevant sind. Es erläutert die eingesetzten Technologien, deren Funktionen und die bewussten Sicherheitslücken, die in die Anwendung integriert wurden.

Die Architektur der Anwendung folgt einer typischen Drei-Schichten-Struktur, die aus einer Datenbank, einem Backend und einem Frontend besteht. Diese Trennung ermöglicht eine klare Verantwortungsverteilung und verbessert die Wartbarkeit des Systems.

Die PostgreSQL-Datenbank bildet die Grundlage der Datenhaltung und speichert alle wichtigen Informationen wie Benutzerdaten, Produkte, Bewertungen und Bestellungen. Diese Daten sind über Beziehungen miteinander verknüpft, was es ermöglicht, komplexe Funktionen wie die Anzeige von Produktbewertungen oder das Verwalten von Bestellungen abzubilden.

Das Backend der Anwendung wurde mit Node.js und dem darauf aufbauenden Framework Express.js implementiert. Node.js bietet als serverseitige JavaScript Laufzeitumgebung eine hohe Flexibilität und eignet sich besonders gut für die Entwicklung von APIs. Express.js erweitert Node.js um Funktionen, die die Entwicklung von Webanwendungen und APIs vereinfachen. Das Backend dient als Vermittler zwischen der Datenbank und dem Frontend.

Es stellt eine REST-API bereit, die verschiedene CRUD-Endpunkte (Create, Read, Update, Delete) enthält. Diese Endpunkte ermöglichen es, Benutzerdaten, Produkte, Bewertungen und Bestellungen zu verwalten.

Dabei wurde bewusst auf Sicherheitsmaßnahmen verzichtet, um typische Schwachstellen realitätsnah demonstrieren zu können. So wurden beispielsweise keine Zugriffsbeschränkungen für bestimmte API-Endpunkte definiert und keine serverseitige Validierung von Eingaben implementiert, wodurch manipulierte oder

ungültige Daten in die Datenbank gelangen können. Werden diese ungefiltert im Frontend ausgegeben, kann dies zu Cross-Site-Scripting (XSS)-Angriffen führen.

Zwar werden zur Authentifizierung in der Anwendung auch JSON Web Token (JWT) eingesetzt, allerdings weisen diese einige Schwachstellen auf, da die Token-Signatur mit dem statischen Wert „secure“ erfolgt. Dadurch kann jeder, der diesen Wert kennt, ein gültiges Token generieren und sich als beliebiger Benutzer ausgeben. Zudem haben die JWTs kein Ablaufdatum, sodass sie nach einem einmaligen Diebstahl oder Leak dauerhaft gültig bleiben. Dies erhöht das Risiko von Session-Hijacking, da ein Angreifer ein einmal angefangenes Token unbegrenzt nutzen kann. Darüber hinaus fehlt ein Mechanismus zur Token-Invalidierung, was bedeutet, dass sich Nutzer nach einer Abmeldung oder einem Passwortwechsel nicht effektiv ausloggen können, da ältere Tokens weiterhin gültig bleiben.

Ein weiteres Sicherheitsrisiko stellt die Speicherung von Passwörtern im Klartext dar, ohne dass ein Hashing-Algorithmus wie bcrypt verwendet wird. Dadurch besteht bei einem Datenleck ein erhebliches Risiko, dass Passwörter kompromittiert werden.

Ein weiteres Risiko stellt das Fehlen eines Schutzes gegen Cross-Site Request Forgery (CSRF) dar. Ein Angreifer könnte ein eingeloggtes Opfer dazu bringen, ohne dessen Wissen eine schadhafte Aktion auszuführen, da die API keine Mechanismen zur Herkunftsprüfung implementiert hat wie beispielsweise die Verwendung von CSRF-Tokens in Formularen oder die Überprüfung des Origin- und Referer-Headers bei sicherheitskritischen Anfragen.

Während viele gängige Angriffsvektoren bewusst eingebaut wurden, sind SQL-Injections nicht möglich, da alle Datenbankzugriffe über das PostgreSQL-Modul *postgres.js* erfolgen, welches parametrisierte Abfragen nutzt und somit das Einspeisen von schädlichem SQL-Code verhindert.

Das Frontend wurde mit PHP entwickelt, um die Benutzerschnittstelle zu realisieren und die API des Backends anzusprechen. Für die Gestaltung der Oberfläche kam das CSS-Framework Bootstrap zum Einsatz. Die Kommunikation zwischen Frontend und Backend erfolgt über CURL-Aufrufe, bei denen JSON-Daten gesendet oder empfangen werden. Diese Architektur ermöglicht eine einfache Darstellung der

Daten im Frontend und erlaubt es Benutzern, Aktionen wie die Registrierung, den Login oder das Bewerten von Produkten durchzuführen.

Zusätzlich zu den bewussten Schwachstellen im Backend enthält die Anwendung auch Sicherheitslücken im Frontend. Die Benutzersitzung wird mithilfe der PHP-Session-Funktionalität verwaltet, ohne zusätzliche Schutzmaßnahmen wie die Bindung der Session an die IP-Adresse des Benutzers oder eine Ablaufzeit. Dies erhöht die Wahrscheinlichkeit von Angriffen wie Session-Hijacking.

Darüber hinaus wurde im Frontend bewusst auf clientseitige Validierungen verzichtet, wodurch Angreifer in der Lage sind, ungültige oder schädliche Daten direkt an die API zu senden, ohne dass das Frontend dies überprüft.

Zudem gibt es keinen Schutz gegen Cross-Site-Scripting (XSS), da Benutzereingaben nicht gefiltert werden. Dies ermöglicht es einem Angreifer, JavaScript-Code in Formulare einzuschleusen, der dann im Browser anderer Benutzer ausgeführt wird, um beispielsweise Cookies zu stehlen oder Schadcode nachzuladen.

Des Weiteren gibt es auch keine Schutzmechanismen gegen automatisierte Angriffe, da kein Rate-Limiting oder CAPTCHA implementiert wurde. Dadurch sind Brute-Force-Attacken auf Login-Formulare oder Spam-Anfragen an die API problemlos durchführbar.

4. Umsetzung

Die technische Umsetzung der Anwendung erfolgte in mehreren Schritten, beginnend mit der Modellierung der Datenbank, der Entwicklung des Backends und der Gestaltung des Frontends. Der gesamte Code des Projekts wird in einem GitHub-Repository verwaltet, wodurch Versionierung, sowie Kollaboration ermöglicht werden.

4.1 Containerisierung mit Docker

Um eine konsistente Entwicklungsumgebung sicherzustellen, wird die gesamte Anwendung in Docker-Containern ausgeführt. Dies ermöglicht eine einfache Bereitstellung, vermeidet Abweichungen zwischen verschiedenen Systemkonfigurationen und stellt sicher, dass alle Komponenten nahtlos zusammenarbeiten. Die Docker-Compose-Konfigurationsdatei *docker-compose.yml* definiert drei zentrale Dienste der Anwendung.

Der PostgreSQL-Container mit dem Service-Namen *db*, speichert alle persistenten Daten der Anwendung, darunter Benutzer, Produkte, Bestellungen sowie Bewertungen. Die Daten werden auf Port 5432 bereitgestellt.

Ein Node.js/Express-Container mit dem Service-Namen *api stellt* die REST-API bereit und kommuniziert mit der PostgreSQL-Datenbank. Der API-Container ist von der Datenbank abhängig, um sicherzustellen, dass die Datenbank vor dem Start des Backends verfügbar ist. Alle API-Endpunkte werden über Port 8080 bereitgestellt, sodass das Frontend darauf zugreifen kann.

Zudem existiert auch ein Container für den PHP-Webserver mit dem Service-Namen *frontend*, der die Benutzeroberfläche bereitstellt. Die Kommunikation mit der API erfolgt über HTTP-Anfragen. Der Webserver ist über Port 80 erreichbar.

Die Docker-Compose-Datei sorgt somit dafür, dass alle Dienste mit dem Befehl *docker-compose up* gestartet und automatisch miteinander vernetzt werden.

4.2 Datenbankdesign und Implementierung

Die PostgreSQL-Datenbank bildet die Grundlage der gesamten Anwendung und wurde so gestaltet, dass sie die wesentlichen Funktionen eines Online-Shops unterstützt. Die Datenbankstruktur wurde durch das SQL-Initialisierungsskript *sql.init* definiert, welches beim Start der Anwendung automatisch ausgeführt wird, um die Tabellen und Daten zu erstellen. Das Initialisierungsskript wird über die Docker-Container zur Verfügung gestellt, womit gesichert ist, dass die Anwendung jederzeit in einem fehlerfreien Zustand ist.

Die Datenbank besteht aus mehreren Tabellen, die die verschiedenen Entitäten und deren Beziehungen modellieren. Die Tabelle *users* speichert die Benutzerdaten, darunter Benutzernamen, E-Mail-Adressen und Passwörter. Die Passwörter werden grundsätzlich im Klartext gespeichert, ohne dass ein Hashing-Algorithmus verwendet wird, um gezielt eine Sicherheitslücke zu implementieren.

Die Tabelle *products* verwaltet die Informationen zu den Produkten, einschließlich Name, Preis, Lagerbestand und optionalen Rabatten. Für jedes Produkt kann auch ein Thumbnail-Bild gespeichert werden, was die Darstellung der Produkte im Frontend ermöglicht.

Bewertungen werden in der Tabelle *reviews* gespeichert. Sie sind sowohl mit Benutzern als auch mit Produkten verknüpft. Hier wurde bewusst auf Validierungen der Bewertungsskala oder der Bewertungslänge verzichtet, sodass potenziell manipulierte Bewertungen gespeichert werden können.

Bestellungen und die dazugehörigen Produkte werden in den Tabellen *orders* und *order_items* gespeichert. Eine Bestellung wird in der Tabelle *orders* abgelegt, während die spezifischen Produkte, die in der Bestellung enthalten sind, in *order_items* gespeichert werden. Die Tabellen sind miteinander durch eine 1:n-Beziehung verknüpft, wobei jede Bestellung mehrere Produkte umfassen kann. Diese klare Struktur ermöglicht es, die Beziehungen zwischen Benutzern, Produkten, Bewertungen und Bestellungen effizient abzubilden.

Es wurde bewusst darauf verzichtet, Datenvalidierungen oder Constraints zu integrieren. Dies hat zur Folge, dass manipulierte oder ungültige Daten in die

Datenbank gelangen können. Beispielsweise können negative Bestellmengen oder ungültige Bewertungen gespeichert werden, da keine Prüfungen auf logische Werte oder Eingabe Beschränkungen vorgenommen werden.

4.3 Backend-Implementierung

Das Backend der Anwendung wurde mit Node.js und dem Framework Express.js umgesetzt, um die REST-API bereitzustellen. Die API umfasst Endpunkte für die Verwaltung von Benutzerdaten, Produkten, Bewertungen und Bestellungen. Jeder dieser Endpunkte stellt eine potenzielle Sicherheitslücke dar, die in diesem Abschnitt beleuchtet und die bewussten Schwächen erläutert werden.

In der Benutzerverwaltung wurden die Routen */users/register* und */users/login* implementiert, um die Registrierung und Anmeldung von Benutzern zu ermöglichen. Bei der Registrierung werden Passwörter im Klartext in der Datenbank gespeichert und beim Login direkt mit dem eingegebenen Passwort verglichen. Diese bewusste Unsicherheit soll die Gefahren ungesicherter Passwortspeicherung aufzeigen. Passwörter sollten mit einem sicheren Algorithmus wie bcrypt oder Argon2 gehasht werden, um sie vor einem möglichen Diebstahl zu schützen. Diese Hashes sollten dann in der Datenbank gespeichert werden, anstatt der Klartext-Passwörter. Beim Login sollten somit auch die Hashes verglichen werden, nicht das Klartext-Passwort.

Zur Authentifizierung wurde JSON Web Token (JWT) verwendet, jedoch ohne ein Ablaufdatum oder zusätzliche Sicherheitsmaßnahmen zu implementieren, wodurch gestohlene Tokens unbegrenzt gültig bleiben. Jeder JWT sollte mit einer Ablaufzeit versehen werden, um das Risiko eines Token-Diebstahls zu minimieren. Zudem sollte ein Mechanismus, der abgelaufene Tokens durch Refresh Tokens ersetzt, implementiert werden. Die JWTs werden zudem nur mit einem statischen Secret "*secure*" signiert. Ein schwaches oder öffentlich bekanntes Secret wie "*secure*" ermöglicht es einem Angreifer, ein gültiges Token zu generieren, indem er die Signatur manipuliert und dieselbe erzeugt, ohne den echten Secret-Schlüssel zu kennen. Ein gutes Secret für die Signierung eines JWTs sollte zufällig und lang sein, um es nahezu unmöglich zu machen, es zu erraten oder zu berechnen. In diesem Fall ist es allerdings extrem anfällig für Brute-Force- oder Wörterbuchangriffe.

Die Produktverwaltung umfasst Endpunkte wie */products*, die eine Liste aller Produkte zurückgeben, und */products/:id*, die die Details eines spezifischen Produkts abrufen. Die Route */products/search/:name* erlaubt die Suche nach Produkten anhand eines Namensmusters. Die Benutzereingabe kann sicher in die SQL-Abfrage eingebunden werden, indem postgres.js verwendet wird, das parametrisierte Abfragen unterstützt. Das bedeutet, dass die Benutzereingaben nicht direkt in den SQL-Code eingefügt werden, sondern als separate Parameter übergeben werden. Diese Methode schützt die Anwendung vor SQL-Injection-Angriffen, da schadhafter SQL-Code nicht als Teil der Abfrage interpretiert wird.

Die Verwaltung von Bewertungen erfolgt über die Endpunkte */reviews* und */reviews/product/:productId*. Benutzer können Bewertungen für Produkte abgeben, wobei die Authentifizierung über JWT erfolgt. Durch das Fehlen eines Ablaufdatums der Tokens können potenzielle Angreifer auf die Endpunkte zugreifen, sofern sie ein gültiges Token besitzen. Bei der Implementierung wurde auf Validierungen wie die Prüfung der Bewertungsskala oder die Länge der Bewertung verzichtet, was das System anfällig für manipulierte Daten macht.

Das Bestellsystem ermöglicht es Benutzern, Bestellungen zu erstellen und bestehende Bestellungen einzusehen. Die Route */order* erlaubt das Aufgeben neuer Bestellungen, während */orders* alle Bestellungen eines Benutzers anzeigen. Hierfür wird die *user_id* aus dem JWT extrahiert, um die richtigen Bestellungen des Benutzers abzurufen. Alle zugehörigen Bestellpositionen werden ebenfalls aus der order_items-Tabelle geladen und dem Benutzer angezeigt. Über die Route */orders/:id* können Benutzer die Details einer bestimmten Bestellung einsehen. Die Bestellung wird mit der order_id aus der URL abgefragt, und es wird überprüft, ob die Bestellung dem aktuell angemeldeten Benutzer gehört.

Bei der Erstellung einer Bestellung wird grundsätzlich keine Überprüfung der Bestellmenge oder Lagerbestände vorgenommen. Ein Angreifer könnte daher manipulierte Bestellungen mit ungültigen Mengen einreichen. Es müsste eine Validierung stattfinden, so dass man beispielsweise sicherstellt, dass die Bestellmenge nicht negativ ist und dass genügend Produkte im Lager verfügbar sind, bevor die Bestellung abgeschlossen wird.

4.4 Frontend-Implementierung

Das Frontend wurde mit PHP und Bootstrap entwickelt und bietet eine intuitive Benutzeroberfläche, die alle zentralen Funktionen eines Online-Shops abbildet.

Die Login-Seite ermöglicht es den Nutzern, sich mithilfe ihres Benutzernamens und Passworts anzumelden. Die eingegebenen Daten werden an die API gesendet, und bei erfolgreicher Authentifizierung wird ein JWT zurückgegeben, das im Browser als Cookie gespeichert wird. Dieses Cookie wird für weitere Anfragen verwendet, jedoch ohne zusätzliche Sicherheitsmaßnahmen wie HTTPOnly oder Secure-Flags, was die Gefahr für Cookie-Diebstahl erhöht. Um die Sicherheit zu erhöhen, sollte das JWT im HTTPOnly-Flag gespeichert werden, sodass es nur über den Backend-Server zugänglich ist, und das Secure-Flag sollte gesetzt werden, damit das Cookie nur über eine gesicherte HTTPS-Verbindung übertragen wird.

Die Registrierungsseite bietet den Benutzern die Möglichkeit, ein neues Konto zu erstellen. Dabei werden keine Sicherheitsüberprüfungen wie Prüfung der Passwortstärke oder Validieren der E-Mail-Adresse durchgeführt. Nach der Registrierung wird das Konto direkt aktiv, ohne dass zusätzliche Maßnahmen wie eine E-Mail-Bestätigung erforderlich sind.

Die Hauptseite des Online-Shops zeigt die verfügbaren Produkte an, die über die API abgerufen werden. Die Produkte können durch Suchbegriffe und Filteroptionen durchsucht werden. Benutzer können auf ein Produkt klicken, um detaillierte Informationen zu sehen und Bewertungen zu hinterlassen. Die Benutzereingaben in der Suchleiste und Filteroptionen werden ohne Validierung oder Escaping verarbeitet, wodurch XSS-Angriffe möglich sind. Ein Angreifer könnte schadhafter JavaScript-Code in die Suchfelder eingeben, der dann im Browser eines anderen Benutzers ausgeführt wird, um beispielsweise Cookies zu stehlen oder bösartigen Code nachzuladen. Um diesen Sicherheitsaspekt zu lösen, müssten alle Benutzereingaben validiert und escaped werden, bevor sie in das DOM eingefügt werden.

Der Warenkorb zeigt die vom Benutzer ausgewählten Produkte an, die in einem Cookie gespeichert werden. Dieses Cookie enthält die Produkt-IDs der Artikel, die der Benutzer in seinen Warenkorb gelegt hat. Der Benutzer kann zudem die Anzahl

der Produkte anpassen und Artikel aus dem Warenkorb entfernen. Da das Cookie allerdings wie bereits erwähnt Schwachstellen aufweist, sollte entweder das Cookie sicherer gesetzt oder der Warenkorb auf der Serverseite gespeichert werden.

Über die Checkout-Seite hat der Benutzer die Möglichkeit, seine Bestellinformationen wie Name, Adresse, Stadt und PLZ einzugeben. Zwar wird überprüft, ob alle Felder ausgefüllt sind und ob die PLZ eine gültige Länge hat, jedoch fehlt eine umfassende Validierung der Eingaben, um Manipulationen zu verhindern. Nach der Eingabe der Bestelldaten kann der Benutzer dann die Bestellung abschließen. Dabei werden die Produkt-IDs und die dazugehörige Menge an die Backend-API gesendet, die die Bestellung verarbeitet, den Lagerbestand aktualisiert und die Bestellung in der Datenbank speichert. Jedoch fehlt hier ein Schutzmechanismus gegen Cross-Site Request Forgery (CSRF)-Angriffe. Ohne einen solchen Schutz könnte ein Angreifer einen authentifizierten Benutzer auf eine bösartige Webseite locken, die dann im Namen des Benutzers eine Bestellung absendet, ohne dass der Benutzer dies merkt.

5. Zusammenfassung

Im Rahmen dieses Projekts wurde somit eine bewusst unsichere Webanwendung entwickelt, die als Online-Shop dient und typische Sicherheitslücken in modernen Webanwendungen realitätsnah demonstriert. Die Architektur folgt einer klassischen Drei-Schichten-Struktur mit einer PostgreSQL-Datenbank, einem Node.js/Express.js-Backend und einem PHP-basierten Frontend. Alle Teile der Anwendung wurden in Docker-Containern ausgeführt, um eine konsistente Entwicklungsumgebung zu gewährleisten, und über Swagger dokumentiert, um eine transparente API-Dokumentation zu bieten.

Ein zentraler Fokus lag auf der sicheren Speicherung und Verarbeitung von Benutzerdaten. Passwörter wurden absichtlich im Klartext gespeichert, um das Risiko ungesicherter Passwortspeicherung aufzuzeigen. Zudem wurde JSON Web Token (JWT) für die Authentifizierung verwendet, jedoch ohne Ablaufdatum und mit einem unsicheren statischen Secret ("secure"), was die Anwendung anfällig für Token-Diebstahl macht.

Die bewusste Implementierung von weiteren Schwachstellen machte die Anwendung zudem anfällig für Angriffe wie Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) und Session-Hijacking. Eingaben von Benutzern wurden auch im Frontend ohne Validierung oder Escaping verarbeitet, was XSS ermöglichte. Zusätzlich fehlte der Schutz vor CSRF, was das Risiko von Manipulationen durch Angreifer erhöhte. Darüber hinaus wurden in der Benutzer-Sitzungsverwaltung keine Maßnahmen wie die Bindung an IP-Adressen implementiert, wodurch Angreifer Sessions übernehmen könnten.

Zusammenfassend verdeutlicht die umgesetzte Webanwendung, welche Risiken durch Sicherheitslücken in verschiedenen Schichten einer Anwendung entstehen. Durch die realitätsnahe Implementierung eines Online-Shops wird die Anfälligkeit unsicherer APIs und die Bedeutung sicherer Softwareentwicklung in einem praktischen Kontext eindringlich demonstriert.

6. Ausblick und Erweiterungsmöglichkeiten

Obwohl die Anwendung ihren Zweck erfüllt, typische Schwachstellen zu demonstrieren, gibt es zahlreiche Möglichkeiten, das Projekt weiterzuentwickeln und um zusätzliche Funktionalitäten zu erweitern.

Eine wichtige Weiterentwicklung könnte die Erstellung einer Mustervorlage sein, die Studierenden als Grundlage dient, um eigene Schutzmaßnahmen zu entwickeln und ihre Lösungen mit einer bereits etablierten und dokumentierten Vorlage zu vergleichen.

Die Mustervorlage würde eine strukturierte Anleitung zur Implementierung grundlegender Sicherheitsmaßnahmen bieten, wobei Studierende eigene Lösungen zur Verbesserung der Anwendung entwerfen könnten. Sie könnten zum Beispiel eigenständig den sicheren Umgang mit Benutzerdaten oder die Implementierung von Schutzmechanismen gegen Angriffe wie XSS oder CSRF entwickeln und dabei auf die Mustervorlage zurückgreifen, um die Best Practices und die Wirksamkeit ihrer Lösungen zu überprüfen und weiter zu optimieren.

Neben der Verbesserung der Sicherheit könnten auch funktionale Erweiterungen vorgenommen werden. Eine mögliche Erweiterung wäre die Einführung eines Admin-Panels, das es Administratoren ermöglicht, die Benutzerverwaltung, Produktkataloge, Bestellungen und Bewertungen effizient zu überwachen und zu verwalten. Diese Admin-Oberfläche könnte es den Administratoren ermöglichen, Produkte hinzuzufügen, zu bearbeiten oder zu löschen sowie Bestellungen nachzuverfolgen und ggf. zu bearbeiten. Dabei können auch gezielt Sicherheitsrisiken eingebaut werden, wofür daraufhin zusätzliche Sicherheitsvorkehrungen getroffen werden müssen , um Missbrauch zu verhindern. Wenn das Admin-Panel nicht sicher implementiert wird, könnte ein Angreifer zum Beispiel mit Administratorrechten Zugriff auf sensible Benutzerdaten oder Finanzinformationen erhalten.

Zusätzlich würde die Implementierung von Benutzerrollen und Berechtigungen eine sinnvolle Erweiterung darstellen. So würden unterschiedliche Benutzerrollen, wie Admin oder Moderator, eingeführt werden, die verschiedene Zugriffsrechte auf die Anwendung erhalten. Das würde den administrativen Zugriff sicherer und flexibler gestalten und gleichzeitig den Betrieb des Systems skalierbarer machen. Doch auch hier kann es zu Sicherheitslücken kommen, wenn die Implementierung fehlerhaft ist. Eine unsichere Verwaltung der Rollen und Berechtigungen kann dazu führen, dass unberechtigte Benutzer zu Administratorrechten gelangen, etwa durch eine fehlerhafte Rollenverwaltung oder fehlende Prüfung der Berechtigungen auf API-Ebene. Es ergäben sich somit weitere Sicherheitslücken, welcher einer Implementierung bedürfen.

Des Weiteren könnte eine Funktion zur Echtzeitüberwachung von Bestellungen und Lagerbeständen entwickelt werden, um Administratoren sofort zu benachrichtigen, wenn Produkte knapp werden oder Bestellungen nicht wie erwartet abgewickelt werden. Hierbei könnten WebSockets oder ähnliche Technologien verwendet werden, um Benachrichtigungen in Echtzeit zu ermöglichen. Auch bei dieser Funktionalität müssen Sicherheitsaspekte beachtet werden. So darf die Echtzeitüberwachung keine vertraulichen Informationen preisgeben, die von unbefugten Benutzern eingesehen werden könnten. Eine unsichere Implementierung

könnte zum Beispiel dazu führen, dass sensitive Daten in Echtzeit an den falschen Benutzer übertragen werden.

Langfristig kann die Anwendung somit als Grundlage für Schulungs- und Trainingsprogramme dienen, um Entwickler und Sicherheitsexperten im Umgang mit Schwachstellen in Webanwendungen zu schulen. Durch die gezielte Integration neuer Sicherheitslücken oder Szenarien könnten spezifische Angriffsmethoden simuliert werden, um effektive Gegenmaßnahmen zu entwickeln. Gleichzeitig könnten Sicherheitsfeatures in der Anwendung Schritt für Schritt implementiert werden, um deren Wirkung in der Praxis zu testen.